Version 1.1

xHarbour's
Language Reference Guide

The complete reference guide for the professional xHarbour developer

© 2006 - 2007 xHarbour.com Inc. All rights reserved.
http://www.xHarbour.com

Copyright

© 2006-2007 xHarbour.com Inc. All rights reserved.

xHarbour Language Reference Guide version 1.1.

If this guide is distributed with software that includes an end user agreement, this guide, as well as the software described in it, is furnished under license and may be used or copied only in accordance with the terms of such license. Except as permitted by any such license, no part of this guide may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, recording, or otherwise, without the prior written permission of xHarbour.com Inc. Please note that the content in this guide is protected under copyright law even if it is not distributed with software that includes an end user license agreement.

The content of this guide is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by xHarbour.com Inc. xHarbour.com Inc. assumes no responsibility or liability for any errors or inaccuracies that may appear in the informational content contained in this guide.

xHarbour.com Inc. 10624 S. Eastern Avenue, NV 89052 Henderson, USA.

Table of Contents

Contents

Table of Contents I

Read Me First! 1

Class Reference (textmode) 3

C Structure class 4
Error() 10
Get() 17
HbCheckBox() 34
HbListBox() 40
HBObject() 55
HBPersistent() 60
HbPushButton() 63
HbRadioButton() 69
HbRadioGroup() 76
HbScrollBar() 87
MenuItem() 92
Popup() 96
TBColumn() 105
TBrowse() 110
TopBarMenu() 131
THtmlDocument() 141
THtmlIterator() 148
THtmlIteratorRegEx() 151
THtmlIteratorScan() 153
THtmlNode() 155
TIpClient() 174
TIpClientFtp() 180
TIpClientHttp() 188
TIpClientPop() 191
TIpClientSmtp() 196
TIpMail() 201
TStream() 211
TStreamFileReader() 213
TStreamFileWriter() 218
TUrl() 222
TXmlDocument() 227
TXmlIterator() 236
TXmlIteratorRegEx() 239
TXmlIteratorScan() 242
TXmlNode() 244
Win32Bmp() 254
Win32Prn() 257

Command Reference 290

? | ?? 291
@...BOX 293
@...CLEAR 295
@...GET 296

@...GET CHECKBOX 300
@...GET LISTBOX 303
@...GET PUSHBUTTON 306
@...GET RADIOGROUP 310
@...GET TBROWSE 313
@...PROMPT 317
@...SAY 318
@...TO 321
ACCEPT 322
APPEND BLANK 323
APPEND FROM 325
AVERAGE 329
CANCEL 331
CLEAR ALL 332
CLEAR GETS 333
CLEAR MEMORY 334
CLEAR SCREEN 335
CLEAR TYPEAHEAD 336
CLOSE 337
CLOSE LOG 339
COMMIT 340
CONTINUE 341
COPY FILE 342
COPY STRUCTURE 343
COPY STRUCTURE EXTENDED 344
COPY TO 346
COUNT 350
CREATE 351
CREATE FROM 353
DEFAULT TO 355
DELETE 356
DELETE FILE 358
DELETE TAG 359
DO 361
EJECT 362
ERASE 363
FIND 364
GO 365
INDEX 366
INIT LOG 370
INPUT 374
JOIN 375
KEYBOARD 377
LIST 379
LOCATE 381
LOG 383
MENU TO 384
PACK 386
QUIT 387
READ 388
RECALL 390
REINDEX 392
RELEASE 393
RENAME 395

REPLACE 397
RESTORE 399
RUN 401
SAVE 402
SEEK 403
SELECT 405
SET ALTERNATE 407
SET AUTOPEN 409
SET AUTORDER 411
SET AUTOSHARE 412
SET BACKGROUND TASKS 413
SET BACKGROUNDTICK 414
SET BELL 415
SET CENTURY 416
SET COLOR 417
SET CONFIRM 418
SET CONSOLE 419
SET CURSOR 420
SET DATE 421
SET DBFLOCKSCHEME 423
SET DECIMALS 425
SET DEFAULT 426
SET DELETED 427
SET DELIMITERS 429
SET DESCENDING 430
SET DEVICE 431
SET DIRCASE 432
SET DIRSEPARATOR 433
SET EOL 434
SET EPOCH 435
SET ERRORLOG 436
SET ERRORLOOP 437
SET ESCAPE 438
SET EVENTMASK 439
SET EXACT 441
SET EXCLUSIVE 443
SET FILECASE 444
SET FILTER 445
SET FIXED 446
SET FUNCTION 447
SET HARDCOMMIT 449
SET INDEX 450
SET INTENSITY 452
SET KEY 453
SET LOG STYLE 454
SET MARGIN 456
SET MEMOBLOCK 457
SET MESSAGE 459
SET OPTIMIZE 460
SET ORDER 461
SET PATH 463
SET PRINTER 464
SET RELATION 466
SET SCOPE 468

SET SCOPEBOTTOM 470
SET SCOPETOP 471
SET SCOREBOARD 472
SET SOFTSEEK 473
SET STRICTREAD 475
SET TIME 476
SET TRACE 478
SET TYPEAHEAD 479
SET UNIQUE 480
SET VIDEOMODE 481
SET WRAP 482
SKIP 483
SORT 485
STORE 487
SUM 488
TEXT 490
TOTAL 491
UNLOCK 493
UPDATE 494
USE 495
WAIT 497
ZAP 498
Function Reference 499
AAdd() 500
Abs() 501
AChoice() 502
AClone() 507
ACopy() 508
ACos() 510
AddASCII() 511
AddMonth() 512
ADel() 513
ADir() 515
AEval() 517
AFields() 519
AFill() 521
AfterAtNum() 523
AIns() 524
ALenAlloc() 526
Alert() 527
Alias() 529
AllTrim() 530
AltD() 531
AmPm() 533
AnsiToHtml() 534
Array() 537
Asc() 538
AScan() 539
ASCIISum() 541
AscPos() 542
ASin() 543
ASize() 544
ASizeAlloc() 545

ASort() 546
At() 548
AtAdjust() 550
ATail() 552
ATan() 553
ATn2() 554
AtNum() 555
AtRepl() 556
AtSkipStrings() 557
AtToken() 559
BeforAtNum() 560
Bin2I() 561
Bin2L() 562
Bin2U() 563
Bin2W() 564
BitToC() 565
Blank() 566
BlobDirectExport() 567
BlobDirectGet() 569
BlobDirectImport() 571
BlobDirectPut() 573
BlobExport() 575
BlobGet() 577
BlobImport() 578
BlobRootDelete() 580
BlobRootGet() 581
BlobRootLock() 582
BlobRootPut() 583
BlobRootUnlock() 585
BoF() 586
BoM() 588
BoQ() 589
BoY() 590
Break() 591
Browse() 593
CallDll() 595
CDoW() 596
Ceiling() 597
Celsius() 598
Center() 599
CFTSAdd() 601
CFTSClose() 602
CFTSCrea() 603
CFTSDelete() 604
CFTSIfDel() 605
CFTSNext() 606
CFTSOpen() 607
CFTSRecn() 608
CFTSReplac() 609
CFTSSet() 610
CFTSUndel() 611
CFTSVeri() 612
CFTSVers() 613
CharAdd() 614

CharAND() 615
CharEven() 616
CharHist() 617
CharList() 619
CharMirr() 620
CharMix() 621
CharNoList() 623
CharNOT() 624
CharOdd() 625
CharOne() 626
CharOnly() 627
CharOR() 628
CharPack() 629
CharRela() 630
CharRelRep() 631
CharRem() 632
CharRepl() 633
CharRLL() 634
CharRLR() 635
CharSHL() 636
CharSHR() 637
CharSList() 638
CharSort() 639
CharSpread() 641
CharSub() 642
CharSwap() 643
CharUnpack() 644
CharWin() 645
CharXOR() 646
CheckSum() 647
Chr() 648
ClearBit() 650
ClearEol() 651
ClearSlow() 652
ClearWin() 653
ClEol() 654
ClWin() 655
CMonth() 656
Col() 658
ColorRepl() 659
ColorSelect() 660
ColorToN() 662
ColorWin() 663
Complement() 664
ConvToAnsiCP() 665
ConvToOemCP() 666
Cos() 667
CosH() 668
Cot() 669
CountGets() 670
CountLeft() 671
CountRight() 672
CreateObject() 673
Crypt() 679

CSetAtMuPa() ... 680
CSetCent() ... 681
CSetCurs() ... 682
CSetKey() ... 683
CSetRef() ... 684
CSetSafety() ... 686
CStr() ... 687
CStrToVal() ... 688
CtoBit() ... 689
CtoD() ... 690
CtoDoW() ... 691
CtoF() ... 692
CtoMonth() ... 693
CtoN() ... 694
CtoT() ... 695
CurDir() ... 696
CurDirX() ... 697
CurDrive() ... 698
CurrentGet() ... 699
Date() ... 700
DateTime() ... 701
Day() ... 702
Days() ... 703
DaysInMonth() ... 704
DaysToMonth() ... 705
DbAppend() ... 706
DbClearFilter() ... 708
DbClearIndex() ... 709
DbClearRelation() ... 710
DbCloseAll() ... 711
DbCloseArea() ... 712
DbCommit() ... 713
DbCommitAll() ... 714
DbCopyExtStruct() ... 715
DbCopyStruct() ... 716
DbCreate() ... 717
DbCreateIndex() ... 719
DbDelete() ... 721
DbEdit() ... 722
DbEval() ... 727
Dbf() ... 729
DbFieldInfo() ... 730
DbFileGet() ... 732
DbFilePut() ... 733
DbFilter() ... 734
DbfSize() ... 735
DbGoBottom() ... 736
DbGoto() ... 737
DbGoTop() ... 739
DbInfo() ... 740
DbJoin() ... 746
Dblist() ... 747
DbOrderInfo() ... 749
DbRecall() ... 755

DbRecordInfo() 756
DbReindex() 757
DbRelation() 758
DbRLock() 760
DbRLockList() 762
DbRSelect() 763
DbRUnlock() 765
DbSeek() 766
DbSelectArea() 768
DbSetDriver() 770
DbSetFilter() 771
DbSetIndex() 773
DbSetOrder() 774
DbSetRelation() 775
DbSkip() 777
DbSkipper() 778
DbSort() 780
DbStruct() 782
DbTableExt() 784
DbTotal() 785
DbUnlock() 787
DbUnlockAll() 788
DbUpdate() 789
DbUseArea() 790
Default() 792
DefPath() 793
Deleted() 794
DeleteFile() 795
Descend() 796
DevOut() 798
DevOutPict() 799
DevPos() 800
DirChange() 801
Directory() 802
DirectoryRecurse() 804
DirMake() 806
DirName() 807
DirRemove() 808
DisableWaitLocks() 809
DiskChange() 810
DiskFormat() 812
DiskFree() 813
DiskName() 814
DiskReady() 815
DiskReadyW() 816
DiskSpace() 817
DiskTotal() 819
DiskUsed() 820
DispBegin() 821
DispBox() 823
DispCount() 825
DispEnd() 826
DispOut() 827
DispOutAt() 828

DispOutAtSetPos() 830
DllCall() 831
DllExecuteCall() 834
DllLoad() 836
DllPrepareCall() 837
DllUnload() 839
DMY() 840
DosError() 841
DosParam() 844
DoW() 845
DoY() 846
DriveType() 847
DtoC() 848
DtoR() 849
DtoS() 850
ElapTime() 852
Empty() 853
Enhanced() 855
Eof() 856
EoM() 858
EoQ() 859
EoY() 860
ErrorBlock() 861
ErrorLevel() 863
ErrorNew() 865
ErrorSys() 867
Eval() 868
ExeName() 869
Exp() 870
Expand() 871
Exponent() 872
Fact() 873
Fahrenheit() 874
FCharCount() 875
FClose() 876
FCount() 878
FCreate() 879
FErase() 881
FError() 882
FieldBlock() 883
FieldDec() 885
FieldDeci() 886
FieldGet() 887
FieldLen() 888
FieldName() 889
FieldNum() 890
FieldPos() 891
FieldPut() 892
FieldSize() 893
FieldType() 894
FieldWBlock() 895
File() 897
FileAppend() 899
FileAttr() 900

FileCClose() 902
FileCCont() 903
FileCOpen() 904
FileCopy() 905
FileDate() 907
FileDelete() 908
FileMove() 909
FileReader() 910
FileScreen() 911
FileSeek() 912
FileSize() 914
FileStats() 915
FileStr() 917
FileTime() 918
FileValid() 920
FileWriter() 922
FkLabel() 923
FkMax() 924
FLineCount() 925
FLock() 926
Floor() 928
FloppyType() 929
FOpen() 930
Found() 932
FParse() 934
FParseEx() 935
FParseLine() 936
FRead() 938
FReadStr() 940
FreeLibrary() 942
FRename() 943
FSeek() 945
FtoC() 947
Fv() 948
FWordCount() 949
FWrite() 950
GetActive() 952
GetActiveObject() 953
GetApplyKey() 954
GetClearA() 955
GetClearB() 956
GetClrBack() 957
GetClrFore() 958
GetClrPair() 959
GetCurrentThread() 960
GetDefaultPrinter() 961
GetDoSetKey() 962
GetE() 963
GetEnv() 964
GetFldCol() 965
GetFldRow() 966
GetFldVar() 967
GetLastError() 968
GetNew() 969

GetPairLen()....970
GetPairPos()....971
GetPostValidate()....972
GetPrec()....973
GetPreValidate()....974
GetPrinters()....975
GetProcAddress()....977
GetRegistry()....978
GetSecret()....980
GetSystemThreadID()....982
GetThreadID()....983
GetVolInfo()....985
GuiApplyKey()....986
GuiGetPostValidate()....987
GuiGetPrevalidate()....988
GuiReader()....989
HaaDelAt()....990
HaaGetKeyAt()....991
HaaGetPos()....992
HaaGetRealPos()....994
HaaGetValueAt()....996
HaaSetValueAt()....998
HAllocate()....1000
HardCR()....1001
Hash()....1002
HbConsoleLock()....1003
HbConsoleUnlock()....1004
HB_AExpressions()....1005
HB_AnsiToOem()....1006
HB_AParams()....1008
HB_ArgC()....1009
HB_ArgCheck()....1010
HB_ArgString()....1011
HB_ArgV()....1012
HB_ArrayBlock()....1014
HB_ArrayId()....1015
HB_ArrayToStructure()....1016
HB_ATokens()....1018
HB_AtX()....1020
HB_BackGroundActive()....1022
HB_BackGroundAdd()....1023
HB_BackGroundDel()....1025
HB_BackGroundReset()....1026
HB_BackGroundRun()....1027
HB_BackGroundTime()....1028
HB_Base64Decode()....1029
HB_Base64DecodeFile()....1030
HB_Base64Encode()....1031
HB_Base64EncodeFile()....1032
HB_BitAnd()....1033
HB_BitIsSet()....1034
HB_BitNot()....1035
HB_BitOr()....1036
HB_BitReset()....1037

HB_BitSet() 1039
HB_BitShift() 1041
HB_BitXOr() 1043
HB_BldLogMsg() 1044
HB_BuildDate() 1045
HB_BuildInfo() 1046
HB_CheckSum() 1048
HB_Clocks2Secs() 1049
HB_CloseProcess() 1050
HB_ClrArea() 1051
HB_CmdArgArgV() 1052
HB_ColorIndex() 1053
HB_ColorToN() 1055
HB_Compiler() 1056
HB_Compress() 1057
HB_CompressBufLen() 1059
HB_CompressError() 1060
HB_CompressErrorDesc() 1061
Hb_CRC32() 1062
HB_CreateLen8() 1063
HB_Crypt() 1064
HB_Decode() 1065
HB_DecodeOrEmpty() 1067
HB_Decrypt() 1068
HB_DeserialBegin() 1069
HB_DeSerialize() 1070
HB_DeserializeSimple() 1071
HB_DeserialNext() 1072
HB_DiskSpace() 1073
HB_DumpVar() 1075
HB_EnumIndex() 1076
HB_Exec() 1078
HB_ExecFromArray() 1080
HB_FCommit() 1084
HB_FCreate() 1085
HB_FEof() 1087
HB_FGoBottom() 1088
HB_FGoto() 1089
HB_FGoTop() 1090
HB_FInfo() 1091
HB_FLastRec() 1092
HB_FNameMerge() 1093
HB_FNameSplit() 1094
HB_FReadAndSkip() 1096
HB_FReadLine() 1097
HB_FreadLN() 1099
HB_FRecno() 1100
HB_FSelect() 1101
HB_FSize() 1102
HB_FSkip() 1103
HB_FTempCreate() 1104
HB_FuncPtr() 1106
HB_FUse() 1107
HB_F_Eof() 1111

HB_GCAll() 1112
HB_GCStep() 1113
HB_GetLen8() 1114
HB_IdleAdd() 1115
HB_IdleDel() 1117
HB_IdleReset() 1118
HB_IdleSleep() 1119
HB_IdleSleepMSec() 1120
HB_IdleState() 1121
HB_IdleWaitNoCPU() 1122
HB_IsArray() 1123
HB_IsBlock() 1124
HB_IsByRef 1125
HB_IsDate() 1127
HB_IsDateTime() 1128
HB_IsHash() 1129
HB_IsLogical() 1130
HB_IsMemo() 1131
HB_IsNIL() 1132
HB_IsNull() 1133
HB_IsNumeric() 1134
HB_IsObject() 1135
HB_IsPointer() 1136
HB_IsRegExString() 1137
HB_IsString() 1138
HB_KeyPut() 1139
HB_LangErrMsg() 1140
HB_LangMessage() 1141
HB_LangName() 1143
HB_LangSelect() 1144
HB_LibDo() 1147
HB_LogDateStamp() 1149
HB_MacroCompile() 1150
HB_MD5() 1151
HB_MD5File() 1152
HB_MultiThread() 1153
HB_MutexCreate() 1154
HB_MutexLock() 1157
HB_MutexTimeoutLock() 1158
HB_MutexTryLock() 1159
HB_MutexUnlock() 1160
HB_ObjMsgPtr() 1161
HB_OemToAnsi() 1163
HB_OpenProcess() 1164
HB_OsDriveSeparator() 1167
HB_OsError() 1168
HB_OsNewLine() 1169
HB_OsPathDelimiters() 1170
HB_OsPathListSeparator() 1172
HB_OsPathSeparator() 1173
HB_PCodeVer() 1174
HB_Pointer2String() 1175
HB_ProcessValue() 1176
HB_QSelf() 1177

HB_QWith() 1178
HB_Random() 1179
HB_RandomInt() 1180
HB_RandomSeed() 1181
HB_ReadIni() 1182
HB_ReadLine() 1184
HB_RegEx() 1187
HB_RegExAll() 1190
HB_RegExAtX() 1194
HB_RegExComp() 1196
HB_RegExMatch() 1197
HB_RegExReplace() 1199
HB_RegExSplit() 1201
HB_ResetWith() 1203
HB_RestoreBlock() 1205
HB_SaveBlock() 1206
HB_Serialize() 1207
HB_SerializeSimple() 1209
HB_SerialNext() 1210
HB_SetCodePage() 1211
HB_SetIniComment() 1213
HB_SetKeyArray() 1214
HB_SetKeyCheck() 1215
HB_SetKeyGet() 1216
HB_SetKeySave() 1217
HB_SetMacro() 1219
HB_SetWith() 1220
HB_Shadow() 1222
HB_SizeofCStructure() 1223
HB_String2Pointer() 1225
HB_StructureToArray() 1226
HB_SysLogClose() 1228
HB_SysLogMessage() 1229
HB_SysLogOpen() 1230
HB_ThisArray() 1231
HB_Translate() 1232
HB_Uncompress() 1233
HB_UUDecode() 1235
HB_UUDecodeFile() 1236
HB_UUEncode() 1237
HB_UUEncodeFile() 1238
HB_ValToStr() 1239
HB_VMExecute() 1240
HB_VMMode() 1241
HB_WithObjectCounter() 1242
HB_WriteIni() 1243
HB_XmlErrorDesc() 1245
HClone() 1246
HCopy() 1248
HDel() 1250
HDelAt() 1251
Header() 1252
HEval() 1253
HexToNum() 1254

HexToStr() 1255
HFill() 1256
HGet() 1257
HGetAACompatibility() 1258
HGetAutoAdd() 1259
HGetCaseMatch() 1260
HGetKeyAt()....1262
HGetKeys() 1263
HGetPairAt() 1264
HGetPartition()....1265
HGetPos()....1266
HGetVaaPos() 1267
HGetValueAt()....1268
HGetValues() 1269
HHasKey() 1270
HMerge()....1271
Hour()....1272
HScan() 1273
HSet() 1275
HSetAACompatibility() 1276
HSetAutoAdd() 1277
HSetCaseMatch() 1278
HSetPartition() 1280
HSetValueAt() 1282
HS_Add() 1283
HS_Close()....1285
HS_Create() 1286
HS_Delete() 1289
HS_Filter() 1291
HS_IfDel() 1293
HS_Index()....1294
HS_KeyCount() 1297
HS_Next() 1298
HS_Open() 1300
HS_Replace() 1302
HS_Set()....1304
HS_Undelete() 1305
HS_Verify() 1306
HS_Version() 1308
HtmlToAnsi()....1309
HtmlToOem()....1310
I2Bin()....1311
If() | IIf()....1313
IndexExt() 1315
IndexKey() 1316
IndexOrd()....1317
INetAccept()....1318
INetAddress()....1319
INetCleanup()....1320
INetClearError()....1321
INetClearPeriodCallback()....1322
INetClearTimeout()....1323
INetClose()....1324
INetConnect()....1325

INetConnectIP() ... 1327
INetCount() ... 1328
INetCreate() ... 1330
INetCRLF() ... 1331
INetDataReady() ... 1332
INetDGram() ... 1333
INetDGramBind() ... 1334
INetDGramRecv() ... 1335
INetDGramSend() ... 1336
INetErrorCode() ... 1340
INetErrorDesc() ... 1341
INetGetAlias() ... 1342
INetGetHosts() ... 1343
INetGetPeriodCallback() ... 1344
INetGetTimeout() ... 1345
INetInit() ... 1346
INetPort() ... 1347
INetRecv() ... 1348
INetRecvAll() ... 1350
INetRecvEndblock() ... 1351
INetRecvLine() ... 1352
INetSend() ... 1353
INetSendAll() ... 1355
INetServer() ... 1356
INetSetPeriodCallback() ... 1360
INetSetTimeout() ... 1362
Infinity() ... 1363
Inkey() ... 1364
Int() ... 1366
InvertAttr() ... 1367
InvertWin() ... 1368
IsAffirm() ... 1369
IsAlNum() ... 1370
IsAlpha() ... 1371
IsAscii() ... 1372
IsBit() ... 1373
IsCntrl() ... 1374
IsColor() ... 1375
IsDefColor() ... 1376
IsDigit() ... 1377
IsDir() ... 1378
IsDirectory() ... 1379
IsDisk() ... 1380
IsGraph() ... 1382
IsLeap() ... 1383
IsLocked() ... 1384
IsLower() ... 1385
IsNegative() ... 1386
IsPrint() ... 1387
IsPrinter() ... 1388
IsPunct() ... 1389
IsSameThread() ... 1390
IsSpace() ... 1391
IsUpper() ... 1392

IsValidThread() 1393
IsXDigit() 1394
JoinThread() 1395
JustLeft() 1397
JustRight() 1398
KbdStat() 1399
KeySec() 1400
KeyTime() 1401
KillAllThreads() 1402
KillThread() 1403
KSetCaps() 1404
KSetIns() 1405
KSetNum() 1406
KSetScroll() 1407
L2bin() 1408
LastDayoM() 1410
LastKey() 1411
LastRec() 1413
Left() 1414
Len() 1415
LenNum() 1416
LibFree() 1417
LibLoad() 1418
LoadLibrary() 1419
Log() 1421
Log10() 1422
Lower() 1423
LtoC() 1424
LtoN() 1425
LTrim() 1426
LUpdate() 1427
MakeDir() 1428
Mantissa() 1430
Max() 1431
MaxCol() 1432
MaxLine() 1433
MaxRow() 1434
MCol() 1435
MDblClk() 1436
MDY() 1437
MemoEdit() 1438
MemoLine() 1445
MemoRead() 1448
Memory() 1449
MemoTran() 1450
MemoWrit() 1451
MemVarBlock() 1452
MenuModal() 1453
MHide() 1455
MilliSec() 1456
Min() 1457
Minute() 1458
MLCount() 1459
MLCToPos() 1461

MLeftDown() ... 1463
MlPos() ... 1464
Mod() ... 1466
Month() ... 1467
MPosToLC() ... 1468
MPresent() ... 1470
MRestState() ... 1471
MRightDown() ... 1472
MRow() ... 1473
MSaveState() ... 1474
MSetBounds() ... 1475
MSetCursor() ... 1476
MSetPos() ... 1477
MShow() ... 1478
NationMsg() ... 1479
NetAppend() ... 1481
NetCancel() ... 1482
NetCommitAll() ... 1483
NetDbUse() ... 1484
NetDelete() ... 1486
NetDisk() ... 1487
NetErr() ... 1488
NetError() ... 1490
NetFileLock() ... 1491
NetFunc() ... 1492
NetLock() ... 1493
NetName() ... 1494
NetOpenFiles() ... 1495
NetPrinter() ... 1496
NetRecall() ... 1497
NetRecLock() ... 1498
NetRedir() ... 1499
NetRmtName() ... 1500
Network() ... 1501
NextKey() ... 1502
NNetwork() ... 1504
Notify() ... 1505
NotifyAll() ... 1506
NtoC() ... 1508
NtoCDoW() ... 1509
NtoCMonth() ... 1510
NtoColor() ... 1511
Nul() ... 1512
NumAND() ... 1513
NumAndX() ... 1514
NumAt() ... 1515
NumButtons() ... 1516
NumCount() ... 1517
NumDiskL() ... 1518
NumHigh() ... 1519
NumLine() ... 1520
NumLow() ... 1521
NumMirr() ... 1522
NumMirrX() ... 1523

NumNOT() 1524
NumNotX() 1525
NumOR() 1526
NumOrX() 1527
NumRoL() 1528
NumRolX() 1530
NumToHex() 1531
NumToken() 1532
NumXOR() 1533
NumXorX() 1534
Occurs() 1535
OemToHtml() 1536
Ole2TxtError() 1537
OleError() 1538
OrdBagExt() 1539
OrdBagName() 1540
OrdCondSet() 1542
OrdCount() 1545
OrdCreate() 1547
OrdCustom() 1549
OrdDescend() 1550
OrdDestroy() 1552
OrdFindRec() 1553
OrdFor() 1555
OrdIsUnique() 1556
OrdKey() 1557
OrdKeyAdd() 1559
OrdKeyCount() 1561
OrdKeyDel() 1563
OrdKeyGoTo() 1565
OrdKeyNo() 1567
OrdKeyRelPos() 1569
OrdKeyVal() 1571
OrdListAdd() 1572
OrdListClear() 1573
OrdListRebuild() 1574
OrdName() 1575
OrdNumber() 1577
OrdScope() 1579
OrdSetFocus() 1581
OrdSetRelation() 1583
OrdSkipRaw() 1584
OrdSkipUnique() 1585
OrdWildSeek() 1587
Os() 1589
Os_IsWin2000() 1590
Os_IsWin2000_Or_Later() 1591
Os_IsWin2003() 1592
Os_IsWin95() 1593
Os_IsWin98() 1594
Os_IsWin9X() 1595
Os_IsWinME() 1596
Os_IsWinNT() 1597
Os_IsWinNT351() 1598

Os_IsWinNT4() 1599
Os_IsWinVista() 1600
Os_IsWinXP() 1601
Os_IsWtsClient() 1602
Os_NetRegOk() 1603
OS_NetVRedirOk() 1604
Os_VersionInfo() 1605
OutErr() 1606
OutStd() 1607
PadC() | PadL() | PadR() 1608
PadLeft() 1610
PadRight() 1611
Payment() 1612
PCol() 1613
PCount() 1615
Periods() 1616
Pi() 1617
PosAlpha() 1618
PosChar() 1619
PosDel() 1621
PosDiff() 1622
PosEqual() 1623
PosIns() 1624
PosLower() 1625
PosRange() 1626
PosRepl() 1627
PosUpper() 1628
PrgExpToVal() 1629
PrinterExists() 1630
PrinterPortToName() 1631
PrintFileRaw() 1632
PrintReady() 1634
PrintSend() 1635
PrintStat() 1636
ProcFile() 1637
ProcLine() 1638
ProcName() 1639
PRow() 1640
Pv() 1641
PValue() 1642
QOut() | QQOut() 1643
Quarter() 1645
QueryRegistry() 1646
RangeRem() 1648
RangeRepl() 1649
RAscan() 1650
RAt() 1652
Rate() 1654
RddInfo() 1655
RddList() 1657
RddName() 1659
RddRegister() 1660
RddSetDefault() 1661
ReadExit() 1662

ReadInsert() 1663
ReadKey() 1664
ReadKill() 1665
ReadModal() 1666
ReadUpdated() 1668
ReadVar() 1669
RecCount() 1670
RecNo() 1671
RecSize() 1672
RemAll() 1673
RemLeft() 1674
RemRight() 1675
RenameFile() 1676
ReplAll() 1677
Replicate() 1678
ReplLeft() 1679
ReplRight() 1680
RestCursor() 1681
RestGets() 1682
RestScreen() 1683
RestSetKey() 1685
RestToken() 1686
Right() 1687
RLock() 1688
Round() 1689
Row() 1691
RtoD() 1692
RTrim() 1693
SaveCursor() 1694
SaveGets() 1695
SaveScreen() 1696
SaveSetKey() 1698
SaveToken() 1699
SayDown() 1700
SayMoveIn() 1701
SayScreen() 1702
SaySpread() 1703
ScreenAttr() 1704
ScreenFile() 1705
ScreenMark() 1706
ScreenMix() 1708
ScreenStr() 1709
Scroll() 1710
Scrollfixed() 1712
Seconds() 1713
SecondsCpu() 1714
SecondsSleep() 1715
Secs() 1716
SecToTime() 1717
Select() 1718
Set() 1719
SetAtLike() 1726
SetBit() 1727
SetBlink() 1728

SetCancel() 1729
SetClearA() 1731
SetClearB() 1732
SetClrPair() 1733
SetColor() 1734
SetCursor() 1736
SetDate() 1738
SetErrorMode() 1739
SetFAttr() 1740
SetFCreate() 1741
SetFDaTi() 1742
SetKey() 1743
SetLastError() 1746
SetLastKey() 1747
SetMode() 1748
SetMouse() 1750
SetNetDelay() 1751
SetNetMsgColor() 1752
SetNewDate() 1753
SetNewTime() 1754
SetPos() 1755
SetPosBS() 1756
SetPrc() 1757
SetPrec() 1758
SetRegistry() 1759
SetTime() 1761
ShowTime() 1762
Sign() 1764
Sin() 1765
SinH() 1766
SoundEx() 1767
Space() 1768
Sqrt() 1769
Standard() 1770
StartThread() 1771
StoD() 1774
StopThread() 1775
StoT() 1776
Str() 1777
Strdel() 1779
StrDiff() 1780
StrFile() 1781
StringToLiteral() 1783
StrPeek() 1784
StrPoke() 1785
StrScreen() 1786
StrSwap() 1787
StrToHex() 1788
StrTran() 1789
StrZero() 1791
Stuff() 1793
Subscribe() 1795
SubscribeNow() 1797
SubStr() 1798

SX_Decrypt() 1799
SX_DtoP() 1800
SX_Encrypt() 1801
SX_FCompress() 1802
SX_FDecompress() 1803
SX_PtoD() 1804
TabExpand() 1805
TabPack() 1806
Tan() 1807
TanH() 1808
TBMouse() 1809
TBrowseDB() 1810
TBrowseNew() 1812
ThreadSleep() 1813
Throw() 1814
THtmlCleanup() 1815
THtmlInit() 1816
THtmlIsValid() 1817
Time() 1818
TimeToSec() 1819
TimeValid() 1820
Token() 1821
TokenAt() 1823
TokenEnd() 1824
TokenExit() 1825
TokenInit() 1826
TokenLower() 1830
TokenNext() 1831
TokenNum() 1832
TokenSep() 1833
TokenUpper() 1834
Tone() 1835
TraceLog() 1836
Transform() 1837
Trim() 1839
TrueName() 1840
TString() 1841
TtoC() 1842
TtoS() 1843
Type() 1844
U2bin() 1846
Unselected() 1848
UntextWin() 1849
Updated() 1850
Upper() 1851
Used() 1852
UsrRdd_AreaData() 1853
UsrRdd_AreaResult() 1854
UsrRdd_ID() 1855
UsrRdd_RddData() 1856
UsrRdd_SetBof() 1857
UsrRdd_SetBottom() 1858
UsrRdd_SetEof() 1859
UsrRdd_SetFound() 1860

UsrRdd_SetTop() 1861
Val() 1862
ValPos() 1864
ValToPrg() 1865
ValToPrgExp() 1866
Valtype() 1867
Version() 1869
VolSerial() 1870
Volume() 1871
W2bin() 1872
WAClose() 1873
WaitForThreads() 1874
WaitPeriod() 1875
WBoard() 1876
WBox() 1877
WCenter() 1879
WClose() 1880
WCol() 1881
Week() 1882
WfCol() 1883
WfLastCol() 1884
WfLastRow() 1885
WFormat() 1886
WfRow() 1887
Wild2RegEx() 1888
WildMatch() 1890
WInfo() 1892
WLastCol() 1894
WLastRow() 1895
WMode() 1896
WMove() 1897
WMSetPos() 1898
WNum() 1899
WoM() 1900
WOpen() 1901
Word() 1904
WordOne() 1905
WordOnly() 1906
WordRem() 1907
WordRepl() 1908
WordSwap() 1909
WordToChar() 1910
WRow() 1911
WSelect() 1912
WSetMouse() 1913
WSetMove() 1914
WSetShadow() 1915
WStack() 1916
WStep() 1917
XtoC() 1918
Year() 1919
Operator Reference 1920
$ 1921

% 1922
& (bitwise AND)....1923
& (macro operator)....1925
( ) 1930
* 1932
** 1933
+ 1934
++ 1936
- 1937
-- 1939
-> 1940
.AND. 1942
.NOT. 1943
.OR. 1944
: 1946
:= 1948
< 1949
<< 1951
<= 1952
<> != # 1954
<|| > 1956
= (assignment)....1958
= (comparison) 1959
= (compound assignment) 1961
== 1963
> 1964
>= 1966
>> 1968
@ 1969
@() 1970
HAS....1972
IN 1974
LIKE 1976
[ ] (array) 1978
[ ] (string) 1979
^^ 1981
{ } 1983
{=>} 1984
{^ } 1986
{|| } 1988
| (bitwise OR) 1990
Preprocessor Reference 1992
#command | #translate....1993
#define....1999
#error....2002
#if 2003
#ifdef....2005
#ifndef....2006
#include....2007
#pragma....2009
#stdout....2011
#uncommand | #untranslate....2012
#undef 2013

#xcommand | #xtranslate 2014
#xuncommand | #xuntranslate 2015
Statement Reference 2016
(struct) 2017
ACCESS 2018
ANNOUNCE 2020
ASSIGN 2021
ASSOCIATE CLASS 2024
BEGIN SEQUENCE 2028
CLASS 2030
CLASSDATA 2035
CLASSMETHOD 2038
DATA 2039
DELEGATE 2041
DESTRUCTOR 2043
DO CASE 2045
DO WHILE 2047
ENABLE TYPE CLASS 2049
ERROR HANDLER 2051
EXIT PROCEDURE 2053
EXPORTED: 2055
EXTEND CLASS...WITH DATA 2056
EXTEND CLASS...WITH METHOD 2058
EXTERNAL 2060
FIELD 2061
FOR 2063
FOR EACH 2065
FUNCTION 2067
GLOBAL 2070
HIDDEN: 2072
IF 2074
INIT PROCEDURE 2076
INLINE METHOD 2077
LOCAL 2079
MEMVAR 2081
MESSAGE 2082
METHOD (declaration) 2084
METHOD (implementation) 2087
METHOD...OPERATOR 2090
METHOD...VIRTUAL 2092
OPERATOR 2093
OVERRIDE METHOD 2095
PARAMETERS 2097
pragma pack() 2098
PRIVATE 2099
PROCEDURE 2100
PROTECTED: 2102
PUBLIC 2103
REQUEST 2105
RETURN 2106
STATIC 2107
SWITCH 2109
TRY...CATCH 2111

typedef struct 2113
VAR 2116
WITH OBJECT 2118
Categories 2120
Array functions 2120
Assignment operators 2120
Associative arrays 2120
Background processing 2120
Binary functions 2121
Bitwise functions 2121
Bitwise operators 2122
Blob functions 2122
C Structure support 2122
CFTS functions 2122
Character functions 2123
Character operators 2126
Checksum functions 2126
Class declaration 2126
Code block functions 2126
Comparison operators 2127
Console commands 2127
Control structures 2127
Conversion functions 2127
CT:Database 2129
CT:DateTime 2129
CT:DiskUtil 2129
CT:GetSys 2130
CT:Math 2130
CT:Miscellaneous 2131
CT:Network 2131
CT:NumBits 2131
CT:Printer 2132
CT:Settings 2132
CT:String manipulation 2132
CT:Video 2134
CT:Window 2135
Database commands 2135
Database drivers 2136
Database functions 2137
Datagram functions 2139
Date and time 2139
Debug functions 2141
Declaration 2141
Directory functions 2142
Disks and Drives 2143
DLL functions 2143
Encoding/Decoding 2143
Environment commands 2144
Environment functions 2144
Error functions 2145
Field functions 2145
File commands 2146
File functions 2146

Financial functions 2147
Get system 2147
Hash functions 2148
HiPer-SEEK functions 2149
HTML functions 2149
Index commands 2150
Index functions 2150
Indirect execution 2151
Info functions 2151
Input commands 2151
Internet functions 2152
Keyboard functions 2152
Language specific 2153
Log commands 2153
Log functions 2153
Logical functions 2153
Logical operators 2154
Low level file functions 2154
Mathematical functions 2154
Mathematical operators 2155
Memo functions 2155
Memory commands 2156
Miscellaneous functions 2156
Mouse functions 2156
Multi-threading functions 2156
Mutex functions 2157
Network functions 2157
New topics in this help file 2158
Numbers and Bits 2168
Numeric functions 2169
Object functions 2169
OLE Automation 2170
Operators 2170
Output commands 2171
Output functions 2172
Pointer functions 2172
Preprocessor directives 2172
Printer commands 2172
Printer functions 2172
Process functions 2173
Random generator 2173
Registry functions 2173
Regular expressions 2173
Screen functions 2173
Serialization functions 2175
SET commands 2175
Six driver 2176
Sockets functions 2176
Special operators 2177
Statements 2177
Text file functions 2177
Token functions 2178
Trigonometric functions 2178
UI functions 2178

User-defined RDD 2179
Windows (text mode) 2179
xHarbour extensions 2179
ZIP compression 2189
Index 2190

Read Me First!

Welcome to the *xHarbour Language Reference Guide*. This page informs you about what you can expect from this *xHarbour Language Reference Guide*.

First, we'd like to thank you for patience and ongoing support while we were preparing this *xHarbour Language Reference Guide*. It turned out to be a huge documentation project! We are confident that the large amount of quality information will prove valuable for every xHarbour programmer.

What's in the xHarbour Language Reference Guide?

As xHarbour's language is based on the Clipper programming language, we started with documenting xHarbour's core Clipper language compatible topics. All of the documented Clipper language topics are written from scratch. None of it is taken or copied from the original Clipper documentation.

During the second stage, we focused on documenting all xHarbour's extensions to the Clipper language. There is a wealth of new features added to xHarbour that do not exist in Clipper (see "Notes for Clipper programmers" below). For example, a complete OOP model with new classes exists. New data types and operators are added. Additional preprocessor directives are available, plus numerous new functions and statements. As a matter of fact, the core documentation of xHarbour is about twice as much as the original Clipper Norton Guides.

It is structured into the following sections:

1. Operator Reference
2. Statement Reference
3. Command Reference
4. Class Reference (textmode)
5. Function Reference
6. Preprocessor Reference

These sections should aid you in finding information on core features of xHarbour easily. In addition, a comprehensive **Index** and **Full Text Search** capabilities, plus a **sophisticated Cross Reference Link System** are all included. Please try these features for searching and finding the information you need in the xHarbour Documentation.For users coming from Version 1.0 of the *xHarbour Language Reference Guide*, we constructed a list of new topics since version 1.0 as wel as a list of changed topics since version 1.0.

Notes for Clipper programmers

The xHarbour programming language has its roots in CA-Clipper, a powerful xBase dialect, and is modelled around Clipper. xHarbour extends Clipper and has received many new features, which can be detected very easily:

Extended function arguments are highlighted in yellow.

Refer to function AIns() for an example.

All topics that do not exist in Clipper, but are added by xHarbour, are listed in the category xHarbour extensions.

The xHarbour Language Reference Guide Online

We have made great effort to put all of the information from the xHarbour Language Reference Guide available online at xHarbour.com. The online version of the xHarbour Language Reference Guide can be found on the xHDN (xHarbour Developers Network) pages at http://www.xHarbour.com/xHDN. Because of the 'online' nature, this edition has a higher update frequency than the offline edition.

Copyright

Please read the Copyright note carefully before using this guide.

Your contributions

This first version of the *xHarbour Language Reference Guide* can only reflect the current stage of xHarbour and its documentation, as of June 2007. We are constantly adding new features to xHarbour and new information to the documentation. If you miss something particular, please let us know.

This file should give you a clear idea of xHarbour.com commitment to the xHarbour programming language. We are grateful for any suggestion you may have to improve xHarbour.

Thank you for your support.

Class Reference (textmode)

C Structure class

Abstract class for C structure support.

Description

xHarbour supports C structures as they are required by many external API functions, such as the Windows API, or Third Party libraries. The C Structure class is an abstract class serving as super class for all structures declared with the typedef struct declaration. Instance variables and methods of this class are available in all declared structures.

On the PRG level, a structure is represented by an object whose instance variables hold the values of structure members. On the C level it is a contiguous binary character string holding the structure members. This character string is maintained by a structure object in an internal buffer which can be retrieved with the :value() method, or changed with the :buffer() method.

In contrast to other classes, C structures are not instantiated with a :new() or :init() method, but with the structure creation statement (struct).

Predefined structures in xHarbour

The #include file Winapi.ch contains a large number of predefined structures that are required by Windows API functions. If a Windows API function is called on the PRG level via DllCall() that requires a structure as parameter, the corresponding structure declaration can be done by simply including the Winapi.ch file. The following table lists the structures that are available when WinApi.ch is included:

Structures for Windows API functions

Declared	Structure	Names
CHOOSEFONT	NMHEADER	RECT
CLIENTCREATESTRUCT	NMPGCALCSIZE	SCROLLBARINFO
COLORSCHEME	NMPGSCROLL	SCROLLINFO
CREATESTRUCT	NMREBARCHEVRON	SIZE
DLGITEMTEMPLATE	NMREBAR	TBADDBITMAP
DLGTEMPLATEEX	NMTBCUSTOMDRAW	TBBUTTON
DLGTEMPLATE	NMTBHOTITEM	TBBUTTONINFOA
DRAWITEMSTRUCT	NMTOOLBARA	TCITEMA
HDITEM	NMTREEVIEWA	TEXTMETRIC
ICONINFO	NMTVCUSTOMDRAW	TOOLINFOA
LITEM	NMTVGETINFOTIP	TOOLTIPTEXT
LOGFONT	NMTVKEYDOWN	TRACKMOUSEEVENT
LVCOLUMNA	NONCLIENTMETRICS	TVHITTESTINFO
LVITEMA	OPENFILENAME	TVINSERTSTRUCT
MDINEXTMENU	OSVERSIONINFO	TVITEMEX
MEASUREITEMSTRUCT	OSVERSIONINFOEX	TVITEM
MENUINFO	PAINTSTRUCT	WINDOWPLACEMENT
MENUITEMINFO	POINTS	WINDOWPOS
MSG	POINT	WNDCLASSEX
NCCALCSIZE_PARAMS	RBHITTESTINFO	WNDCLASS
NMCUSTOMDRAW	REBARBANDINFOA	
NMHDR	REBARINFO	

Instance variables

:aCMembers

Array holding the structure member names.

Data type: A (READONLY)
Default: NIL

Description

The instance variable :aCMembers contains a one dimensional array. Its elements contain character strings with the names of structure members.

:aCTypes

Array holding the data types of structure members.

Data type: A (READONLY)
Default: NIL

Description

The instance variable :aCTypes contains a one dimensional array. Its elements contain numeric values encoding the C data types of the structure members. Constants listed in CStruct.ch are used for data type encoding:

C data types for structure members

Constant	Value
CTYPE_CHAR	1
CTYPE_UNSIGNED_CHAR	-1
CTYPE_CHAR_PTR	10
CTYPE_UNSIGNED_CHAR_PTR	-10
CTYPE_SHORT	2
CTYPE_UNSIGNED_SHORT	-2
CTYPE_SHORT_PTR	20
CTYPE_UNSIGNED_SHORT_PTR	-20
CTYPE_INT	3
CTYPE_UNSIGNED_INT	-3
CTYPE_INT_PTR	30
CTYPE_UNSIGNED_INT_PTR	-30
CTYPE_LONG	4
CTYPE_UNSIGNED_LONG	-4
CTYPE_LONG_PTR	40
CTYPE_UNSIGNED_LONG_PTR	-40
CTYPE_FLOAT	5
CTYPE_FLOAT_PTR	50
CTYPE_DOUBLE	6
CTYPE_DOUBLE_PTR	60
CTYPE_VOID_PTR	7
CTYPE_STRUCTURE	1000
CTYPE_STRUCTURE_PTR	10000

:nAlign

Byte alignment of structure members

Data type: N (READONLY)
Default: NIL

Description

The instance variable :nAlign contains a numeric value used for the byte alignment of the internal binary buffer of a structure object. This value is identical with the value set with pragma pack() before typedef struct is declared.

Methods

:array()

Fills an array with all structure member values.

Syntax

```
:array() --> aMemberValues
```

Description

Method :array() returns an array filled with the values of all structure members.

:buffer()

Assigns a binary character string to the internal buffer.

Syntax

```
:buffer( <cBuffer>, [<lRecurse>] ) --> self
```

Arguments

<cBuffer>

This is a character string with replaces the internal binary buffer of a structure object.

<lRecurse>

This parameter defaults to .F. (false). It should be set to .T. (true) when the structure object maintains a nested structure. That is, it contains other structure objects in its instance variables.

Description

Method :buffer() is used to change the internal binary buffer of a structure object. *<cBuffer>* must be a binary character string with :sizeOf() bytes. If the string contains less characters, Chr(0) is added up to the correct length. The method extracts the values for each structure member from *<cBuffer>* and assigns them to the corresponding instance variables of the structure object.

:deValue()

Re-loads binary structure data from memory.

Syntax

```
:deValue( [<lRecurse>] ) --> self
```

Arguments

`<lRecurse>`

This parameter defaults to .F. (false). It should be set to .T. (true) when the structure object maintains a nested structure. That is, it contains other structure objects in its instance variables.

Description

Method :deValue() reads structure data from memory and assigns values to all instance variables representing structure members. This is required when the structure data is passed to an API function which changes values of structure members. The changed values are then copied into the instance variables of the structure object with method :deValue().

:getPointer()

Retrieves the pointer to the internal binary buffer.

Syntax

```
:getPointer() --> pStructure
```

Description

Method :getPointer() returns a pointer to the internal binary buffer of a structure object. This pointer can then be passed via DllCall() to an external API function requiring a structure as parameter. Note that method :deValue() must be called when the API function changes members of the structure. Otherwise the changes do not become visible in the structure object.

:pointer()

Changes the pointer to the internal binary buffer.

Syntax

```
:pointer( <pStructure>, [<lRecurse>] ) --> self
```

Arguments

`<pStructure>`

This is a pointer to structure data in memory.

`<lRecurse>`

This parameter defaults to .F. (false). It should be set to .T. (true) when the structure object maintains a nested structure. That is, it contains other structure objects in its instance variables.

Description

Method :pointer() is used to change the internal binary buffer of a structure object via a pointer. *<pStructure>* must be the memory address of :sizeOf() bytes. The method reads this number of bytes from memory and extracts from it the values for structure members. These values are assigned to the corresponding instance variables of the structure object.

:reset()

Voids the values of all structure members.

Syntax

```
:reset() --> self
```

Description

Method :reset() clears the internal binary buffer of a struture object and assignes NIL to all instance variables holding the values of structure members.

:sizeOf()

Determines the size of a structure in bytes.

Syntax

```
:sizeOf() --> nStructureSize
```

Description

Method :sizeOf() returns a numeric value representing the number of bytes a structure occupies in memory. This value is quite often required in structures used by the Windows API. If one structure member exist that indicates the structure size, it must be assigned the return value of method :sizeOf().

:value()

Retrieves the internal binary buffer of a structure object

Syntax

```
:value() --> cBinaryStructure
```

Description

Method :value() retrieves the binary representation of a structure and all its members and returns it as a character string.

Info

See also:	DllCall(), HB_Pointer2String(), HB_String2Pointer(), pragma pack(), (struct), typedef struct
Category:	C Structure support, xHarbour extensions
Header:	cstruct.ch, winapi.ch, wintypes.ch
Source:	rtl\cstruct.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example outlines the steps necessary for calling a
// Windows API function which requires a structure as parameter.

   #include "CStruct.ch"         // required for "typedef struct"
   #include "Wintypes.ch"        // required Windows C data types

   pragma pack(4)                // all Windows API structures
                                 // are 4 byte aligned

                                 // structure declaration taken via
                                 // copy&paste from Windows SDK
   typedef struct _OSVERSIONINFOEX { ;
      DWORD dwOSVersionInfoSize; //  ^ this ";" must be added
      DWORD dwMajorVersion;
      DWORD dwMinorVersion;
      DWORD dwBuildNumber;
      DWORD dwPlatformId;
```

```
   TCHAR szCSDVersion[128];
   WORD wServicePackMajor;
   WORD wServicePackMinor;
   WORD wSuiteMask;
   BYTE wProductType;
   BYTE wReserved;
} OSVERSIONINFOEX, *POSVERSIONINFOEX, *LPOSVERSIONINFOEX;

#define DC_CALL_STD            0x0020

PROCEDURE Main
   LOCAL oVerInfo

   // Create OSVERSIONINFOEX structure object
   oVerInfo := (struct OSVERSIONINFOEX)

   // assign structure size to structure member
   oVerInfo:dwOSVersionInfoSize := oVerInfo:sizeOf()

   // pass structure object by reference to DllCall()
   // (it is an OUT parameter)
   DllCall( "Kernel32.dll", DC_CALL_STD , ;
            "GetVersionEx", @oVerInfo )

   // display result of API
   ? oVerInfo:dwMajorVersion            // result:    5
   ? oVerInfo:dwMinorVersion            // result:    1
   ? oVerInfo:dwBuildNumber             // result: 2006

   // this member contains a byte array
   // retrieve it as character string
   ? oVerInfo:szCSDVersion:asString()   // result: Service Pack 2
RETURN
```

Error()

Creates a new Error object.

Syntax

```
Error( [<cSubSystem>]   , ;
       [<nGenCode>]     , ;
       [<nSubCode>]     , ;
       [<cOperation>]   , ;
       [<cDescription>], ;
       [<aArgs>]        , ;
       [<cModuleName>] , ;
       [<cProcName>]    , ;
       [<nProcLine> ]     ) --> oError
```

Arguments

<cSubSystem>

This is a character string holding the name of the subsystem where the error occurred. It is assigned to oError:subSystem.

<nGenCode>

This is a numeric value indicating a generic error code. It is assigned to oError:genCode. The file Error.ch contains #define constants used for this parameter. They begin with the prefix EG_.

<nSubCode>

This is a numeric value indicating an error code specific for the subsystem where the error occurred. It is assigned to oError:subCode.

<cOperation>

This is a character string holding the name of the failed operation. It is assigned to oError:operation.

<cDescription>

This is a character string holding a description of the failed operation. It is assigned to oError:description.

<aArgs>

This is an array whose elements hold the arguments of the failed operation. It is assigned to oError:args.

<cModuleName>

This is a character string holding the file name of the module where the error occurred. It is assigned to oError:moduleName.The parameter *<cModuleName>* defaults to the return value of function ProcFile().

<cProcName>

This is a character string holding the name of the function, method or procedure where the error occurred. It is assigned to oError:procName. The parameter *<cProcName>* defaults to the return value of function ProcName().

<nProcLine>

This is a numeric value indicating the line number of the source code file where the error occurred. It is assigned to oError:procLine. The parameter *<nProcLine>* defaults to the return value of function ProcLine().

Return

The function returns a new Error object, optionally filled with information about a runtime error.

Description

Objects of the Error class are used for error handling and error recovery by the xHarbour runtime system. An Error object is created when a runtime error occurs. Information about a runtime error is assigned to instance variables of the object before an exception is triggered by the runtime system. The Error object is then passed to the error handling routine programmed in the xHarbour application.

There are two statements available for user-defined error handling routines: BEGIN SEQUENCE and TRY...CATCH. See these statements for examples of local error handling.

Note: the default error handling routine is function DefError() implemented in the file source\rtl\ErrorSys.prg

Instance variables

:args

Array holding the arguments of a failed operation.

Data type: A
Default: NIL

Description

This instance variable receives a one-dimensional array whose elements contain the arguments of a failed operation. When no argument error is detected, the instance variable contains NIL.

:canDefault

Indicates if a default recovery is available.

Data type: L
Default: .F.

Description

If :canDefault contains .T. (true), a default error recovery routine is available. This routine is requested at runtime by returning .F. (false) from the current error code block set with ErrorBlock(). The minimum error recovery is to ignore the error. When the error code block returns .T. (true), the error handling routine is called again.

Note: the value of :canDefault is never .T. (true) when :canSubstitute is .T. (true), i.e. if :canSubstitute contains .T. (true), :canDefault contains .F. (false).

:canRetry

Indicates if a retry operation is possible.

Data type: L
Default: .F.

Description

If :canDefault contains .T. (true), a retry operation is possible. The failed operation is retried, when the current error code block set with ErrorBlock() returns .T. (true). Each time the failed operation is retried and fails again, the value of :tries is incremented by one.

Note: the value of :canRetry is never .T. (true) when :canSubstitute is .T. (true), i.e. if :canSubstitute contains .T. (true), :canRetry contains .F. (false).

:canSubstitute

Indicates if an erroneous result can be substituted.

Data type: L
Default: .F.

Description

If :canSubstitute contains .T. (true), the result of an erroneous operation can be substituted. The result of the failed operation is replaced with the return value of the current error code block set with ErrorBlock()

Note: the value of :canSubstitute is never .T. (true) if either :canDefault or :canRetry is .T. (true).

:cargo

Instance variable for user-defined purposes.

Data type: ANY
Default: NIL

Description

This instance variable is not used by an Error object. It is available for user-defined purposes when an arbitrary value should be attached to an Error object.

:description

Character string holding a description of the error.

Data type: C
Default: ""

Description

The instance variable contains a character string describing the error condition. If the subsystem provides no error description, :description contains an empty string ("").

:fileName

Name of the disk file associated with the operation

Data type: C
Default: ""

Description

The instance variable contains a character string holding the name of a disk file participating in the failed operation. If there is no file operation, :fileName contains an empty string ("").

:genCode

Generic numeric error code.

Data type: N
Default: NIL

Description

The instance variable contains a numeric value representing a generic error code. If :genCode contains zero, an unknown error has occurred. #define constants are listed in Error.ch for other generic error conditions.

Generic error codes

Constant	Value	Description
EG_ARG	1	Argument error
EG_BOUND	2	Bound error
EG_STROVERFLOW	3	Bound error,
EG_NUMOVERFLOW	4	Numeric overflow
EG_ZERODIV	5	Zero divisor
EG_NUMERR	6	Zero divisor
EG_SYNTAX	7	Syntax error
EG_COMPLEXITY	8	Operation too complex
EG_MEM	11	Memory low
EG_NOFUNC	12	Undefined function
EG_NOMETHOD	13	No exported method
EG_NOVAR	14	Variable does not exist
EG_NOALIAS	15	Alias does not exist
EG_NOVARMETHOD	16	No exported variable
EG_BADALIAS	17	Illegal characters in alias
EG_DUPALIAS	18	Alias already in use
EG_NOOBJECT	19	Not an object
EG_CREATE	20	Create error
EG_OPEN	21	Open error
EG_CLOSE	22	Close error
EG_READ	23	Read error
EG_WRITE	24	Write error
EG_PRINT	25	Print error
EG_UNSUPPORTED	30	Operation not supported
EG_LIMIT	31	Limit exceeded
EG_CORRUPTION	32	Corruption detected
EG_DATATYPE	33	Data type error
EG_DATAWIDTH	34	Data width error
EG_NOTABLE	35	Workarea not in use
EG_NOORDER	36	Workarea not indexed
EG_SHARED	37	Exclusive required
EG_UNLOCKED	38	Lock required
EG_READONLY	39	Write not allowed
EG_APPENDLOCK	40	Append lock failed
EG_LOCK	41	Lock Failure
EG_ARRACCESS	46	array access
EG_ARRASSIGN	47	array assign
EG_ARRDIMENSION	48	array dimension
EG_NOTARRAY	49	not an array
EG_CONDITION	50	conditional
EG_BADSELF	51	Invalid self

EG_ARRREF	52	reserved
EG_OLEEXECPTION	1001	(OLE exception occurred)

Note: if an Error object is created in a user-defined error handling routine, the above generic error codes should be used, if possible. A textual description of the error can then be retrieved using HB_LangErrMsg() which is then assigned to oError:description.

:moduleName

Character string holding the name of the PRG source file where the error occurred.

Data type: C
Default: ""

Description

The instance variable contains a character string holding the name of the PRG source code file where the failed operation is implemented. If this information is not available, :moduleName contains an empty string ("").

:operation

Character string holding a description of the failed operation.

Data type: C
Default: ""

Description

The instance variable contains a character string holding the name of the failed operation. If this information is not available, :operation contains an empty string ("").

:osCode

Numeric operating system error code.

Data type: N
Default: 0

Description

When an operation fails due to an Operating system error, its numeric error code is assigned to :osCode. It can also be retrieved using function DosError() or FError(). If the failed operation is not an operating system error, :osCode contains zero.

:procName

Character string holding the name of the function where the error occurred.

Data type: C
Default: ""

Description

The instance variable contains a character string holding the name of the function method or procedure where the failed operation occurred. If this information is not available, :procName contains an empty string ("").

:procLine

Numeric value indicating the line number of the source code file where the error occurred.

Data type: N
Default: 0

Description

The instance variable contains the line number of the PRG source code file where the failed operation occurred. If this information is not available, :procLine contains zero.

:severity

Numeric value indicating error severity.

Data type: N
Default: 0

Description

The instance variable contains a numeric value representing the severity of the error. #define constants are available in Error.ch to classify the severity of a runtime error:

Severity levels of runtime errors

Constant	Value	Description
ES_WHOCARES	0	Just informational, not really an error
ES_WARNING	1	This error is not a real error yet, but may lead to one later
ES_ERROR	2	This error must be corrected immediately in an error handling routine
ES_CATASTROPHIC	3	This error requires immediate termination of the application

:subCode

Numeric subsystem-specific error code.

Data type: N
Default: 0

Description

The instance variable contains a numeric value representing a subsystem-specific error code. If :subCode contains zero, the subsystem provides no further error information.

:subSystem

Character string holding the name of the subsystem where the error occurred.

Data type: C
Default: ""

Description

The instance variable contains a character string describing the subsystem that has raised the runtime error. If the subsystem is unknown, :subSystem contains an empty string ("").

:tries

Number of times the failed operation was attempted to be executed.

Data type: N
Default: 0

Description

The instance variable contains a numeric value which is initially zero. When :canRetry is .T. (true) and a failed operation is retried, the value of :tries is incremented by one. Thus, :tries is a counter for how often a failed operation is retried.

Info

See also: BEGIN SEQUENCE, DosError(), ErrorBlock(), FError(), HB_LangErrMsg(), Neterr(), TRY...CATCH, Throw()
Category: Object functions
Header: Error.ch
Source: rtl\terror.prg, rtl\error.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example is a code snippet from the file WIN32OLE.PRG.
// It demonstrates how a runtime error can be handled using the
// TRY...CATCH statements and how to fill an Error object with
// information about a runtime error.

   METHOD OleValuePlus( xArg ) CLASS TOleAuto

      LOCAL xRet, oErr

      TRY
         xRet := ::OleValue + xArg
      CATCH
         oErr := ErrorNew()
         oErr:Args          := { Self, xArg }
         oErr:CanDefault    := .F.
         oErr:CanRetry      := .F.
         oErr:CanSubstitute := .T.
         oErr:Description   := "argument error"
         oErr:GenCode       := EG_ARG
         oErr:Operation     := '+'
         oErr:Severity      := ES_ERROR
         oErr:SubCode       := 1081
         oErr:SubSystem     := "BASE"

         RETURN Throw( oErr )
      END

   RETURN xRet
```

Get()

Creates a new Get object.

Syntax

```
Get():new( [<nRow>]     , ;
           [<nCol>]     , ;
           <bVarBlock>  , ;
           [<cVarName>] , ;
           [<cPicture>] , ;
           [<cColorSpec>] ) --> oGet
```

Arguments

<nRow>

This is the numeric row coordinate on the screen where the Get object is displayed. It defaults to the return value of function Row(). *<nRow>* is assigned to oGet:row.

<nCol>

This is the numeric column coordinate on the screen where the Get object is displayed. It defaults to the return value of function Col(). *<nCol>* is assigned to oGet:col.

<bVarBlock>

This is the data code block connecting a Get object with a variable holding the edited value (see description below). *<bVarBlock>* is assigned to oGet:block.

<cVarName>

This is a character string holding the symbolic name of a memory variable of PRIVATE or PUBLIC scope. If the Get object should edit such a variable, *<bVarBlock>* is optional. *<cVarName>* defaults to an empty string ("") and is assigned to oGet:name.

<cPicture>

This is a character string holding a PICTURE format. *<cPicture>* defaults to an empty string ("") and is assigned to oGet:picture.

<cColorSpec>

This is a character string holding a color string of up to four color values. (refer to function SetColor() for color values). *<cColorSpec>* is assigned to oGet:colorSpec.

Return

Function Get() returns a new Get object and method :new() initializes the object.

Description

Get objects provide the means for data entry in text mode applications. They are generally created with the @...GET command and collected in a PUBLIC array named GetList. All Get objects stored in the GetList array are processed with the READ command, or the ReadModal() function, respectively.

A Get object displays data to be edited and provides methods for keyboard input and cursor movement when it receives input focus. The data to be edited can be stored a memory or field variable, it is not stored in the Get object itself. Instead, a Get object must receive a data code block with the *<bVarBlock>* parameter, which can be created using functions MemVarBlock(), FieldBlock() or FieldWBlock().

The data code block can also be coded in a general form like this:

```
LOCAL variable

bVarBlock := {|xValue| IIf( xValue==NIL, variable, variable := xValue ) }
```

This code block has one parameter and refers to the variable whose value is edited. A Get object retrieves the variable's value by evaluating the data code block with no parameter, and it assigns the edited value to the variable by passing the edited value to the data code block when it looses input focus.

Instance variables

:badDate

Indicates if the editing buffer contains an invalid date value.

Data type: L
Default: .F.

Description

The instance variable :badDate contains .T. (true) when a Date value is being edited and the current value in the edit buffer of a Get object represents an invalid date. In all other cases, :badDate contains .F. (false).

:block

Data code block of the Get object.

Data type: B
Default: NIL

Description

The instance variable :block contains the data code block of a Get object. It is the most important instance variable and must be provided upon object creation. The data code block is the connection between the Get object and the edited variable.

Note: a Get object evaluates the data code block during different stages of an editing cycle and it is not recommended to evaluate :block directly. Instead, use methods :varGet() and :varPut() when the data code block must be evaluated explicitely. This way, a Get object receives knowledge when the edited variable is changed outside the Get object.

:buffer

Character string held in the editing buffer.

Data type: C
Default: NIL

Description

The instance variable :buffer contains the editing buffer while a Get object has input focus. This buffer is the edited value in form of a character string.

:capCol

Numeric column position of a caption.

Data type: N
Default: 0

Description

The instance variable :capCol contains the numeric column position of the caption of a Get object. It is only meaningful when instance variable :caption contains a character string to display as a caption.

:capRow

Numeric row position of a caption.

Data type: N
Default: 0

Description

The instance variable :capRow contains the numeric row position of the caption of a Get object. It is only meaningful when instance variable :caption contains a character string to display as a caption.

:caption

Character string defining the caption.

Data type: C
Default: ""

Description

The instance variable :caption can be assigned a character string to be displayed at position :capRow and :capCol whithin the Get object's :display() method. If the :caption string contains an ampersand (&), the ampersand is removed from the :caption string and the next character to the right of the ampersand is displayed using the fourth color value of :colorSpec.

:cargo

Instance variable for user-defined purposes.

Data type: ANY
Default: NIL

Description

This instance variable is not used by a Get object. It is available for user-defined purposes when an arbitrary value should be attached to a Get object.

:changed

Indicates if the editing buffer was changed.

Data type: L
Default: .F.

Description

The instance variable :changed contains .T. (true) when the edit buffer of the Get object is changed while the object has input focus. In all other cases, :changed contains .F. (false).

:clear

Indicates if the editing buffer should be cleared

Data type: L
Default: .F.

Description

The instance variable :clear contains .T. (true) when the Get object has input focus and the edit buffer of the object should be cleared with the next key stroke. This is the case when the first key press is numeric and the edited value is numeric, or when the picture function "@K" is used in the :picture string. In all other cases, :clear contains .F. (false).

:col

Numeric column position of the Get object.

Data type: N
Default: Col()

Description

The instance variable :col contains the numeric column position where a Get object displays the value of the edited variable on the screen.

:colorSpec

Color string for display and editing.

Data type: C
Default: SetColor()

Description

The instance variable :colorSpec contains a color string holding up to four color values. They are used for the following purposes:

Color usage of Get objects

Color value	Description
#1	Displays edited value when Get does not have input focus
#2	Displays edited value when Get has input focus
#3	Displays caption string of Get object
#4	Displays character marked with an ampersan (&) in the caption string

Note: if :colorSpec contains less than four color values, the last color value is used for the missing colors.

:control

Optional object for extended editing.

Data type: O
Default: NIL

Description

The instance variable :control can be assigned an object that must have the method :hitTest(). The method is used in the same way as Get:hitTest(). The :control object, such as HbCheckBox() or TBrowse(), is associated with the Get object.

The following objects are stored in this instance variable when a Get object is created with an extended @...GET command:

Associated control objects for Gets

Command	Control
@...GET CHECKBOX	HbCheckBox() object
@...GET LISTBOX	HbListBox() object
@...GET PUSHBUTTON	HbPushButton() object
@...GET RADIOGROUP	HbRadioGroup() object
@...GET TBROWSE	TBrowse() object

:decPos

Numeric position of the decimal point during editing.

Data type: N
Default: NIL

Description

The instance variable :decPos contains a numeric value greater than zero when a Get object has input focus and the edited value is numeric and has a decimal fraction. In this case, :decPos is the position of the decimal point in the edit buffer. In all other cases, :decPos contains zero.

:exitState

Numeric code indicating how editing was terminated.

Data type: N
Default: GE_NOEXIT

Description

The instance variable :exitState contains a numeric value indicating how the editing process was terminated. #define constants are available in the file GETEXIT.CH that identify exit states.

Exit states of Get objects

Constant	Value	Description
GE_NOEXIT	0	No exit attempted
GE_UP	1	Go to previous Get
GE_DOWN	2	Go to next Get
GE_TOP	3	Go to first Get
GE_BOTTOM	4	Go to last Get
GE_ENTER	5	Terminate editing of current Get and proceed with next
GE_WRITE	6	Terminate editing of all Gets in GetList array (terminate READ command)

GE_ESCAPE	7	Terminate editing of all Gets without saving data
GE_WHEN	8	Pre-validation failed, don't give input focus
GE_SHORTCUT	9	Accelerator key is pressed (see :caption)
GE_MOUSEHIT	10	Another Get is clicked with the mouse

:hasFocus

Indicates if the Get object has input focus.

Data type: L
Default: .F.

Description

The instance variable :hasFocus contains .T. (true) when Get object has input focus, otherwise .F. (false).

:minus

Indicates if a minus sign is entered for numbers.

Data type: L
Default: .F.

Description

The instance variable :minus contains .T. (true) when Get object has input focus, the edited value is a number and the minus sign is entered inti the edit buffer. In all other cases, :minus contains .F. (false).

:message

Character string to be displayed as message when Get receives input focus.

Description

The instance variable :message can be assigned a character string which is displayed in the message area defined with the MSG AT option of the READ command. The message is displayed when the Get object receives input focus.

:name

Character string holding the symbolic name of the edited variable.

Data type: C
Default: ""

Description

The instance variable :name contains the character string passed with parameter <*cVarName*> to the :new method. It is the symbolic name of the edited variable. Note that this is only relevant when the edited variable is of PRIVATE or PUBLIC scope. Symbolic names of such variables are available at runtime and can be used to generate a data code block for the Get object.

:original

Character string holding the original value before editing started.

Data type: C
Default: ""

Description

Before a Get object starts editing a value, it creates a copy of the edited value and stores it in the instance variable :original in form of a character string. When the user has changed the edit buffer, the original value can be recalled with method :undo(). :original holds the original value only while the Get object has input focus.

:picture

Character string holding a PICTURE format.

Data type: C
Default: ""

Description

The instance variable :picture holds a PICTURE formatting string used for displaying and editing the variable. Refer to command @...GET for a comprehensive description of PICTURE formatting strings.

:pos

Current cursor position within the editing buffer.

Data type: N
Default: 0

Description

The instance variable :pos contains a numeric value greater than zero while a Get object has input focus. It is the current cursor position within the edit buffer. In all other cases, :pos contains zero.

:postBlock

Code block to validate a new value before Get looses input focus.

Data type: B
Default: NIL

Description

Optionally, a code block can be assigned to :postValidate that is used to validate edited data. The code block receives as parameter the Get object and must return a logical value. When the code block returns .T. (true) the Get object looses input focus. When it returns .F. (false), the current data available in the edit buffer is assumed invalid and the Get object retains input focus.

:preBlock

Code block to validate a value before Get receives input focus.

Data type: B
Default: NIL

Description

Optionally, a code block can be assigned to :preValidate that is used to test if a Get object may receive input focus. The code block receives as parameter the Get object and must return a logical value. When the code block returns .T. (true), the Get object receives input focus. When it returns .F. (false), the next Get object is searched that may receive input focus.

:reader

Code block calling an alternative Get reader procedure.

Data type: B
Default: NIL

Description

The instance variable :reader is reserved for a user-defined Get reader procedure. :reader can be assigned a code block which receives the Get object and must call a procedure that processes user input for the Get object. Refer to METHOD Reader in source\rtl\Getlist.prg for an example of a Get reader implementation.

:rejected

Indicates if the last key stroke was rejected during editing.

Data type: L
Default: .F.

Description

The instance variable :rejected contains .T. (true) when the last key stroke is rejected during editing. For example, when a number is edited and the user typed an alphabetic character. When the Get object has no input focus or when a valid key is pressed, :rejected contains .F. (false).

:row

Numeric row position of the Get object.

Data type: N
Default: Row()

Description

The instance variable :row contains the numeric row position where a Get object displays the value of the edited variable on the screen.

:subscript

Array element subscript when array is edited.

Data type: A
Default: NIL

Description

When the data code block references a variable holding an array, the instance variable :subscript must be assigned an array whose element(s) contain the numeric subscript(s) pointing to the array element that is to be edited by the Get object. For example:

```
LOCAL aArray := { "One", "Two" }
LOCAL bData  := {|| aArray }
LOCAL oGet   := Get():new( ,, bData )

oGet:subscript := {2}

? oGet:varGet()     // result: Two
```

:type

Data type of edited value.

Data type: C
Default: ""

Description

The instance variable :type contains a single letter indicating the data type of the edited value while the Get object has input focus. This letter is equivalent with the return value of function Valtype().

:typeOut

Indicates attempt to move the cursor out of editing buffer.

Data type: L
Default: .F.

Description

The instance variable :typeOut contains .T. (true) when the user attempts to move the cursor outside the edit buffer, or when the edit buffer is full and a new key is pressed in insert mode. In all other cases, :typeOut contains .F. (false).

State changing methods

:assign()

Assigns the contents of the edit buffer to the edited variable

Syntax

```
:assign() --> self
```

Description

The :assign() method evaluates the data code block and passes the current value in the edit buffer on to it, thus assigning the edited value to the edited variable. The method is only meaningful when the Get object has input focus.

:colorDisp()

Changes the color and redisplays the Get.

Syntax

```
:colorDisp( <cColorSpec> ) --> self
```

Arguments

`<cColorSpec>`

This is a character string holding a color string of up to four color values. *<cColorSpec>* is assigned to :colorSpec.

Description

The method :colorDisp() assigns a new color string to :colorSpec and redisplays the Get object.

:display()

Displays the Get on the screen.

Syntax

```
:display() --> self
```

Description

When the Get object has input focus, method :display() displays the edited value using the second color value of :colorSpec and positions the cursor on the first editable character in the edit buffer. If the Get object does not have input focus, the edited value is displayed using the first color value of :colorSpec and the cursor remains unchanged.

:hitTest()

Tests if a Get is hit by the mouse cursor.

Syntax

```
:hitTest( <nMouseRow>, <nMouseCol> ) --> nHitCode
```

Arguments

`<nMouseRow>`

This is the numeric row position of the mouse cursor. It can be queried using function MRow().

`<nMouseCol>`

This is the numeric column position of the mouse cursor. It can be queried using function MCol().

Description

Method :hitTest() accepts the numeric row and column position of the mouse cursor and returns a numeric value indicating whether or not the mouse cursor is located within the Get object. #define constants are available in the BUTTON.CH file identifying the return value of :hitTest().

Return values of oGet:hitTest()

Constant	Value	Description
HTNOWHERE	0	Mouse cursor is outside Get object
HTCLIENT	-2049	Mouse cursor is inside Get object

:killFocus()

Removes input focus from the Get object.

Syntax

```
:killFocus() --> self
```

Description

Method :killFocus() removes the input focus from a Get object and releases all internal variables required during editing, such as :buffer or :original.

:reset()

Resets internal state information of a Get object.

Syntax

```
:reset() --> self
```

Description

Method :reset() rebuilds the character string held in the edit buffer of a Get object from the current value of the edited variable, and moves the cursor to the first editable character. The method is only meaningful while the Get object has input focus.

:setFocus()

Gives input focus to the Get object.

Syntax

```
:setFocus() --> self
```

Description

Method :setFocus() gives input focus to a Get object and assigns values to instance variables required for editing. This includes :buffer, :original, :pos and :decPos.

:undo()

Assigns the original value back to the edited variable.

Syntax

```
:undo() --> self
```

Description

Method :undo() discards all changes applied to the edit buffer during editing and assigns the contents of :original back to the edited variable.

:unTransform()

Converts the editing buffer to its original data type.

Syntax

```
:unTransform() --> xValue
```

Description

Method :unTransform() converts the character string held in :buffer to the original data type of the edited value and returns the converted value. This method is only meaningful while the Get object has input focus.

:updateBuffer()

Updates the editing buffer and redisplays the Get.

Syntax

```
:updateBuffer() --> self
```

Description

Method :updateBuffer() rebuilds the editing buffer from the current value of the edited variable and displays the buffer. This method is only meaningful while the Get object has input focus.

:varGet()

Returns the current value of the edited variable.

Syntax

```
:varGet() --> xValue
```

Description

Method :varGet() evaluates the data code block stored in :block without passing a value to it, and returns the value of the edited variable. If the variable holds an array, :varGet() returns the contents of the array element specified with :subscript.

:varPut()

Assigns a new value to the edited variable.

Syntax

```
:varPut( <xValue>) --> xValue
```

Description

Method :varPut() evaluates the data code block stored in :block passing *<xValue>* to it, and returns this value. If the edited variable holds an array, :varPut() assigns *<xValue>* to the array element specified with :subscript.

Cursor movement methods

:end()

Moves the cursor to the last position in the edit buffer.

Syntax

```
:end() --> self
```

Description

When a Get object has input focus, method :end() moves the cursor to the rightmost editable character in the edit buffer.

:home()

Moves the cursor to the first position in the edit buffer.

Syntax

```
:home() --> self
```

Description

When a Get object has input focus, method :home() moves the cursor to the leftmost editable character in the edit buffer.

:left()

Moves the cursor one character to the left.

Syntax

```
:left() --> self
```

Description

When a Get object has input focus, method :left() moves the cursor one character to the left in the edit buffer, unless the cursor is positioned on the first editable character.

:right()

Moves the cursor one character to the right.

Syntax

```
:right() --> self
```

Description

When a Get object has input focus, method :right() moves the cursor one character to the right in the edit buffer, unless the cursor is positioned on the last editable character.

:toDecPos()

Moves the cursor to the immediate right of Get:decPos.

Syntax

```
:toDecPos() --> self
```

Description

When a Get object has input focus and the edited value is a number with a decimal fraction, method :toDecPos() moves the cursor to the immediate right of the decimal point.

:wordLeft()

Moves the cursor to the beginning of the next left word.

Syntax

```
:wordLeft() --> self
```

Description

When a Get object has input focus, method :wordLeft() moves the cursor to the first character of the next word left of the current cursor position.

:wordRight()

Moves the cursor to the beginning of the next right word.

Syntax

```
:wordRight() --> self
```

Description

When a Get object has input focus, method :wordRight() moves the cursor to the first character of the next word right of the current cursor position.

Editing methods

:backspace()

Moves the cursor to the left and deletes a character.

Syntax

```
:backspace() --> self
```

Description

When a Get object has input focus, method :backSpace() moves the cursor one character to the left and deletes this character.

:delete()

Deletes the character at the current cursor position.

Syntax

```
:delete() --> self
```

Description

When a Get object has input focus, method :delete() deletes the character at the current cursor position.

:delEnd()

Deletes all characters from the current cursor position to the end of the edit buffer.

Syntax

```
:delEnd() --> self
```

Description

When a Get object has input focus, method :delEnd() deletes the characters in the edit buffer beginning at the current cursor position up to the last character in the buffer.

:delLeft()

Deletes the character to the left of the cursor.

Syntax

```
:delLeft() --> self
```

Description

When a Get object has input focus, method :delLeft() deletes the character to the left of the cursor position without moving the cursor.

:delRight()

Deletes the character to the right of the cursor.

Syntax

```
:delRight() --> self
```

Description

When a Get object has input focus, method :delRight() deletes the character to the right of the cursor position without moving the cursor.

:delWordLeft()

Deletes the word to the left of the cursor.

Syntax

```
:delWordLeft() --> self
```

Description

When a Get object has input focus, method :delWordLeft() moves the cursor to the first character of the next word left of the current cursor position and deletes the entire word.

:delWordRight()

Deletes the word to the right of the cursor

Syntax

```
:delWordRight() --> self
```

Description

When a Get object has input focus, method :delWordRight() moves the cursor to the first character of the next word right of the current cursor position and deletes the entire word.

:insert()

Inserts a single character into the edit buffer.

Syntax

```
:insert( <cCharacter> ) --> self
```

Description

When a Get object has input focus, method :insert() inserts the character *<cCharacter>* into the edit buffer at the current cursor posiion. If *<cCharacter>* is not valid, or if the edit buffer is full, :rejected is set to .T. (true) and the edit buffer remains unchanged.

:overStrike()

Overwrites a single character in the editing buffer.

Syntax

```
:overStrike( <cCharacter> ) --> self
```

Description

When a Get object has input focus, method :overstrike() replaces the character at the current cursor posiion in the edit buffer with *<cCharacter>*. If *<cCharacter>* is not valid, :rejected is set to .T. (true) and the edit buffer remains unchanged.

Info

See also:	@...GET, READ, ReadModal(), ReadVar(), Valtype(), SetColor()
Category:	Get system, Object functions
Header:	Button.ch, Getexit.ch
Source:	rtl\getsys.prg, rtl\tget.prg, rtl\tgetlist.prg, rtl\mssgline.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example compares the creation of a single Get
// using command oriented and object oriented syntax

   PROCEDURE Main
```

```
   LOCAL cString:= "Testing Gets      "
   LOCAL bBlock

   @ 2, 2 SAY "String 1"        ;
          GET cString           ;
        COLOR "N/BG,W+/B"       ;
        VALID !Empty( cString ) ;
      PICTURE "@K"

   READ

   bBlock := {|x| IIf( x==NIL, cString, cString := x ) }

   @ 4, 2 SAY "String 2"

   oGet           := Get():new()
   oGet:row       := Row()
   oGet:col       := Col() + 1
   oGet:block     := bBlock
   oGet:name      := "cString"
   oGet:picture   := "@K"
   oGet:colorSpec := "N/BG,W+/B"
   oGet:postBlock := {|o| ! Empty(o:varGet()) }
   oGet:display()

   ReadModal( {oGet} )

RETURN
```

HbCheckBox()

Creates a new HbCheckBox object.

Syntax

```
HbCheckBox():new( <nRow>    , ;
                  <nCol>    , ;
                 [<cCaption>] ) --> oHbCheckBox
```

Arguments

`<nRow>`

This is the numeric row coordinate on the screen where the HbCheckBox object is displayed. *<nRow>* is assigned to oHbCheckBox:row.

`<nCol>`

This is the numeric column coordinate on the screen where the HbCheckBox object is displayed. *<nCol>* is assigned to oHbCheckBox:col.

`<cCaption>`

This is an optional character string holding the caption of the check box. *<cCaption>* is assigned to oHbCheckBox:caption.

Return

Function HbCheckBox() returns a new HbCheckBox object and method :new() initializes the object.

Description

Objects of the HbCheckBox class are used in text mode applications to display and edit a logical state (On or Off). They are designed to be integrated into the standard Get system and are usually created with the @...GET CHECKBOX command. This command creates a Get() object and an associated HbCheckBox object. Editing the logical edit buffer of the check box is then accomplished by the READ command.

Instance variables

:buffer

Logical edit buffer indicating the (un)checked state.

Data type: L
Default: .F.

Description

The instance variable :buffer represents the edit buffer of a HbCheckBox object. This buffer contains either .T. (true) for the Checked state, or .F. (false) for the Unchecked state.

:capCol

Numeric column position of the caption.

Data type: N
Default: NIL

Description

The instance variable :capCol contains the numeric column position of the caption of a HbCheckBox object. It is only meaningful when instance variable :caption contains a character string to display as a caption.

:capRow

Numeric row position of the caption.

Data type: N
Default: NIL

Description

The instance variable :capRow contains the numeric row position of the caption of a HbCheckBox object. It is only meaningful when instance variable :caption contains a character string to display as a caption.

:caption

Character string defining the caption.

Data type: C
Default: NIL

Description

The instance variable :caption can be assigned a character string to be displayed at position :capRow and :capCol whithin the :display() method. If the :caption string contains an ampersand (&), the ampersand is removed from the :caption string and the next character to the right of the ampersand is used as accelerator key while READ is active. The accelerator is displayed in a separate color (see :colorSpec).

:caption receives the third parameter passed to the :new() method.

:cargo

Instance variable for user-defined purposes.

Data type: ANY
Default: NIL

Description

This instance variable is not used by a HbCheckBox object. It is available for user-defined purposes when an arbitrary value should be attached to the object.

:col

Numeric column position of the HbCheckBox object.

Data type: N
Default: NIL

Description

The instance variable :col contains the numeric column position where a HbCheckBox object displays the check box on the screen. :col receives the second parameter passed to the :new() method.

:colorSpec

Color string for display and editing.

Data type: C
Default: W/N,W+/N,W/N,W+/N

Description

The instance variable :colorSpec contains a SetColor() string holding up to four color values. They are used for the following purposes:

Color values for check boxes

Color value	Description
1	Color when the check box does not have input focus
2	Color when the check box has input focus
3	Color for the check box's caption
4	Color for the check box's accelerator key

:fBlock

Code block evaluated when input focus changes.

Data type: B
Default: NIL

Description

The instance variable :fBlock can be assigned a code block. It is evaluated during the standard READ command when the HbCheckBox object receives input focus. The code block is evaluated without parameters.

:hasFocus

Indicates if the HbCheckBox object has input focus.

Data type: L
Default: .F.

Description

The instance variable :hasFocus contains .T. (true) when a HbCheckBox object has input focus, otherwise .F. (false).

:message

Character string to be displayed as message when the check box receives input focus.

Data type: C
Default: ""

Description

The instance variable :message can be assigned a character string which is displayed in the message area defined with the MSG AT option of the READ command. The message is displayed when the HbCheckBox object receives input focus.

:row

Numeric row position of the HbCheckBox object.

Data type: N
Default: NIL

Description

The instance variable :row contains the numeric row position where a HbCheckBox object displays the check box on the screen. :row receives the first parameter passed to the :new() method.

:sBlock

Code block evaluated when the (un)checked state changes.

Data type: N
Default: NIL

Description

The instance variable :sBlock can be assigned a code block. It is evaluated during the standard READ command when the (un)checked state of the HbCheckBox object changes. The code block is evaluated without parameters.

:style

Character string defining the check box delimiters.

Data type: C

Description

The display style of a check box can be specified with an optional character string of four characters. They are used as follows:

Characters for the display style

Position	Description
1	Left delimiter of check box
2	Character for the Checked state
3	Character for the Unchecked state
4	Right delimiter of check box

:typeOut

Logical value .F. (false).

Data type: L
Default: .F.

Description

The instance variable :typeOut contains always .F. (false). It is required as place holder in the standard Get system.

Methods

:display()

Displays the check box on the screen.

Syntax

```
:display() --> self
```

Description

Method :display() displays the check box and takes care of the (un)checked state and different colors depending on the input focus. It uses the instance variables :buffer, :row, :col, :capRow, :capCol, :caption, :hasFocus, :colorSpec and :style.

:hitTest()

Tests if a check box is hit by the mouse cursor.

Syntax

```
:hitTest( <nMouseRow>, <nMouseCol> ) --> nHitCode
```

Arguments

<nMouseRow>

This is the numeric row position of the mouse cursor. It can be queried using function MRow().

<nMouseCol>

This is the numeric column position of the mouse cursor. It can be queried using function MCol().

Description

Method :hitTest() accepts the numeric row and column position of the mouse cursor and returns a numeric value indicating whether or not the mouse cursor is located within the HbCheckBox object. #define constants are available in the BUTTON.CH file identifying the return value of :hitTest().

Return values of oHbCheckBox:hitTest()

Constant	Value	Description
HTNOWHERE	0	Mouse cursor is outside the check box and its caption
HTCAPTION	-1025	Mouse cursor is inside the check box's caption
HTCLIENT	-2049	Mouse cursor is inside the check box

:killFocus()

Removes input focus from the HbCheckBox object.

Syntax

```
:killFocus() --> self
```

Description

Method :killFocus() removes the input focus from a HbCheckBox object and displays it in its normal color.

:select()

Toggles or defines the (un)cheked state.

Syntax

```
:select( [<lChecked>] ) --> self
```

Arguments

`<lCHecked>`

This is an optional logical value defining the Checked (.T.) or Unchecked (.F.) state.

Description

When a parameter is passed to the :select() method, it defines the new state of a HbCheckBox object. Calling the method without parameter toggles its state from Checked to Unchecked and vice versa.

:setFocus()

Gives input focus to the HbCheckBox object

Syntax

```
:setFocus() --> self
```

Description

Method :setFocus() gives input focus to a HbCheckBox object and displays it in its highlighted color.

Info

See also:	@...GET, @...GET CHECKBOX, Get(), READ, ReadModal(), ReadVar(), Valtype(), SetColor()
Category:	Object functions, Get system
Header:	button.ch
Source:	rtl\checkbox.prg
LIB:	lib\xhb.lib
DLL:	dll\xhbdll.dll

HbListBox()

Creates a new HbListBox object

Syntax

```
HbListBox():new( <nTop>      , ;
                 <nLeft>     , ;
                 <nBottom>   , ;
                 <nRight>    , ;
                [<lDropDown>] ) --> oHbListBox
```

Arguments

`<nTop>, <nLeft>, <nBottom>, <nRight>`

These numeric parameters indicate the screen coordinates for the upper left and lower right corner of the list box output. The range for rows on the screen is 0 to MaxRow(), and for columns it is 0 to MaxCol(). The coordinate 0,0 is the upper left corner of the screen.

`<lDropDown>`

This parameter defaults to .F. (false), indicating that the list box is displayed as a multi-line list box. When set to .F. (true) a drop-down list box is displayed in a single line on the screen.

Return

Function HbListBox() returns a new HbListBox object and method :new() initializes the object.

Description

Objects of the HbListBox class are used in text mode applications to display a list of items and let the user select one of them. HbListBox objects are designed to be integrated into the standard Get system and are usually created with the @...GET LISTBOX command. This command creates a Get() object and an associated HbListBox object. Selecting a single item from the list box is then accomplished by the READ command.

When a HbListBox object is created with the :new() method, instead of the @...GET LISTBOX command, it must be filled with items using method :addItem().

Instance Variables

:bottom

Numeric bottom row coordinate of the list box display.

Data type: N
Default: 0

Description

The instance variable :bottom contains the numeric bottom row position of the rectangle where a HbListBox object displays its items on the screen. :bottom receives the third parameter passed to the :new() method.

:buffer

Numeric ordinal position of the selected item.

Data type: N
Default: 0

Description

The instance variable :buffer represents the edit buffer of a HbListBox object. This buffer contains a numeric value indicating the ordinal position of the currently selected item in the list box.

:capCol

Numeric column position of the caption.

Data type: N
Default: NIL

Description

The instance variable :capCol contains the numeric column position of the caption of a HbListBox object. It is only meaningful when instance variable :caption contains a character string to display as a caption.

:capRow

Numeric row position of the caption.

Data type: N
Default: NIL

Description

The instance variable :capRow contains the numeric row position of the caption of a HbListBox object. It is only meaningful when instance variable :caption contains a character string to display as a caption.

:caption

Character string defining the caption.

Data type: C
Default: ""

Description

The instance variable :caption can be assigned a character string to be displayed at position :capRow and :capCol whithin the :display() method. If the :caption string contains an ampersand (&), the ampersand is removed from the :caption string and the next character to the right of the ampersand is used as accelerator key while READ is active. The accelerator is displayed in a separate color (see :colorSpec).

:cargo

Instance variable for user-defined purposes.

Data type: ANY
Default: NIL

Description

This instance variable is not used by a HbListBox object. It is available for user-defined purposes when an arbitrary value should be attached to the object.

:coldBox

Box string for border when list box has no input focus.

Data type: C
Default: B_SINGLE

Description

The instance variable :coldBox contains a DispBox() compatible character string defining the border drawn around the list box when the HbListBox object has no input focus. #define constants are available in the Box.ch file that can be used for :coldBox.

Pre-defined box strings for borders

Constant	Description
B_SINGLE *)	Single-line box
B_DOUBLE	Double-line box
B_SINGLE_DOUBLE	Single-line top, double-line sides
B_DOUBLE_SINGLE	Double-line top, single-line sides

**) default*

:colorSpec

Color string for display and selecting.

Data type: N
Default: W/N,W+/N,W+/N,N/W,W/N,W/N,W+/N,W/N

Description

The instance variable :colorSpec contains a SetColor() string holding up to eight color values. They are used for the following purposes:

Color values for list boxes

Color value	Description
1	Color for unselected list box items when the list box does not have input focus
2	Color for the selected list box item when the list box does not have input focus
3	Color for unselected list box items when the list box has input focus
4	Color for the selected list box item when the list box has input focus
5	Color for the list box's border
6	Color for the list box's caption

7	Color for the list box's accelerator key
8	Color for the list box's drop down button

:dropDown

Logical value indicating a drop-down list box.

Data type: L
Default: .F.

Description

If the instance variable :dropDown contains .T. (true), the list box is initially displayed in one row and can be dropped down by pressing the Space key when the list box has input focus, or clicking the dowp down button. By default, :dropDown contains .F. (false) which displays the list box in multiple lines.

:fBlock

Code block evaluated when input focus changes.

Data type: B
Default: NIL

Description

The instance variable :fBlock can be assigned a code block. It is evaluated during the standard READ command when the HbListBox object receives input focus. The code block is evaluated without parameters.

:hasFocus

Indicates if the HbListBox object has input focus.

Data type: L
Default: .F.

Description

The instance variable :hasFocus contains .T. (true) when a HbListBox object has input focus, otherwise .F. (false).

:hotBox

Box string for border when list box has input focus.

Data type: C
Default: B_DOUBLE

Description

The instance variable :hotBox contains a DispBox() compatible character string defining the border drawn around the list box when the HbListBox object has input focus. #define constants are available in the Box.ch file that can be used for :hotBox.

Pre-defined box strings for borders

Constant	Description
B_SINGLE	Single-line box
B_DOUBLE *)	Double-line box
B_SINGLE_DOUBLE	Single-line top, double-line sides

B_DOUBLE_SINGLE	Double-line top, single-line sides

**) default*

:isOpen

Indicates if a drop-down list box is dropped down, or open.

Data type: L
Default: .F.

Description

The instance variable :isOpen contains .T. (true) when a drop-down list box is dropped down, or open, otherwise .F. (false). Method :open() opens a drop-down list box and :close() closes it.

:itemCount

Numeric value indicating the total number of items in the list box.

Data type: N
Default: 0

Description

The instance variable :itemCount contains a numeric value representing the number of items maintained by a HbListBox object. Items are added to the list box with the :addItem() method.

:left

Numeric left column coordinate of the list box display.

Data type: N
Default: NIL

Description

The instance variable :left contains the numeric left column position of the rectangle where a HbListBox object displays its items on the screen. :left receives the second parameter passed to the :new() method.

:message

Character string to be displayed as message when the list box receives input focus.

Data type: C
Default: ""

Description

The instance variable :message can be assigned a character string which is displayed in the message area defined with the MSG AT option of the READ command. The message is displayed when the HbListBox object receives input focus.

:right

Numeric right column coordinate of the list box display.

Data type: N
Default: NIL

Description

The instance variable :right contains the numeric right column position of the rectangle where a HbListBox object displays its items on the screen. :right receives the fourth parameter passed to the :new() method.

:sBlock

Code block evaluated when the selected item changes.

Data type: B
Default: NIL

Description

The instance variable :sBlock can be assigned a code block. It is evaluated during the standard READ command when the currently selected item of the HbListBox object is changed. The code block is evaluated without parameters.

:style

Character string defining drop-down button.

Data type: C
Default: Chr(31)

Description

The display style of a drop-down list box can be specified with a single character assigned to :style. It is displayed as the drop-down button. The instance variable :dropDown must be set to .T. (true) when a HbListBox object should display a drop-down list box.

:top

Numeric top row coordinate of the list box display.

Data type: N
Default: NIL

Description

The instance variable :top contains the numeric top row position of the rectangle where a HbListBox object displays its items on the screen. :top receives the first parameter passed to the :new() method.

:topItem

Numeric ordinal position of the first visible list box item.

Data type: N
Default: 0

Description

The instance variable :topItem contains a numeric value representing the ordinal position of the list box item visible in the first line of the list box display. Items are added to the list box with the :addItem() method.

:typeOut

Logical value indicating if a list box is empty.

Data type: L
Default: .F.

Description

The instance variable :typeOut contains .T. (true) when a HbListBox object is empty and has no items, otherwise .F. (false).

:vScroll

Optional vertical ScrollBar object.

Data type: HbScrollBar()
Default: NIL

Description

The instance variable :vScroll can be assigned a HbScrollBar() object which must be configured for displaying a vertical scroll bar. This can be accomplished using the SCROLLBAR option of the extended @...GET LISTBOX command.

Methods

:addItem()

Adds a new item to a list box.

Syntax

```
:addItem( <cText>, [<xValue>] ) --> self
```

Arguments

`<cText>`

This is a character string holding the textual information of a list box item.

`<xValue>`

Optionally, an arbitrary value can be passed which is associated with *<cText>*.

Description

Method :addItem() accepts a text string and an optional associated value which are added it to the internal list of list box items. The return value is the HbListBox object. The instance variable :itemCount reflects the new number of list box items maintained by a HbListBox object.

:close()

Closes an open drop-down list box.

Syntax

```
:close() --> self
```

Description

This method closes a drop-down list box previously opened with the :open() method. When the object is not configured as a drop-down list box, or when the list box is not dropped down, the :close() method has no effect.

:delItem()

Removes a list box item from a HbListBox object.

Syntax

```
:delItem( <nItemPos> ) --> self
```

Arguments

<nItemPos>

This is a numeric value specifying the ordinal position of the list box item to remove from the list box. Items are numbered from 1 to :itemCount.

Description

Method :delItem() is used to remove a single list box item from a HbListBox object. The method returns the HbListBox object. Use method :getItem() to preserve the list box item before it is deleted.

:display()

Displays the list box on the screen.

Syntax

```
:display()  --> self
```

Description

Method :display() displays the list box and takes care of the different colors depending on the input focus. It uses the instance variables :bottom, :capCol, :capRow, :caption, :coldBox, :colorSpec, :hasFocus, :hotBox, :itemCount, :left, :right, :style, :top, :topItem, and :vScroll.

:findText()

Determines the position of an item within a list

Syntax

```
oHbListBox:findText( <cText>          , ;
                     [<nStartPos>]    , ;
```

```
                    [<lCaseSensitive>], ;
                    [<lExact>]          ) --> nItemPos
```

Arguments

<cText>

This is a character string holding the textual information of a list box item to find.

<nStartPos>

This is a numeric value indicating the first item in the list box to begin the search with. It defaults to 1, the first item in the list.

<lCaseSensitive>

This parameter defaults to .T. (true) causing the method to perform a case sensitive search. Setting *<lCaseSensitive>* to .F. (false) searches case insensitive.

<lExact>

If specified, *<lExact>* defines the SET EXACT setting for the string comparison. during the search. The value .T. (true) represents ON, and .F. (false) means OFF.

Description

Method :findText() searches a list box item by its textual representation displayed in the list box. If *<cText>* is found in the internal item list, the method returns a numeric value indicating the ordinal position of the found list box item. Use method :getItem() to retrieve the found list box item.

When *<cText>* cannot be found, the return value is zero.

:getData()

Retrieves the data associated with a list box item.

Syntax

```
:getData( <nItemPos> ) --> xValue
```

Arguments

<nItemPos>

This is a numeric value specifying the ordinal position of the list box item to retrieve the associated data from. Items are numbered from 1 to :itemCount.

Description

List box items are arrays with two elements. The first element holds the textual information displayed in the list box, while the second element contains an associated value.

Method :getData() returns the second element of the list box item specified with *<nItemPos>*, or NIL when an invalid item position is specified.

:getItem()

Retrieves a list box item.

Syntax

```
:getItem( <nItemPos> ) --> aItem
```

Arguments

`<nItemPos>`

This is a numeric value specifying the ordinal position of the list box item to retrieve. Items are numbered from 1 to :itemCount.

Description

List box items are arrays with two elements. The first element holds the textual information displayed in the list box, while the second element contains an associated value.

Method :getItem() returns the array of the list box item specified with <*nItemPos*>, or NIL when an invalid item position is specified.

:getText()

Retrieves the textual information of a list box item.

Syntax

```
:getText( <nItemPos> ) --> cText
```

Arguments

`<nItemPos>`

This is a numeric value specifying the ordinal position of the list box item to retrieve the textual iformation from. Items are numbered from 1 to :itemCount.

Description

List box items are arrays with two elements. The first element holds the textual information displayed in the list box, while the second element contains an associated value.

Method :getText() returns the first element of the list box item specified with <*nItemPos*>, or NIL when an invalid item position is specified.

:hitTest()

Tests if a list box is hit by the mouse cursor.

Syntax

```
:hitTest( <nMouseRow>, <nMouseCol> ) --> nHitCode
```

Arguments

`<nMouseRow>`

This is the numeric row position of the mouse cursor. It can be queried using function MRow().

`<nMouseCol>`

This is the numeric column position of the mouse cursor. It can be queried using function MCol().

Description

Method :hitTest() accepts the numeric row and column position of the mouse cursor and returns a numeric value indicating whether or not the mouse cursor is located within the HbListBox object. #define constants are available in the BUTTON.CH file identifying the return value of :hitTest().

Return values of oHbListBox:hitTest()

Constant	Value	Description
	> 0	Ordinal position of list box item hit by the mouse cursor
HTNOWHERE	0	Mouse cursor is outside the list box
HTTOPLEFT	-1	Mouse cursor is on the top left corner of the list box
HTTOP	-2	Mouse cursor is on the top border of the list box
HTTOPRIGHT	-3	Mouse cursor is on the top right corner of the list box
HTRIGHT	-4	Mouse cursor is on the right border of the list box
HTBOTTOMRIGHT	-5	Mouse cursor is on the bottom right corner of the list box
HTBOTTOM	-6	Mouse cursor is on the bottom border of the list box
HTBOTTOMLEFT	-7	Mouse cursor is on the bottom left corner of the list box
HTLEFT	-8	Mouse cursor is on left border of the list box
HTCAPTION	-1025	Mouse cursor is on the list box's caption
HTDROPBUTTON	-4097	Mouse cursor is on the list box's drop-down button

:insItem()

Inserts a new item into a list box.

Syntax

```
oHbListBox:insItem( <nItemPos>, ;
                     <cText>   , ;
                    [<xValue>]   ) --> self
```

Arguments

<nItemPos>

This is a numeric value specifying the ordinal position where to insert the list box item. Items are numbered from 1 to :itemCount.

<cText>

This is a character string holding the textual information of the list box item to insert.

<xValue>

Optionally, an arbitrary value can be passed which is associated with *<cText>*.

Description

Method :insItem() accepts a text string and an optional associated value which are inserted at position *<nItemPos>* into the internal list of list box items. The return value is the HbListBox object. The instance variable :itemCount reflects the new number of list box items maintained by a HbListBox object.

:killFocus()

Removes input focus from the HbListBox object.

Syntax

```
:killFocus() --> self
```

Description

Method :killFocus() removes the input focus from a HbListBox object and displays it in its normal color.

:nextItem()

Selects the next item in the list box.

Syntax

```
:nextItem() --> self
```

Description

Method :nextItem() changes the currently selected item to the next item in the list box. Use function :select() to query the currently selected list box item.

:open()

Opens an drop-down list box.

Syntax

```
:open() --> self
```

Description

This method opens a drop-down list. This requires the instance variable :dropDown be set to .T. (true). When the object is not configured as a drop-down list box, or when the list box is already dropped down, the :open() method has no effect.

:prevItem()

Selects the previous item in the list box.

Syntax

```
:prevItem() --> self
```

Description

Method :nextItem() changes the currently selected item to the previous item in the list box. Use function :select() to query the currently selected list box item.

:scroll()

Scrolls items in the list box.

Syntax

```
:scroll( <nMode> ) --> self
```

Arguments

<nMode>

This is a numeric value indicating how to scroll the list box. It is usually the return value of the :hitTest() method when a list box is displayed with a vertical scroll bar. The following #define constants from the Button.ch file can be used for *<nMode>*:

Constants for the :scroll() method

Constant	Value	Description
HTSCROLLUNITDEC	-3074	Scroll down one line
HTSCROLLUNITINC	-3075	Scroll up one line

HTSCROLLBLOCKDEC	-3076	Scroll down one page
HTSCROLLBLOCKINC	-3077	Scroll up one page

Description

Method :scroll() scrolls the items visible in the list box window up or down by one line or an entire page. This is usually done in response to a scroll bar being clicked with the mouse when the list box is displayed with a vertical scroll bar.

:select()

Queries or changes the selected item in a list box.

Syntax

```
:select( [<nItemPos>] ) --> nCurrentItemPos
```

Arguments

<nItemPos>

If specified, this is a numeric value defining the ordinal position of the list box item to select as current item. Items are numbered from 1 to :itemCount.

Description

The method returns a numeric value indicating the ordinal position of the currently selected list box item.

If *<nItemPos>* is specified, the corresponding list box item is selected as current item.

:setData()

Changes the data associated with a list box item

Syntax

```
:setData( <nItemPos>, <xValue> ) --> self
```

Arguments

<nItemPos>

This is a numeric value specifying the ordinal position of the list box item to associate new data with. Items are numbered from 1 to :itemCount.

<xValue>

This is an arbitrary value that is associated with the list box item specified with *<nItemPos>*.

Description

List box items are arrays with two elements. The first element holds the textual information displayed in the list box, while the second element contains an associated value.

Method :setData() assigns the value *<xValue>* to the second element of the list box item array specified with *<nItemPos>*.

:setFocus()

Gives input focus to the HbListBox object.

Syntax

```
:setFocus() --> self
```

Description

Method :setFocus() gives input focus to a HbListBox object and displays it in its highlighted color.

:setItem()

Replaces an item in a list box.

Syntax

```
:setItem( <nItemPos>, <aItem> ) --> self
```

Arguments

`<nItemPos>`

This is a numeric value specifying the ordinal position of the the list box item to replace. Items are numbered from 1 to :itemCount.

`<aItem> := { <cText>, <xValue> }`

This is an array with two elements. The first element contains a character string, and the second an arbitrary value.

Description

List box items are arrays with two elements. The first element holds the textual information displayed in the list box, while the second element contains an associated value.

Method :setItem() replaces both, textual information and associated data, of the list box item specified with *<nItemPos>*.

:setText()

Changes the textual information of a list box item.

Syntax

```
:setText( <nItemPos>, <cText> ) --> self
```

Arguments

`<nItemPos>`

This is a numeric value specifying the ordinal position of the the list box item whose textual information should be changed. Items are numbered from 1 to :itemCount.

`<cText>`

This is a character string holding the new textual information if the list box item at position *<nItemPos>*.

Description

List box items are arrays with two elements. The first element holds the textual information displayed in the list box, while the second element contains an associated value.

Method :setText() assigns the string <*cText*> to the first element of the list box item array specified with <*nItemPos*>.

Info

See also:	@...GET, @...GET LISTBOX, Get(), READ, ReadModal(), ReadVar(), Valtype(), SetColor()
Category:	Object functions, Get system
Header:	box.ch, button.ch
Source:	rtl\listbox.prg
LIB:	lib\xhb.lib
DLL:	dll\xhbdll.dll

HBObject()

Abstract base class for user-defined classes.

Description

The HBObject() class serves as base class for all other built-in and user-defined classes declared with the CLASS statement. This class has no member variables but only methods required for creating instances, or objects, of a class (note: an *instance* is an *object* of a class).

All methods of HBObject() can be called in user-defined classes. They define a standard behaviour required in all classes. This standard behaviour can be changed in user-defined classes when methods of HBObject() are re-declared in a new class and re-implemented.

Note: the most commonly re-defined method in user-defined classes is the :init() method.

Methods for object creation and initialization

:new()

Creates a new instance of a class.

Syntax

```
:new( [<params,...>] ) --> oInstance
```

Arguments

`<params,...>`

Optionally, a comma separated list of parameters can be passed to :new(). They are forwarded in the same sequence to the :init() method of the class. The :init() method must be declared in a user-defined class for accessing the passed parameters when an object is initialized in the :init() method.

Return

The method returns a new instance of the class.

Description

Method :new() is the only method that creates instances of a class. The method is special since it can only be used with the class function of a class, not with an object. This is the basic syntax pattern for creating a new object:

```
oNewObject := ClassFunction():new()
```

Note: it is **not** recommended to re-define the :new() method in a user-defined class.

:init()

Initializes a new instance (object) of a class.

Syntax

```
:init( [<params,...>] ) --> self
```

Arguments

`<params,...>`

All parameters passed to the :new() method are forwarded to :init().

Return

The method returns the object *self*.

Description

Method :init() is called from method :new() and serves the purpose of initializing a newly created object. The parameters passed to :new() are forwarded to :init() where the received values are usually stored in instance variables of the new object. An alternative initilization of instance variables is provided with the INIT clause of the DATA declaration of a class.

:initClass()

Initializes class variables of a class.

Syntax

```
:initClass( [<params,...>] ) --> self
```

Arguments

`<params,...>`

All parameters passed to the Class function are forwarded to :initClass().

Return

The method returns the object *self*.

Description

Method :initClass() is implicitly called from the Class function declared with the CLASS declaration. It serves the purpose of initializing class members which exist only once in a class and have the same values for all instances of a class. An alternative initialization of class variables is provided with the INIT clause of the CLASSDATA declaration of a class. This is the recommended way for initializing class variables.

Methods for object inspection

:className()

Retrieves the name of the class an object belongs to.

Syntax

```
:className() --> cClassName
```

Return

The method returns a character string holding the name of the class.

Description

It is often required in object oriented programming to identify the exact class an object belongs to. This is accomplished by retrieving the object's class name with :className().

:classSel()

Retrieves the messages understood by an object.

Syntax

```
:classSel() --> aMessages
```

Return

The method returns a one-dimensional array. Its elements contain character strings holding the symbolic names of the messages declared in the object's class.

!DESRCIPTION Method :classSel() is used to query the symbolic member names of an object's class and super class(es). Member names are declared in the CLASS declaration. They form the messages understood by an object and are declared with DATA or METHOD within the class declaration.

:isDerivedFrom()

Checks if an object belongs to or is derived from a class.

Syntax

```
:isDerivedFrom( <oObject>|<cClassName> ) --> lIsDerived
```

Return

The method returns .T. (true) if the object executing the method belongs to or is derived from the specified class, otherwise .F. (false) is returned.

Description

Method :isDerivedFrom() is a synonym for method :isKindOf(). Refer to this method.

:isKindOf()

Checks if an object is kind of a class.

Syntax

```
:isKindOf( <oObject>|<cClassName> ) --> lIsKindOf
```

Arguments

`<oObject>`

This is a second object which is compared with *self*

`<cClassName>`

Alternatively, the class to compare *self* with can be specified as a character string holding the class name.

Return

The method returns .T. (true) if the object executing the method is kind of the specified class, otherwise .F. (false) is returned.

Description

Method :isKindOf() is used to check if an object is either an instance of the specified class, or if it knows the same member variables and methods. The latter is the case when the class of *<oObject>* is a super class of *self*. A sub-class is always "kind of" a super class.

Methods for error handling

:error()

Creates a runtime error.

Syntax

```
oHBObject:error( <cDescription> , ;
                 <cClassName>   , ;
                 <cMethod>      , ;
                 <nGenCode>     , ;
                 <aArgs>          ) --> xReturn
```

Arguments

`<cDescription>`

This is a character string describing the error (see oError:description).

`<cClassName>`

This is the class name of the object (self:className()).

`<cMethod>`

This is a character string holding the name of the method currently being executed.

`<nGenCode>`

This is a numeric error code (see oError:genCode).

`<aArgs>`

This is an array holding the values involved with the runtime error (see oError:genCode).

Return

The method returns the value of the current error handling code block.

Description

Method :error() provides a convenient way for raising a runtime error within methods. The parameters passed to the method are filled into an Error() object which is passed to the current ErrorBlock().

A simple usage scenario for the :error() method is shown below. It reduces the amount of code for creating an Error() object by a great extent:

```
#include "hbclass.ch"
#include "Error.ch"

PROCEDURE Main
   LOCAL obj := Test():new()

   ? obj:calc( 2, 9 )

   ? obj:calc( "ABCD", 0x77 )
RETURN


CLASS Test
   EXPORTED:
   METHOD Calc
ENDCLASS
```

```
METHOD calc( n1, n2 ) CLASS Test
   IF Valtype( n1 ) + Valtype( n2) <> "NN"
      RETURN ::error( "Type mismatch", ::className(), "calc", EG_ARG, {n1,n2} )
   ENDIF
RETURN n1 * n2
```

:msgNotFound()

Handles unknown messages sent to an object.

Syntax

```
:msgNotFound( <cMessage> ) --> xReturn
```

Arguments

<cMessage>

This is a character string holding the symbolic name sent to the object *self*.

Return

If not overloaded, the method returns the result of the :error() method.

Description

Method :msgNotFound() is called when the symbolic name of a message sent to an object does not exist in the class or its super class(es). It receives as parameter a character string holding the message name that does not exist.

Note: This method cannot be overloaded in a user-defined class with a regular METHOD declaration. :msgNotFound() is only replaced with the ERROR HANDLER declaration.

Info

See also:	CLASS, DATA, ERROR HANDLER, METHOD (declaration)
Category:	Object functions, xHarbour extensions
Source:	rtl\tobject.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

HBPersistent()

Abstract base class for user-defined persistent object.

Description

The HBPersistent() class is an abstract class for the creation of persistent objects. When user-defined classes should become persistent, they must inherit from HBPersistent() which provides methods for converting an object to/from a binary string and/or file on disk.

All instance variables declared with the PERSISTENT attribute are preserved and re-assigned when a persistent object is converted to a binary string and back to the Object data type.

Note: if code blocks are stored in instance variables of a persistent object, the same restrictions apply as outlined with function HB_Serialize().

Methods for saving an object

:saveToFile()

Writes the persistent object data to a file.

Syntax

```
:saveToFile( <cFileName> ) --> lSuccess
```

Arguments

<cFileName>

This is a character string holding the name of the file to create. It must include path and file extension. If the path is omitted from *<cFileName>*, the file is created in the current directory.

Description

Method :saveToFile() creates the file *<cFileName>* and writes into it the binary data of the object executing the method. If the file exists already, it is overwritten. The save data can be copied back to an object using method :loadFromFile()

:saveToText()

Creates a binary string from the data held in an object.

Syntax

```
:saveToText() --> cBinaryString
```

Description

Method :saveToText() copies the data held in instance variables of the object into a binary string, which is returned. The object can copy the data back to its instance variables using method :loadFromText().

Methods for restoring an object

:loadFromFile()

Loads persistent object from a file.

Syntax

```
:loadFromFile( <cFileName> ) --> lSuccess
```

Arguments

```
<cFileName>
```

This is a character string holding the name of the file to read. It must include path and file extension. If the path is omitted from *<cFileName>*, the file is searched in the current directory.

Description

Method :loadFromFile() reads binary data from the file *<cFileName>* and copies it to the object executing the method. The file must be created by the method :saveToFile() If the file does not exists, a runtime error is raised.

:loadFromText()

Copies data held in a binary string to an object.

Syntax

```
:loadFromText( <cBinaryString> ) --> lSuccess
```

Description

Method :loadFromText() copies the data held in *<cBinaryString>* to the instance variables of the object. Only instance variables declares as PERSISTENT receive values from the binary string. The latter must be created with method :saveToText().

Info

See also:	CLASS, DATA, HB_Serialize()
Category:	Object functions, xHarbour extensions
Source:	rtl\persist.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates the steps required to make objects
// persistent. Only the instance variables with the PERSISTENT
// attribute are written to and restored from the file.

   #include "HbClass.ch"

   PROCEDURE Main
      LOCAL obj, cPersistent

      obj := test():new()
      obj:volatile := "Volatile"
```

```
   obj:show()

   obj:value := "Data from file"


   obj:saveToFile( "Persist.txt" )

   obj := NIL

   WAIT "obj is gone, press a key..."

   obj := Test():new()
   obj:loadFromFile( "Persist.txt" )

   obj:show()

   ? "Elapsed:",  Seconds()-obj:secs
RETURN


CLASS Test FROM HBPersistent
   DATA value    INIT "Test persistent Object"  PERSISTENT
   DATA date     INIT Date()                    PERSISTENT
   DATA time     INIT Time()                    PERSISTENT
   DATA secs     INIT Seconds()                 PERSISTENT
   DATA volatile
   METHOD show
ENDCLASS

METHOD Show
   ? ::value
   ? "Creation Date:", ::date
   ? "Creation Time:", ::time
   ? "    Timestamp:", ::secs
   ? "Volatile data:", ::volatile
RETURN
```

HbPushButton()

Creates a new HbPushButton object.

Syntax

```
HbPushButton():new( <nRow>, <nCol>, [<cCaption>] ) --> oHbPushButton
```

Arguments

`<nRow>`

This is the numeric row coordinate on the screen where the HbPushButton object is displayed. *<nRow>* is assigned to oPushButton:row.

`<nCol>`

This is the numeric column coordinate on the screen where the HbPushButton object is displayed. *<nCol>* is assigned to oHbPushButton:col.

`<cCaption>`

This is an optional character string holding the caption of the push button. *<cCaption>* is assigned to oHbPushButton:caption.

Return

Function HbPushButton() returns a new HbPushButton object and method :new() initializes the object.

Description

Objects of the HbpushButton class are used in text mode applications to display a push button. They are designed to be integrated into the standard Get system and are usually created with the @...GET PUSHBUTTON command. This command creates a Get() object and an associated HbPushButton object. A push button can then be activated along with other Get object by the READ command.

Instance variables

:buffer

Logical edit buffer indicating the pressed/released state.

Data type: L
Default: .F.

Description

The instance variable :buffer represents the edit buffer of a HbPushButton object. This buffer contains either .T. (true) for the Pressed state, or .F. (false) for the Released state.

:caption

Character string defining the caption.

Data type: C
Default: ""

Description

The instance variable :caption can be assigned a character string to be displayed at position :row and :col whithin the :display() method. If the :caption string contains an ampersand (&), the ampersand is

removed from the :caption string and the next character to the right of the ampersand is used as accelerator key while READ is active. The accelerator is displayed in a separate color (see :colorSpec).

:cargo

Instance variable for user-defined purposes.

Data type: ANY
Default: NIL

Description

This instance variable is not used by a HbPushButton object. It is available for user-defined purposes when an arbitrary value should be attached to the object.

:col

Numeric column position of the HbPushButton object.

Data type: N
Default: NIL

Description

The instance variable :col contains the numeric column position where a HbPushButton object displays the push button on the screen. :col receives the second parameter passed to the :new() method.

:colorSpec

Color string for displaying a push button.

Data type: C
Default: W/N,N/W,W+/N,W+/N

Description

The instance variable :colorSpec contains a SetColor() string holding four color values. They are used for the following purposes:

Color values for push buttons

Color value	Description
1	Color when the push button does not have input focus
2	Color when the push button has input focus
3	Color for the push button is pressed
4	Color for the push button's accelerator key

:fBlock

Code block evaluated when input focus changes.

Data type: B
Default: NIL

Description

The instance variable :fBlock can be assigned a code block. It is evaluated during the standard READ command when the HbPushButton object receives input focus. The code block is evaluated without parameters.

:hasFocus

Indicates if the HbPushButton object has input focus.

Data type: L
Default: .F.

Description

The instance variable :hasFocus contains .T. (true) when a HbPushButton object has input focus, otherwise .F. (false).

:message

Character string to be displayed as message when the push button receives input focus.

Data type: C
Default: ""

Description

The instance variable :message can be assigned a character string which is displayed in the message area defined with the MSG AT option of the READ command. The message is displayed when the HbPushButton object receives input focus.

:row

Numeric row position of the HbRadioButton object.

Data type: N
Default: NIL

Description

The instance variable :row contains the numeric row position where a HbPushButton object displays the push buttton on the screen. :row receives the first parameter passed to the :new() method.

:sBlock

Code block evaluated when the pressed/released state changes.

Data type: B
Default: NIL

Description

The instance variable :sBlock can be assigned a code block. It is evaluated during the standard READ command when the pressed/released state of the HbPushButton object changes. The code block is evaluated without parameters.

:style

Character string defining the push button delimiters.

Data type: C
Default: <>

Description

The display style of a push button can be specified with an optional character string of zero, two or eight characters. When a null string is assigned to :style, no delimiting characters are used.

A character string of two characters defines the left and right delimiters of the pus button.

A character string of eight characters defines the border drawn around the pushbutton. They are used in the same way as with the DispBox() function. The #define constants available in Box.ch can be used:

Constants for borders

Constant	Description
B_SINGLE	Single-line border
B_DOUBLE	Double-line border
B_SINGLE_DOUBLE	Single-line top, double-line sides
B_DOUBLE_SINGLE	Double-line top, single-line sides

:typeOut

Logical value .F. (false).

Data type: L
Default: .F.

Description

The instance variable :typeOut contains always .F. (false). It is required as place holder in the standard Get system.

Methods

:display()

Displays the push button on the screen.

Syntax

```
:display() --> self
```

Description

Method :display() displays the push button and takes care of the pressed/released state and different colors depending on the input focus. It uses the instance variables :buffer, :row, :col, :caption, :hasFocus, :colorSpec and :style.

:hitTest()

Tests if a push button is hit by the mouse cursor.

Syntax

```
:hitTest( <nMouseRow>, <nMouseCol> ) --> nHitCode
```

Arguments

<nMouseRow>

This is the numeric row position of the mouse cursor. It can be queried using function MRow().

<nMouseCol>

This is the numeric column position of the mouse cursor. It can be queried using function MCol().

Description

Method :hitTest() accepts the numeric row and column position of the mouse cursor and returns a numeric value indicating whether or not the mouse cursor is located within the HbPushButton object. #define constants are available in the BUTTON.CH file identifying the return value of :hitTest().

Return values of oHbPushButton:hitTest()

Constant	Value	Description
HTNOWHERE	0	Mouse cursor is outside the push button
HTTOPLEFT	-1	Mouse cursor is on the top left corner of the push button
HTTOP	-2	Mouse cursor is on the top border of the push button
HTTOPRIGHT	-3	Mouse cursor is on the top right corner of the push button
HTRIGHT	-4	Mouse cursor is on the right border of the push button
HTBOTTOMRIGHT	-5	Mouse cursor is on the bottom right corner of the push button
HTBOTTOM	-6	Mouse cursor is on the bottom border of the push button
HTBOTTOMLEFT	-7	Mouse cursor is on the bottom left corner of the push button
HTLEFT	-8	Mouse cursor is on left border of the push button
HTCLIENT	-2049	Mouse cursor is on the push button

:killFocus()

Removes input focus from the HbPushButton object.

Syntax

```
:killFocus() --> self
```

Description

Method :killFocus() removes the input focus from a HbPushButton object and displays it in its normal color.

:select()

Activates a push button.

Syntax

```
:select( [<nKey>] ) --> self
```

Arguments

<nKey>

This is a numeric value representing the key code of the key activating the push button. The key code is returned by the Inkey() function. To specify values for this parameter use #define constants from the file INKEY.CH.

Description

Method :select() activates a push button when it has input focus. The method waits for *<nKey>* until it is retrieved a second time from the Inkey() function. Normally, a push button is activated by the Space bar, Enter key or when clicked with the mouse.

:setFocus()

Gives input focus to the HbPushButton object.

Syntax

```
:setFocus() --> self
```

Description

Method :setFocus() gives input focus to a HbPushButton object and displays it in its highlighted color.

Info

See also:	@...GET, @...GET PUSHBUTTON, Get(), READ, ReadModal(), ReadVar(), Valtype(), SetColor()
Category:	Object functions, Get system
Source:	rtl\pushbtn.prg
LIB:	lib\xhb.lib
DLL:	dll\xhbdll.dll

HbRadioButton()

Creates a new HRadioButton object.

Syntax

```
HbRadioButton():new( <nRow>     , ;
                     <nCol>     , ;
                    [<cCaption>], ;
                    [<xValue>     ) --> oHbRadioButton
```

Arguments

`<nRow>`

This is the numeric row coordinate on the screen where the HbRadioButton object is displayed. *<nRow>* is assigned to oRadioButton:row.

`<nCol>`

This is the numeric column coordinate on the screen where the HbRadioButton object is displayed. *<nCol>* is assigned to oHbRadioButton:col.

`<cCaption>`

This is an optional character string holding the caption of the radio button. *<cCaption>* is assigned to oHbRadioButton:caption.

`<xValue>`

This is an optional value of any data type to associate with the radio button object.

Return

Function HbRadioButton() returns a new HbRadioButton object and method :new() initializes the object.

Description

Objects of the HbRadioButton class are used in text mode applications to display and edit a single state from multiple options. A single HbRadioButton object maintains a single state. Radio buttons must be added to a HbRadioGroup() object, which maintains multipe radio buttons. Both classes are designed to be integrated into the standard Get system and are usually created with the @...GET RADIOGROUP command. This command creates a Get() object and an associated HbRadioGroup object. Editing the edit buffer of the radio button group is then accomplished by the READ command.

Instance variables

:buffer

Logical edit buffer indicating the (un)selected state.

Data type: L
Default: .F.

Description

The instance variable :buffer represents the edit buffer of a HbRadioButton object. This buffer contains either .T. (true) for the Selected state, or .F. (false) for the Unselected state.

Only one radio button in a group of radio buttons can be selected.

:caption

Character string defining the caption.

Data type: C
Default: NIL

Description

The instance variable :caption can be assigned a character string to be displayed at position :capRow and :capCol whithin the :display() method. If the :caption string contains an ampersand (&), the ampersand is removed from the :caption string and the next character to the right of the ampersand is used as accelerator key while READ is active. The accelerator is displayed in a separate color (see :colorSpec).

:capCol

Numeric column position of the caption.

Data type: N
Default: :col+4

Description

The instance variable :capCol contains the numeric column position of the caption of a HbRadioButton object. It is only meaningful when instance variable :caption contains a character string to display as a caption.

:capRow

Numeric row position of the caption.

Data type: N
Default: :row

Description

The instance variable :capRow contains the numeric row position of the caption of a HbRadioButton object. It is only meaningful when instance variable :caption contains a character string to display as a caption.

:cargo

Instance variable for user-defined purposes.

Data type: ANY
Default: NIL

Description

This instance variable is not used by a HbRadioButton object. It is available for user-defined purposes when an arbitrary value should be attached to the object.

:col

Numeric column position of the HbRadioButton object.

Data type: N
Default: NIL

Description

The instance variable :col contains the numeric column position where a HbRadioButton object displays the radio button box on the screen. :col receives the second parameter passed to the :new() method.

:colorSpec

Color string for display and selecting.

Data type: C
Default: W/N,W+/N,W+/N,N/W,W/N,W/N,W+/N

Description

The instance variable :colorSpec contains a SetColor() string holding seven color values. They are used for the following purposes:

Color values for radio buttons

Color value	Description
1	Color for unselected radio button when it does not have input focus
2	Color for selected radio button when it does not have input focus
3	Color for unselected radio button when it has input focus
4	Color for selected radio button when it has input focus
5	Color for the caption
6	Color for the accelerator without input focus
7	Color for the accelerator with input focus

:fBlock

Code block evaluated when input focus changes.

Data type: B
Default: NIL

Description

The instance variable :fBlock can be assigned a code block. It is evaluated during the standard READ command when the HbRadioButton object receives input focus. The code block is evaluated without parameters.

:hasFocus

Indicates if the HbRadioButton object has input focus.

Data type: L
Default: .F:

Description

The instance variable :hasFocus contains .T. (true) when a HbRadioButton object has input focus, otherwise .F. (false).

:row

Numeric row position of the HbRadioButton object.

Data type: N
Default: NIL

Description

The instance variable :row contains the numeric row position where a HbRadioButton object displays the radio button on the screen. :row receives the first parameter passed to the :new() method.

:sBlock

Code block evaluated when the (un)selected state changes.

Data type: B
Default: NIL

Description

The instance variable :sBlock can be assigned a code block. It is evaluated during the standard READ command when the (un)selected state of the HbRadioButton object changes. The code block is evaluated without parameters.

:style

Character string defining the radio button delimiters.

Data type: C
Default: (*)

Description

The display style of a radio button can be specified with an optional character string of four characters. They are used as follows:

Characters for the display style

Position	Description
1	Left delimiter of radio button
2	Character for the Selected state
3	Character for the Unselected state
4	Right delimiter of radio button

Methods

:display()

Displays the radio button on the screen.

Syntax

```
:display() --> self
```

Description

Method :display() displays the radio button and takes care of the (un)selected state and different colors depending on the input focus. It uses the instance variables :buffer, :row, :col, :capRow, :capCol, :caption, :hasFocus, :colorSpec and :style.

:hitTest()

Tests if a radio button is hit by the mouse cursor.

Syntax

```
:hitTest( <nMouseRow>, <nMouseCol> ) --> nHitCode
```

Arguments

<nMouseRow>

This is the numeric row position of the mouse cursor. It can be queried using function MRow().

<nMouseCol>

This is the numeric column position of the mouse cursor. It can be queried using function MCol().

Description

Method :hitTest() accepts the numeric row and column position of the mouse cursor and returns a numeric value indicating whether or not the mouse cursor is located within the HbRadioButton object. #define constants are available in the BUTTON.CH file identifying the return value of :hitTest().

Return values of oHbRadioButton:hitTest()

Constant	Value	Description
HTNOWHERE	0	Mouse cursor is outside the radio button and its caption
HTCAPTION	-1025	Mouse cursor is inside the radio button's caption
HTCLIENT	-2049	Mouse cursor is inside the radio button

:isAccel()

Checks if an accelerator key is pressed.

Syntax

```
:isAccel( <nKey>|<cChr> ) --> lIsAccelerator
```

Arguments

<nKey>

This is a numeric Inkey() code to test.

<cChr>

Alternatively, a single character can be passed.

Description

Method :isAccel() tests if the passed key code identifys the accelerator key of a HbRadiohButton object. See :caption for the definition of an accelerator key.

The return value is .T. (true) when the passed parameter identifies the accelerator, otherwise .F. (false) is returned.

:killFocus()

Removes input focus from the HbRadioButton object.

Syntax

```
:killFocus() --> self
```

Description

Method :killFocus() removes the input focus from a HbRadioButton object and displays it in its normal color.

:select()

Toggles or defines the (un)cheked state.

Syntax

```
:select( [<lSelected>] ) --> self
```

Arguments

<lSelected>

This is an optional logical value defining the Selected (.T.) or Unselected (.F.) state.

Description

When a parameter is passed to the :select() method, it defines the new state of a HbRadioButton object. Calling the method without parameter toggles its state from Selected to Unselected and vice versa.

:setFocus()

Gives input focus to the HbRadioButton object.

Syntax

```
:setFocus() --> self
```

Description

Method :setFocus() gives input focus to a HbRadioButton object and displays it in its highlighted color.

Info

See also:	@...GET, @...GET RADIOGROUP, Get(), READ, ReadModal(), ReadVar(), Valtype(), SetColor()
Category:	Object functions, Get system
Source:	rtl\radiobtn.prg
LIB:	lib\xhb.lib
DLL:	dll\xhbdll.dll

HbRadioGroup()

Creates a new HbRadioGroup object.

Syntax

```
HbRadioGroup():new( <nTop>    , ;
                    <nLeft>   , ;
                    <nBottom>, ;
                    <nRight>    ) --> oHbRadioGroup
```

Arguments

`<nTop>, <nLeft>, <nBottom>, <nRight>`

These numeric parameters indicate the screen coordinates for the upper left and lower right corner of the radio button group output. The range for rows on the screen is 0 to MaxRow(), and for columns it is 0 to MaxCol(). The coordinate 0,0 is the upper left corner of the screen. !END

Return

Function HbRadioGroup() returns a new HbRadioGroup object and method :new() initializes the object.

Description

Objects of the HbRadioGroup class are used in text mode applications to display multiple options and select one of it. A single option is represented by a HbRadioButton() object. Radio buttons must be added to a HbRadioGroup object with its :addItem() method. A radio group object maintains multipe radio buttons. Both classes are designed to be integrated into the standard Get system and are usually created with the @...GET RADIOGROUP command. This command creates a Get() object and an associated HbRadioGroup object. Editing the edit buffer of the radio button group is then accomplished by the READ command.

Instance variables

:bottom

Numeric bottom row coordinate of the radio button group display.

Data type: N
Default: NIL

Description

The instance variable :bottom contains the numeric bottom row position of the rectangle where a HbRadioGroup object displays its individual radio buttons on the screen. :bottom receives the third parameter passed to the :new() method.

:buffer

Numeric ordinal position of the selected radio button.

Data type: N
Default: NIL

Description

The instance variable :buffer represents the edit buffer of a HbRadioGroup object. This buffer contains a numeric value indicating the ordinal position of the currently selected radio button in the radio group.

:capCol

Numeric column position of the caption.

Data type: N
Default: :left+2

Description

The instance variable :capCol contains the numeric column position of the caption of a HbRadioGroup object. It is only meaningful when instance variable :caption contains a character string to display as a caption.

:capRow

Numeric row position of the caption.

Data type: N
Default: :top

Description

The instance variable :capRow contains the numeric row position of the caption of a HbRadioGroup object. It is only meaningful when instance variable :caption contains a character string to display as a caption.

:caption

Character string defining the caption.

Data type: C
Default: NIL

Description

The instance variable :caption can be assigned a character string to be displayed at position :capRow and :capCol whithin the :display() method. If the :caption string contains an ampersand (&), the ampersand is removed from the :caption string and the next character to the right of the ampersand is used as accelerator key while READ is active. The accelerator is displayed in a separate color (see :colorSpec).

:cargo

Instance variable for user-defined purposes.

Data type: ANY
Default: NIL

Description

This instance variable is not used by a HbRadioGroup object. It is available for user-defined purposes when an arbitrary value should be attached to the object.

:coldBox

Box string for border when the radio button group has no input focus.

Data type: C
Default: B_SINGLE

Description

The instance variable :coldBox contains a DispBox() compatible character string defining the border drawn around the radio button group when the HbRadioGroup object has no input focus. #define constants are available in the Box.ch file that can be used for :coldBox.

Pre-defined box strings for borders

Constant	Description
B_SINGLE	Single-line box
B_DOUBLE	Double-line box
B_SINGLE_DOUBLE	Single-line top, double-line sides
B_DOUBLE_SINGLE	Double-line top, single-line sides

:colorSpec

Color string for display and selection.

Data type: N
Default: W/N,W/N,W+/N

Description

The instance variable :colorSpec contains a SetColor() string holding three color values. They are used for the following purposes:

Color values for radio button groups

Color value	Description
1	Color for the border around the radio button group
2	Color for the caption
3	Color for the accelerator key

:fBlock

Code block evaluated when input focus changes.

Data type: B
Default: NIL

Description

The instance variable :fBlock can be assigned a code block. It is evaluated during the standard READ command when the HbRadioGroup object receives input focus. The code block is evaluated without parameters.

:hasFocus

Indicates if the HbRadioGroup object has input focus.

Data type: L
Default: .F.

Description

The instance variable :hasFocus contains .T. (true) when a HbRadioGroup object has input focus, otherwise .F. (false).

:hotBox

Box string for border when radio button group box has input focus.

Data type: N
Default: B_DOUBLE

Description

The instance variable :hotBox contains a DispBox() compatible character string defining the border drawn around the radio button group when the HbRadioGroup object has input focus. #define constants are available in the Box.ch file that can be used for :hotBox.

Pre-defined box strings for borders

Constant	Description
B_SINGLE	Single-line box
B_DOUBLE *)	Double-line box
B_SINGLE_DOUBLE	Single-line top, double-line sides
B_DOUBLE_SINGLE	Double-line top, single-line sides

**) default*

:itemCount

Numeric value indicating the total number of radio buttons in the group.

Data type: N
Default: 0

Description

The instance variable :itemCount contains a numeric value representing the number of radio buttons maintained by a HbRadioGroup object. Radio buttons are added to the radio group with the :addItem() method.

:left

Numeric left column coordinate of the radio button group display.

Data type: N
Default: NIL

Description

The instance variable :left contains the numeric left column position of the rectangle where a HbRadioGroup object displays its individual radio buttons on the screen. :left receives the second parameter passed to the :new() method.

:message

Character string to be displayed as message when the radio button group receives input focus.

Data type: C
Default: ""

Description

The instance variable :message can be assigned a character string which is displayed in the message area defined with the MSG AT option of the READ command. The message is displayed when the HbRadioGroup object receives input focus.

:right

Numeric right column coordinate of the radio button group display.

Data type: N
Default: NIL

Description

The instance variable :right contains the numeric right column position of the rectangle where a HbRadioGroup object displays its individual radio buttons on the screen. :right receives the fourth parameter passed to the :new() method.

:top

Numeric top row coordinate of the radio button group display.

Data type: N
Default: NIL

Description

The instance variable :top contains the numeric top row position of the rectangle where a HbRadioGroup object displays its individual radio buttons on the screen. :top receives the first parameter passed to the :new() method.

:typeOut

Logical value indicating if a radio button group is empty.

Data type: L
Default: .F.

Description

The instance variable :typeOut contains .T. (true) when a HbRadioGroup object is empty and has no radio buttons, otherwise .F. (false).

Methods

:addItem()

Adds a new radio button to a radio group

Syntax

```
:addItem( <oHbRadioButton> ) --> self
```

Arguments

<oHbRadioButton>

This is a HbRadioButton() object to be added to the list of radio buttons.

Description

Method :addItem() accepts a HbRadioButton object and adds it to the internal list of radio buttons. The return value is the HbRadioGroup object. The instance variable :itemCount reflects the new number of radio buttons maintained by a HbRadioGroup object.

:delItem()

Removes a radio button from a HbRadioGroup object.

Syntax

```
:delItem( <nPos> ) --> self
```

Arguments

<nPos>

This is a numeric value specifying the ordinal position of the radio button to remove from the radio group. Radio buttons are numbered from 1 to :itemCount.

Description

Method :delItem() is used to remove a single radio button from a HbRadioGroup object. The method returns the HbRadioGroup object. Use method :getItem() to preserve the radio button before it is deleted.

:display()

Displays the radio button group on the screen.

Syntax

```
:display() --> self
```

Description

Method :display() displays the radio button group and takes care of the different colors depending on the input focus. It uses the instance variables :bottom, :capCol, :capRow, :caption, :coldBox, :colorSpec, :hasFocus, :hotBox, :itemCount, :left, :right and :top.

When the HbRadioGroup object has displayed itself, it iterates the internal list of HbRadioButtton objects and calls their :display() method.

:getAccel()

Checks if an accelerator key is pressed for a radio button.

Syntax

```
:getAccel( <nKey>|<cChr> ) --> nPos
```

Arguments

<nKey>

This is a numeric Inkey() code to test.

<cChr>

Alternatively, a single character can be passed.

Description

Method :getAccel() tests if the passed key code identifys the accelerator key of a HbRadioButton object stored in the internal list of radio buttons. See :caption for the definition of an accelerator key.

The method returns a numeric value indicating the ordinal position of the radio button whose accelerator matches *<nKey>*|*<cKey>*. When no radio button has the specified accelerator, the return value is zero.

:getItem()

Retrieves a radio button object.

Syntax

```
:getItem( <nPos> ) --> oHbRadioButton
```

Arguments

<nPos>

This is a numeric value specifying the ordinal position of the radio button to retrieve. Radio buttons are numbered from 1 to :itemCount.

Description

Method :getItem() returns the HbRadioButton() object stored at position *<nPos>* in the internal list of radio buttons, or NIL when an invalid position is specified.

:hitTest()

Tests if a radio group is hit by the mouse cursor.

Syntax

```
:hitTest( <nMouseRow>, <nMouseCol> ) --> nHitCode
```

Arguments

<nMouseRow>

This is the numeric row position of the mouse cursor. It can be queried using function MRow().

<nMouseCol>

This is the numeric column position of the mouse cursor. It can be queried using function MCol().

Description

Method :hitTest() accepts the numeric row and column position of the mouse cursor and returns a numeric value indicating whether or not the mouse cursor is located within the HbRadioGroup object. #define constants are available in the BUTTON.CH file identifying the return value of :hitTest().

Return values of oHbRadioGroup:hitTest()

Constant	Value	Description
	> 0	Ordinal position of radio button hit by the mouse cursor
HTNOWHERE	0	Mouse cursor is outside the radio group
HTTOPLEFT	-1	Mouse cursor is on the top left corner of the radio group
HTTOP	-2	Mouse cursor is on the top border of the radio group
HTTOPRIGHT	-3	Mouse cursor is on the top right corner of the radio group
HTRIGHT	-4	Mouse cursor is on the right border of the radio group
HTBOTTOMRIGHT	-5	Mouse cursor is on the bottom right corner of the radio group
HTBOTTOM	-6	Mouse cursor is on the bottom border of the radio group
HTBOTTOMLEFT	-7	Mouse cursor is on the bottom left corner of the radio group
HTLEFT	-8	Mouse cursor is on left border of the radio group
HTCAPTION	-1025	Mouse cursor is on the radio group's caption

:insItem()

Inserts a new radio button into a radio group

Syntax

```
:insItem( <nPos>, <oHbRadioButton> ) --> oHbRadioButton
```

Arguments

<nPos>

This is a numeric value specifying the ordinal position where to insert the radio button. Radio buttons are numbered from 1 to :itemCount.

<oHbRadioButton>

This is a HbRadioButton() object to be inserted into the list of radio buttons.

Description

Method :insItem() accepts a HbRadioButton object and inserts it at position <*nPos*> into the internal list of radio buttons. The return value is the inserted HbRadioButton object. The instance variable :itemCount reflects the new number of radio buttons maintained by a HbRadioGroup object.

:killFocus()

Removes input focus from the HbRadioGroup object.

Syntax

```
:killFocus() --> self
```

Description

Method :killFocus() removes the input focus from a HbRadioGroup object and displays it in its normal color.

:nextItem()

Selects the next radio button in the radio group.

Syntax

```
:nextItem() --> self
```

Description

Method :nextItem() changes the currently selected radio button to the next radio button in the radio group. This updates the instance variable :buffer to reflect the ordinal position of the currently selected radio button.

:prevItem()

Selects the previous radio button in the radio group.

Syntax

```
:prevItem() --> self
```

Description

Method :prevItem() changes the currently selected radio button to the previous radio button in the radio group. This updates the instance variable :buffer to reflect the ordinal position of the currently selected radio button.

:select()

Changes the Selected sate of a radio button in a radio group.

Syntax

```
:select( <nPos> ) --> self
```

Arguments

`<nPos>`

This is a numeric value specifying the ordinal position of the radio button to select. Radio buttons are numbered from 1 to :itemCount.

Description

Method :select() changes the state of the currently selected radio button to Unselected, and the state of the radio button at the ordinal position <*nPos*> to Selected.

The return value is the HbRadioGroup object.

:setColor()

Changes the color of all radio buttons in a group.

Syntax

```
:setColor( <cColor> ) --> self
```

Arguments

<cColor>

This is a SetColor() color string with seven color values.

Description

Method :setColor() iterates the internal list of radio buttons and assigns the color string <*cColor*> to :colorSpec of all radio buttons.

The return value is the HbRadioGroup object.

:setFocus()

Gives input focus to the HbRadioGroup object.

Syntax

```
:setFocus() --> self
```

Description

Method :setFocus() gives input focus to a HbRadioGroup object and displays it in its highlighted color.

:setStyle()

Changes the style attribute of all radio buttons in a group.

Syntax

```
:setStyle( <cStyle> ) --> self
```

Arguments

<cStyle>

This is a character string with style attributes for all radio buttons in the group.

Description

Method :setStyle() iterates the internal list of radio buttons and assigns the string <*cStyle*> to :style of all radio buttons.

The return value is the HbRadioGroup object.

Info

See also:	@...GET, @...GET RADIOGROUP, Get(), READ, ReadModal(), ReadVar(), Valtype(), SetColor()
Category:	Object functions, Get system
Header:	button.ch
Source:	rtl\radiogrp.prg
LIB:	lib\xhb.lib
DLL:	dll\xhbdll.dll

HbScrollBar()

Creates a new HbScrollBar object.

Syntax

```
HbScrollBar():new( <nTop>     , ;
                   <nBottom> , ;
                   <nCol>    , ;
                  [<bScroll>], ;
                  [<nType>]    ) --> oHbScrollBarVertical
or

HbScrollBar():new( <nLeft>    , ;
                   <nRight>   , ;
                   <nRow>     , ;
                  [<bScroll>], ;
                   <nType>     ) --> oHbScrollBarHorizontal
```

Arguments

`<nTop>, <nBottom> and <nCol>`

These numeric parameters are the top and bottom row coordinates plus the screen column of a vertical scroll bar. If *<nType>* is not specified, a vertical scroll bar is created by default. *<nTop>* is assigned to oHbScrollBar:start, *<nBottom>* to oHbScrollBar:end and *<nCol>* to oHbScrollBar:offset.

`<nLeft>, <nRight> and <nRow>`

These numeric parameters are the left and right colum coordinates plus the screen row of a horizontal scroll bar. Set *<nType>* to 2 to create a horizontal scroll bar. *<nLeft>* is assigned to oHbScrollBar:start, *<nRight>* to oHbScrollBar:end and *<nRow>* to oHbScrollBar:offset.

`<bScroll>`

Optionally, a code block can be passed. It is evaluated each time the state of the scroll bar changes. It is assigned to oHbScrollBar:sBlock.

`<nType>`

This parameter defaults to 1, which creates a vertical scroll bar. Setting *<nType>* to 2 creates a horizontal scroll bar. *<nType>* is assigned to oHbScrollBar:orient.

Return

Function HbScrollBar() returns a new HbScrollBar object and method :new() initializes the object.

Description

Objects of the HbScrollBar class are used in text mode applications to display a scroll bar. They are usually created with the SCROLLBAR option of the extended @...GET LISTBOX command and are displayed in the border of a HbListBox() object.

Instance variables

:barLength

Numeric height/width of a vertical/horizontal scroll bar.

Data type: N
Default: NIL

Description

The instance variable :barLength contains a numeric value indicating the height of a vertical (width of a horizontal) scroll bar.

:cargo

Instance variable for user-defined purposes.

Data type: ANY
Default: NIL

Description

This instance variable is not used by a HbScrollBar object. It is available for user-defined purposes when an arbitrary value should be attached to the object.

:colorSpec

Color string for scrol bar display.

Data type: C
Default: SetColor()

Description

The instance variable :colorSpec contains a SetColor() string holding two color values. They are used for the following purposes:

Color values for scroll bars

Color value	Description
1	Color for the scroll bar
2	Color for the scroll bar arrows and thumb

:current

Numeric value indicating the current item the scroll bar refers to.

Data type: N
Default: 0

Description

The instance variable :current holds a numeric value indicating the current item the acroll bar refers to. The range for :current is between 1 and :total.

:end

Numeric screen position of the scroll bar's Next arrow.

Data type: N
Default: 0

Description

The instance variable :end contains a numeric value indicating the row position of the Next arrow at the bottom of a vertical scroll bar, or its column position on the right side of a horizontal scroll bar.

:offset

Numeric screen position of the scroll bar.

Data type: N
Default: 9

Description

The instance variable :offset contains a numeric value indicating the column position of of a vertical scroll bar, or the row position of a horizontal scroll bar.

:orient

Indicates vertical/horizontal orientation of the scroll bar.

Data type: N
Default: 1

Description

The default orientation of a scroll bar is vertical (:orient==1). Setting :orient to 2 displays a horizontal scroll bar. Other values are ignored.

:sBlock

Code block evaluated when the scroll bar's state changes.

Data type: B
Default: NIL

Description

The instance variable :sBlock can be assigned a code block. It is evaluated during the standard READ command when the scroll state of the HbScrollBar object changes. The code block is evaluated without parameters.

:start

Numeric screen position of the scroll bar's Previous arrow.

Data type: N
Default: 0

Description

The instance variable :start contains a numeric value indicating the row position of the Previous arrow at the top of a vertical scroll bar, or its column position on the left side of a horizontal scroll bar.

:style

Indicates characters used by the ScrollBar:display() method

Data type: C

Description

The display style of a scroll bar can be specified with an optional character string of three characters. They are used as follows:

Characters for the display style

Position	Description
1	Previous arrow
2	Character for the client area state
3	Character for the scroll bar thumb
4	Next arrow

:thumbPos

Numeric relative position of the scroll bar thumb

Data type: N
Default: 1

Description

The instance variable :thumbPos contains a numeric value indicating the relative position of the scroll bar's thumb.

:total

Total number of items the scroll bar refers to.

Data type: N
Default: 100

Description

The instance variable :total holds a numeric value indicating the total number of item the acroll bar refers to.

Methods

:display()

Displays a scroll bar, including its thumb and arrows, on the screen

Syntax

```
:display() --> self
```

Description

Method :display() displays the scroll bar button and takes care of its orientation and different colors depending on the arrows, client area and thumb. It uses the instance variables :start, :end, :offset :orient, :thumbPos, :colorSpec and :style.

:update()

Updates the thumb position of a scroll bar on the screen

Syntax

```
:update() --> lIsUpdated
```

Description

Method :update() calculates the thumb position of the scroll bar according to the values in :current and :total and updates it on screen.

The return value is .T. (true) when the thumb position has changes, otherwise .F. (false) is returned.

:hitTest()

Tests if a scroll bar is hit by the mouse cursor.

Syntax

```
:hitTest( <nRow>, <nCol> ) --> nHitCode
```

Arguments

`<nMouseRow>`

This is the numeric row position of the mouse cursor. It can be queried using function MRow().

`<nMouseCol>`

This is the numeric column position of the mouse cursor. It can be queried using function MCol().

Description

Method :hitTest() accepts the numeric row and column position of the mouse cursor and returns a numeric value indicating whether or not the mouse cursor is located within the HbScrollBar object. #define constants are available in the BUTTON.CH file identifying the return value of :hitTest().

Return values of oHbScrollBar:hitTest()

Constant	Value	Description
HTNOWHERE	0	Mouse cursor is outside the scroll bar
HTSCROLLTHUMBDRAG	-3073	Mouse cursor is on the thumb
HTSCROLLUNITDEC	-3074	Mouse cursor is on the Previous arrow
HTSCROLLUNITINC	-3075	Mouse cursor is on the Next arrow
HTSCROLLBLOCKDEC	-3076	Mouse cursor is between the Previous arrow and the thumb
HTSCROLLBLOCKINC	-3077	Mouse cursor is between the thumb and the Next arrow

Info

See also: @...GET, @...GET LISTBOX, Get(), READ, ReadModal(), ReadVar(), Valtype(), SetColor()
Category: Object functions, Get system
Header: button.ch
Source: rtl\scrollbr.prg
LIB: lib\xhb.lib
DLL: dll\xhbdll.dll

MenuItem()

Creates a new MenuItem object.

Syntax

```
MenuItem():new( <cCaption>       , ;
                <bBlock>|<oPopup>, ;
                [<nShortcut>]    , ;
                [<cMessage>]     , ;
                [<nID>]            ) --> oMenuItem
```

Arguments

`<cCaption>`

This is a character string displayed as the caption of a menu item. An accelerator key can be marked in the caption string by prefixing it with an ampersand (&). A menu item is then selected when a user presses the prefixed character. It is assigned to the instance variable :caption.

Instead of a character string, the #define constant MENU_SEPARATOR from Button.ch can be passed for *<cCaption>*. This creates a horizonal separator line to distinguish logical groups of menu items within a Popup menu.

`<bBlock>`

This is the menu activation code block which calls the subroutine after a user has selected the menu item. It is assigned to the instance variable :data.

`<oPopup>`

Instead of a code block, a Popup() menu object can be specified as second parameter. In this case, a sub-menu is opened when the menu item is selected. This allows for programming menu systems with nested sub-menus.

`<nShortCut>`

Optionally, a numeric Inkey() code can be passed that serves as shortcut key for evaluating *<bBlock>*. It is assigned to the instance variable :shortCut.

`<cMessage>`

A character string holding a status message associated with a menu item can optionally be specified. It is displayed in the message row when a TopBarMenu() is activated. *<cMessage>* is assigned to the instance variable :message.

`<nID>`

An optional numeric value can be passed which is stored in the instance variable :ID. If this parameter is used, it should be a unique value for each MenuItem object created.

Return

Function MenuItem() creates a new MenuItem object, and method :new() initializes the object.

Description

The MenuItem() class is a utility class whose objects are only used in conjunction with TopBarMenu() and/or Popup() objects. A MenuItem() object holds in its instance variables all information required to display a single menu item within a top bar menu or a pop-up menu, and to branch program control after a user has selected a menu item.

Note: refer to the example of the TopBarMenu() class for using MenuItem() objects.

Instance variables

:caption

Caption string of the menu item.

Data type: C
Default: ""

Description

The instance variable :caption holds the character string that is actually displayed on screen within a TopBarMenu() or a Popup() menu. The caption may also define the accelerator key for a menu item by prefixing a single character in the caption string with an ampersand (&). If present, the corresponding character is displayed in a different color than the rest of the caption.

When a MenuItem object is contained in a TopBarMenu() object, a user can select a menu item by pressing the accelerator key together with the Alt key. If the MenuItem object is maintained by a Popup() menu, the menu item is selected when a user presses only the accelerator key.

Note: if :caption contains the #define constant MENU_SEPARATOR from Button.ch, the menu item is displayed as a horizontal separator line and becomes not accessible.

:cargo

Instance variable for user-defined purposes.

Data type: ANY
Default: NIL

Description

This instance variable is not used by a MenuItem object. It is available for user-defined purposes when an arbitrary value should be attached to a MenuItem object.

:checked

Logical value indicating a check mark.

Data type: L
Default: .F.

Description

When .T. (true) is assigned to :checked, a check mark is displayed to the left of the caption string within a Popup() menu. Otherwise, no check mark is displayed.

The character used for the check mark is the first character of the string assigned to the :style instance variable.

:data

Menu selection code block or Popu menu object.

Data type: B,O
Default: NIL

Description

The instance variable :data contains either a code block or a Popup() menu object, unless a menu item's caption is the #define constant MENU_SEPARATOR.

If :data is a code block, it is eveluated when a menu item is selected. The code block receives as parameter the MenuItem object containing the code block.

If :data contains a Popup() menu object, this pop-up menu is opened upon menu selection.

:enabled

Logical value indicating if a menu item can be selected.

Data type: L
Default: .T.

Description

When .T. (true) is assigned to :enabled, the menu item is accessible when a menu is activated. Assigning .F. (false) makes the menu item unaccessible for the user.

:id

Numeric identifier for a menu item.

Data type: N
Default: NIL

Description

The instance variable :ID is assigned the fifth parameter passed to the :new() method. It can be used to uniquely identify a menu item when a user makes a menu selection. Refer to the example of the TopBarMenu() class to see how :ID can be used for branching program cotrol within a SWITCH statement.

:message

Character string describing a menu item.

Data type: C
Default: NIL

Description

The instance variable :message contains the fourth parameter passed to the :new() method. It is a character string displayed in the message row of a TopBarMenu() object when the menu is activated.

:shortCut

Numeric shortcut key.

Data type: N
Default: NIL

Description

The instance variable :shortCut contains the third parameter passed to the :new(). It specifies the Inkey() code that evaluates the :data code block when a MenuItem object is contained in a Popup() menu object, and this menu is not open.

If a MenuItem object is contained in a TopBarMenu() object, the :shortCut key has no effect.

:style

Character string for check mark and sub-menus

Data type: ANY
Default: HB_TMENUITEM_STYLE

Description

The instance variable :style contains a string of two characters. The first character is displayed to the left of the caption string, when :checked contains .T. (true). The second character is displayed to the right of the caption string, when :isPopup() returns .T. (true).

The default characters are given by the #define constant HB_TMENUITEM_STYLE of Button.ch.

Methods

:isPopUp()

Checks if the menu item opens a sub-menu.

Syntax

```
:isPopUp() --> lIsPopup
```

Description

Method :isPopup() returns .T. (true) when a Popup() menu object is stored in :data, otherwise .F. (false) is returned.

Info

See also: @...GET, MenuModal(), Popup(), READ, SetColor(), TopBarMenu()
Category: Get system, Object functions
Header: Box.ch, Button.ch
Source: rtl\tmenuitm.prg
LIB: xhb.lib
DLL: xhbdll.dll

Popup()

Creates a new Popup menu object.

Syntax

```
PopUp():new( [<nTop>]    , ;
             [<nLeft>]   , ;
             [<nBottom>], ;
             [<nRight>]    ) --> oPopUpMenu
```

Arguments

`<nTop> and <nLeft>`

Numeric values indicating the screen coordinates for the upper left corner of the Popup menu. The default value for *<nTop>* and *<nLeft>* is -1. These values are assigned to the instance variables :top and :left.

`<nBottom> and <nRight>`

Numeric values indicating the screen coordinates for the lower right corner of the Popup menu. The default value for *<nBottom>* and *<nRight>* is -1. These values are assigned to the instance variables :bottom and :right.

Return

Function Popup() creates a new Popup menu object, and method :new() initializes the object.

Description

The Popup class provides objects maintaining pop-up or pull-down menus. After creation of a Popup object, one or more MenuItem() objects must be added with method :addItem(). They contain information on the menu items to display in a pop-up menu. A Popup menu object is made visible with its :open() method, which maintains a rectangular area on the screen where the menu items are displayed. The :close() method hides the menu.

Popup menu objects are designed to be integrated into a TopBarMenu() object for creating a complete menu system consisting of a horizontal top bar menu that includes a series of (nested) pull-down menus. If a Popup menu object is added to TopBarMenu object, the screen coordinates for the Popup menu are calculated automatically for proper display. In this case, no screen coordinates are required for method :new().

Note: refer to the example of the TopBarMenu() class for using Popup menu objects.

Instance variables

:border

Character string for thesurrounding border.

Data type: C
Default: B_SINGLE

Description

The instance variable :border holds a character string used to display a border around the Popup menu. It is the same string used for function DispBox(). The default is a single line border. To change the default, #define constants listed in Box.ch can be used. They begin with the prefix B_*.

:bottom

Numeric bottom screen row for display.

Data type: N
Default: 0

Description

The instance variable :bottom contains a numeric value. It is the bottom screen row of the Popup menu.

:cargo

Instance variable for user-defined purposes.

Data type: ANY
Default: NIL

Description

This instance variable is not used by a Popup menu object. It is available for user-defined purposes when an arbitrary value should be attached to a Popup menu object.

:colorSpec

Color string for the Popup menu display.

Data type: C
Default: "N/W,W/N,W+/W,W+/N,N+/W,W/N"

Description

The instance variable :colorSpec holds a color string with six color values. It is initialized with the string "N/W,W/N,W+/W,W+/N,N+/W,W/N". The individual color values are used for the following purposes:

Colors for Popup menus

Color value	Used for display of
1	unselected menu items
2	selected menu item
3	accelerator key of unselected menu items
4	accelerator key of selected menu item
5	disabled menu items
6	menu border

:current

Currently selected menu item.

Data type: N
Default: 0

Description

The instance variable :current contains a numeric value indicating the ordinal position of the currently selected menu item. Items are numbered from 1 to :itemCount. If no menu item is selected, :current contains zero.

:itemCount

Number of menu items.

Data type: N
Default: 0

Description

The instance variable :itemCount contains a numeric value indicating the number of menu items, or MenuItem() objects, maintained by a Popup menu object.

:left

Numeric left screen column for display.

Data type: N
Default: -1

Description

The instance variable :left contains a numeric value. It is the screen coordinate of the left column of the Popup menu.

:right

Numeric right screen column for display.

Data type: N
Default: 0

Description

The instance variable :right contains a numeric value. It is the screen coordinate of the right column of the Popup menu.

:top

Numeric top screen row for display.

Data type: N
Default: -1

Description

The instance variable :bottom contains a numeric value. It is the top screen row of the Popup menu.

:width

Numeric width for display.

Data type: N
Default: 0

Description

The instance variable :width contains a numeric value. It is the total width of the Popup menu.

Menu item methods

:addItem()

Adds a MenuItem object to a Popup menu object.

Syntax

```
:addItem( <oMenuItem> ) --> self
```

Arguments

<oMenuItem>

This is the MenuItem() object to add to the Popu object.

Description

Method :addItem() accepts a MenuItem() object and adds it to the internal list of menu items. The return value is the *self* object. The instance variable :itemCount reflects the new number of MenuItem() objects maintained by a Popup object.

:delItem()

Deletes a MenuItem object from a Popup object.

Syntax

```
:delItem( <nItemPos> ) --> self
```

Arguments

<nItemPos>

This is a numeric value specifying the ordinal position of the menu item to delete from the Popup object. Menu items are numbered from 1 to :itemCount.

Description

Method :delItem() can be used to remove a single menu item from a Popup object. This can be useful when the number of selectable menu items changes dynamically at runtime, depending on a condition. Alternatively, single menu items can be disabled or enabled.

:getFirst()

Retrieves the ordinal position of the first selectable menu item.

Syntax

```
:getFirst() --> nFirstMenuItemPos
```

Description

Method :getFirst() returns a numeric value indicating the ordinal position of the first, or toptmost, selectable menu item. When a menu item is disabled, it cannot be accessed by the user. When no menu item can be selected, the return value is zero.

:getItem()

Retrieves a MenuItem object from a Popup object.

Syntax

```
:getItem( <nItemPos> ) --> oMenuItem
```

Arguments

`<nItemPos>`

This is a numeric value specifying the ordinal position of the menu item to retrieve from the Popup object. Menu items are numbered from 1 to :itemCount.

Description

Method :getItem() is used to retrieve the MenuItem() object at position *<nItemPos>* from a Popup object.

:getLast()

Retrieves the ordinal position of the last selectable menu item.

Syntax

```
:getLast() --> nLastMenuItemPos
```

Description

Method :getLast() returns a numeric value indicating the ordinal position of the last selectable menu item at the bottomof aPupup menu. When a menu item is disabled, it cannot be accessed by the user. When no menu item can be selected, the return value is zero.

:getNext()

Retrieves the ordinal position of the next selectable menu item.

Syntax

```
:getNext() --> nNextMenuItemPos
```

Description

Method :getNext() returns a numeric value indicating the ordinal position of the selectable menu item below the currently selected menu item. When a menu item is disabled, it cannot be accessed by the user. If the current menu item is the last one, or when no menu item can be selected, the return value is zero.

:getPrev()

Retrieves the ordinal position of the previous selectable menu item.

Syntax

```
:getPrev() --> nPrevMenuItemPos
```

Description

Method :getPrev() returns a numeric value indicating the ordinal position of the selectable menu item above the currently selected menu item. When a menu item is disabled, it cannot be accessed by the

user. If the current menu item is the first one, or when no menu item can be selected, the return value is zero.

:insItem()

Inserts a MenuItem object into a Popup object.

Syntax

```
:insItem( <nItemPos>, <oMenuItem> ) --> self
```

Arguments

`<nItemPos>`

This is a numeric value specifying the ordinal position where to insert the menu item. The value for *<nItemPos>* must be in the range from 1 to :itemCount. Otherwise, it is ignored.

`<oMenuItem>`

This is the MenuItem() object to insert into the Popup object.

Description

Method :insItem() accepts a MenuItem() object and inserts it into the internal list of menu items at position *<nItemPos>*. The return value is the *self* object. The instance variable :itemCount reflects the new number of MenuItem() objects maintained by a Popup object. If *<nItemPos>* is outside the valid range, the number of menu items remains unchanged.

:setItem()

Replaces a MenuItem object in a Popup object.

Syntax

```
:setItem( <nItemPos>, <oMenuItem> ) --> self
```

Arguments

`<nItemPos>`

This is a numeric value specifying the ordinal position of the menu item to replace. The value for *<nItemPos>* must be in the range from 1 to :itemCount. Otherwise, it is ignored.

`<oMenuItem>`

This is the MenuItem() object that replaces an existing one in the Popup object.

Description

Method :setItem() accepts a MenuItem() object and replaces the existing MenuItem object at position *<nItemPos>*. The return value is the *self* object. If *<nItemPos>* is outside the valid range, no menu item is replaced.

Menu display and selection

:close()

Hides a visible Popup menu.

Syntax

```
:close() --> self
```

Description

Method :close() hides a visible Popup menu and displays the screen contents covered by the menu. If the menu is not visible, the method has no effect. The return value is the *self* object.

:display()

Displays a Popup menu.

Syntax

```
:display() --> self
```

Description

Method :display() draws the border of a Popup menu and then iterates through the internal list of MenuItem() objects to display their captions within the rectangular area of the menu.

:getAccel()

Determines if a key code identifies an accelerator key.

Syntax

```
:getAccel( <nKey> ) --> nMenuItemPos
```

Arguments

<nKey>

This is a numeric Inkey() code to check.

Description

Method :getAccel() is used to check if a numeric inkey code is an accelerator key that should trigger a menu action. Accelerator keys are key combinations of the Alt key an a letter of a menu item caption prefixed with an ampersand (&). If the key code *<nKey>* is an accelerator key, the method returns the ordinal position of the corresponding menu item as a numeric value. If it is not an accelerator, the return value is zero.

:getShortct()

Determines if a key code identifies a shortcut key.

Syntax

```
:getShortct( <nKey> ) --> mMenuItemPos
```

Arguments

<nKey>

This is a numeric Inkey() code to check.

Description

Method :getShortct() is used to check if a numeric inkey code is a shortcut key that should trigger a menu action. Shortcut keys are defined when MenuItem() objects are created. They are stored in the instance variable :shortCut of MenuItem objects. If the key code *<nKey>* is a shortcut key, the method returns the ordinal position of the corresponding menu item as a numeric value. If it is not a shortcut, the return value is zero.

:hitTest()

Checks if a menu item is clicked with the mouse.

Syntax

```
:hitTest( <nMouseRow>, <nMouseCol> ) --> nMenuItemPos
```

Arguments

<nMouseRow>

This is the numeric row position of the mouse cursor. It can be queried using function MRow().

<nMouseCol>

This is the numeric column position of the mouse cursor. It can be queried using function MCol().

Description

Method :hitTest() accepts the numeric row and column position of the mouse cursor and returns a numeric value > 1 indicating the ordinal position of the menu item that is underneath the mouse cursor. If the mouse hit a non-selectable or disabled menu item inside the border of a popup menu, the return value is zero.

When the border of the menu is clicked, the return value is negative and can be tested with #define constants available in Button.ch.

Constants for :hitTest() results

Constant	Value	Description
HTTOPLEFT	-1	Mouse is on the upper left corner
HTTOP	-2	Mouse is on the top border
HTTOPRIGHT	-3	Mouse is on the upper right corner
HTRIGHT	-4	Mouse is on the right border
HTBOTTOMRIGHT	-5	Mouse is on the lower right corner
HTBOTTOM	-6	Mouse is on the bottom border
HTBOTTOMLEFT	-7	Mouse is on the lower left corner
HTLEFT	-8	Mouse is on the left border

:isOpen()

Checks if a Popup menu is visible.

Syntax

```
:isOpen() --> lIsOpen
```

Description

Method :isOpen() returns .T. (true) when a Popup menu is visible, otherwise .F: (false) is returned.

:open()

Opens a Popup menu.

Syntax

```
:open() --> self
```

Description

Method :open() first saves the screen contents that will be covered by a Popup menu and then calls the :display(). If the menu is already visible, the method has no effect. The return value is the *self* object.

:select()

Changes the currently selected menu item.

Syntax

```
:select( <nItemPos> ) --> nMenuItemPos
```

Arguments

`<nItemPos>`

This is a numeric value specifying the ordinal position of the menu item to select as current. The value for *<nItemPos>* must be in the range from 1 to :itemCount. Otherwise, it is ignored.

Description

Method :select() defines the menu item at position *<nItemPos>* as currently selected item. The selected menu item is displayed highlighted when method :display() is executed.

Info

See also:	@...GET, MenuItem(), MenuModal(), TopBarMenu(), SetColor()
Category:	Get system, Object functions
Header:	Box.ch, Button.ch
Source:	rtl\tpopup.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

TBColumn()

Creates a new TBColumn object.

Syntax

```
TBColumn():new( <cHeading>, <bData> ) --> oTBColumn
```

Arguments

<cHeading>

This is a character string which is displayed in the column heading of a TBrowse() display.

<bData>

This is a code block which returns the data to be displayed within the data area of one column of a TBrowse() display.

Return

Function TBColumn() returns a new TBColumn object and method :new() initializes the object.

Description

TBColumn objects are required for creating a browser in text mode applications. They are always used in conjunction with a TBrowse() object which is a container for TBColumn objects. A TBColumn object contains all information required for displaying a single column of data in a browse view. As a consequence, TBColumn objects are useless on their own and must be added to a TBrowse object with its :addColumn() method.

Instance variables

:block

Code block to retrieve data for the column.

Data type: B
Default: NIL

Description

The instance variable :block must be assigned a code block which returns the data to be displayed in a single column of a browse view. When the data source is a database file, the return value of function FieldBlock() or FieldWBlock() can be assigned to :block.

:cargo

Instance variable for user-defined purposes.

Data type: ANY
Default: NIL

Description

This instance variable is not used by a TBColumn object. It is available for user-defined purposes when an arbitrary value should be attached to a TBColumn object.

:colorBlock

Code block selecting the color of data cells depending on their values

Data type: B
Default: NIL

Description

The instance variable :colorBlock can be assigned a code block which selects the color for a single data cell within a column of data. If it is present, :colorBlock is passed the return value of :block, which is the data to be displayed. :colorBlock must return an array with two numeric elements. Each element points to a color value of the color string stored in the instance variable :colorSpec of the TBrowse object containing the TBColumn object. The first element selects the color for normal display of the data cell, and the second points to the color to use for highlighting the data cell.

:colSep

Characters used as vertical column separator.

Data type: C
Default: NIL

Description

The instance variable :colSep holds a character string of one of more characters which are displayed as a separator between data columns in the browse view. The initial value for :colSep is NIL, which causes the column separator defined for the TBrowse object being displayed. If a TBColumn object has its own :colSep assigned, it overrides the column separator specified for the TBrowse object.

:defColor

Array holding numeric indexes into the color table of TBrowse

Data type: A
Default: {1,2,1,1}

Description

The instance variable :defColor contains an array of four numeric values. They are used as indexes into the color string stored in the instance variable :colorSpec of the TBrowse object containing a TBColumn object. The first element selects the color for normal data display, the second highlights a data cell (the browse cursor), the third selects the color for the column heading and the fourth is used for the column footing. Note that :colorBlock overrides :defColor.

:footing

Character string displayed as column footing.

Data type: C
Default: ""

Description

A character string can be assigned to :footing which is displayed as column footing. The semicolon serves as line-break character when a column footing should be displayed in multiple lines. To change a column footing after a browse view is displayed, method :configure() of the TBrowse object containing the column must be called.

:footSep

Characters used as horizontal separator for column footings.

Data type: C
Default: ""

Description

The instance variable :footSep holds a character string of one of more characters which are displayed as a separator between the data area and the column footing. The initial value for :footSep is an empty string ("") which suppresses the display of a footing separator. If :footSep contains more than one character, the last character is displayed across the width of a column, while the first character is displayed at the position of :colSep.

:heading

Character string displayed as column heading.

Data type: C
Default: " "

Description

A character string can be assigned to :heading which is displayed as column heading. The semicolon serves as line-break character when a column heading should be displayed in multiple lines. To change a column heading after a browse view is displayed, method :configure() of the TBrowse object containing the column must be called.

:headSep

Characters used as horizontal separator for column headings.

Data type: C
Default: NIL

Description

The instance variable :headSep holds a character string of one of more characters which are displayed as a separator between the data area and the column heading. The initial value for :headSep is an empty string ("") which suppresses the display of a heading separator. If :headSep contains more than one character, the last character is displayed across the width of a column, while the first character is displayed at the position of :colSep.

:picture

Character string holding a PICTURE format.

Data type: C
Default: NIL

Description

The instance variable :picture holds a PICTURE formatting string used for displaying values in a column. Refer to function Transform() for a comprehensive description of PICTURE formatting strings.

:postBlock

Code block for data validation.

Data type: B
Default: NIL

Description

Optionally, a code block can be assigned to :postValidate that is used to validate edited data. This code block is not used by a TBColumn object but provides a convenient way for implementing editable browsers. :postBlock can be used by a Get() object when the current cell in a column should be edited.

:preBlock

Code block for edit validation.

Data type: B
Default: NIL

Description

Optionally, a code block can be assigned to :preValidate that is used to test if editing is allowed. This code block is not used by a TBColumn object but provides a convenient way for implementing editable browsers. :preBlock can be used by a Get() object when the current cell in a column should be edited.

:width

Numeric width of column.

Data type: N
Default: 0

Description

The instance variable :width contains a numeric value representing the width of a column on the screen. If no value is assigned, :width it is initialized upon initial display of the column. The width is calculated as the maximum length of :heading, :footing and data area of a column when :block is evaluated for the first time.

Assigning 0 to :width makes the column unvisible although it is still present in a TBrowse object.

Methods

:setStyle()

Retrieves or changes a column style.

Syntax

```
:setStyle( <nStyle>, [<lNewOnOff>] ) --> lOldOnOff
```

Arguments

`<nStyle>`

This is a numeric value indicating the style to query or change. #define constants are listed in the file TBrowse.ch which can be used for *<nStyle>*.

Constants for TBColumn styles

Constant	Value	Description
TBC_READWRITE	1	Is the user allowed to edit cells in the column?
TBC_MOVE	2	Is the user allowed to move this column in the browser?
TBC_SIZE	3	Is the user allowed to resize this column in the browser?
TBC_CUSTOM	4	Minimum value for use-defined custom styles.

`<lNewOnOff>`

This is an optional logical value. When .T. (true) is passed, the style *<nStyle>* is switched on, .F. (false) switches it off.

Description

A TBColumn object maintains a set of styles that can be switched on and off. The styles can be used as a convenient way of implementing specific behaviour for a TBColumn object. A TBColumn object itself does not use this information.

Info

See also: TBrowse()
Category: Object functions
Header: tbrowse.ch
Source: rtl\tbcolumn.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// Refer to function TBrowse() for an example of TBColumn usage.
```

TBrowse()

Creates a new TBrowse object.

Syntax

```
TBrowse():new( [<nTop>]    , ;
               [<nLeft>]   , ;
               [<nBottom>], ;
               [<nRight>]    ) --> oTBrowse
```

Arguments

<nTop>

This is the numeric screen coordinate for the top row of the browse display. It defaults to zero and is assigned to the instance variable :nTop.

<nLeft>

This is the numeric screen coordinate for the left column of the browse display. It defaults to zero and is assigned to the instance variable :nLeft.

<nBottom>

This is the numeric screen coordinate for the bottom row of the browse display. It defaults to MaxRow() and is assigned to the instance variable :nBottom.

<nRight>

This is the numeric screen coordinate for the right column of the browse display. It defaults to MaxCol() and is assigned to the instance variable :nRight.

Return

Function TBrowse() returns a new TBrowse object and method :new() initializes the object.

Description

TBrowse objects are used in text mode applications to display tabular data organized in rows and columns in a rectangular area on the screen. Common data sources for TBrowse objects are database files or arrays. In order to display such data, a TBrowse object must receive objects of the TBColumn() class which contain all information required for displaying a single column in a browse view. In addition, a TBrowse object must be assigned code blocks required for navigating the row pointer of the data source as a response to user input. The example below demonstrates all steps required to create a fully functional browse view with TBrowse and TBColumn objects.

Instance variables

:autoLite

Logical value for automatic highlighting of the browse cursor.

Data type: L
Default: .T.

Description

When :autoLite is set to .T. (true), which is the default, a TBrowse object automatically highlights the browse cursor when the browse display is stable. Setting :autoLite to .F. (false) requires the browse cursor be highlighted programmatically using method :hilite().

:border

Character string defining a border to display around the browse display.

Data type: C
Default: ""

Description

The instance variable :border can be assigned a character string of eight characters which are used to display a border around the browse area. #define constants specifying standard borders are listed in the file BOX.CH. They can be used as values for :border. The border is drawn with function DispBox().

:cargo

Instance variable for user-defined purposes.

Data type: ANY
Default: NIL

Description

This instance variable is not used by a TBrowse object. It is available for user-defined purposes when an arbitrary value should be attached to a TBrowse object.

:colCount

Number of columns maintained by the TBrowse object

Data type: N (READONLY)
Default: 0

Description

The instance variable :colCount contains a numeric value representing the number of columns, or TBColumn objects, maintained by a TBrowse object.

:colorSpec

Color string for the TBrowse display.

Data type: C
Default: SetColor()

Description

The instance variable :colorSpec holds a color string with two or more color values. It is initialized with the return value of SetColor(). By default, a TBrowse object uses the first color value for displaying the column headings and data rows, and the second color for highlighting the browse cursor. This default behavior can be overriden for individual columns by using :defColor or :colorBlock of a single TBColumn() object.

:colPos

Numeric column position of the browse cursor.

Data type: N
Default: 0

Description

The instance variable :colPos contains a numeric value indicating the current column position of the browse cursor. Columns are numbered beginning with one, which is the leftmost column. The column position of the browse cursor can be changed by assigning a value between 1 and :colCount to :colPos. This causes the TBrowse object to perform a complete stabilization cycle.

:colSep

Characters used as vertical column separator.

Data type: C
Default: " "

Description

The instance variable :colSep holds a character string of one of more characters which are displayed as a separator between data columns during a stabilization cycle. The initial value for :colSep is a blank space. If a TBColumn object has its own :colSep assigned, it overrides the column separator specified for the TBrowse object.

:footSep

Characters used as horizontal separator for column footings.

Data type: C
Default: ""

Description

The instance variable :footSep holds a character string of one of more characters which are displayed as a separator between the data area and the column footings. The initial value for :footSep is an empty string ("") which suppresses the display of a footing separator. If :footSep contains more than one character, the last character is displayed across the width of a column, while the first character is display at the position of :colSep. If a TBColumn object has its own :footSep assigned, it overrides the footing separator specified for the TBrowse object.

:freeze

Number of columns to freeze in the browse display.

Data type: N
Default: 0

Description

The instance variable :freeze can be assigned a numeric value which indicates the number of columns to remain fixed and visible on the left side of the browse display. Frozen columns are not moved by horizontal scrolling.

:goBottomBlock

Code block that navigates the data source to the last row.

Data type: B
Default: NIL

Description

The instance variable :goBottomBlock must be assigned a code block which navigates the row pointer of the data source to the last row. This code block is evaluated in method :goBottom(). When the data source is a database file, a typical code block for :goBottomBlock is {|| DbGoBottom() }.

:goTopBlock

Code block that navigates the data source to the first row.

Data type: B
Default: NIL

Description

The instance variable :goTopBlock must be assigned a code block which navigates the row pointer of the data source to the first row. This code block is evaluated in method :goTop(). When the data source is a database file, a typical code block for :goTopBlock is {|| DbGoTop() }.

:headSep

Character used as horizontal separator for column headings.

Data type: C
Default: ""

Description

The instance variable :headSep holds a character string of one of more characters which are displayed as a separator between the data area and the column headings. The initial value for :headSep is an empty string ("") which suppresses the display of a heading separator. If :headSep contains more than one character, the last character is displayed across the width of a column, while the first character is display at the position of :colSep. If a TBColumn object has its own :headSep assigned, it overrides the heading separator specified for the TBrowse object.

:hitBottom

Indicates that the browser reached the end of the data source.

Data type: L
Default: .F.

Description

The instance variable :hitBottom contains .T. (true) when a user attempts to navigate the row pointer of the data source past the last row. In all other cases it contains .F. (false). :hitBottom is set during a stabilization cycle.

:hitTop

Indicates that the browser reached the beginning of the data source.

Data type: L
Default: .F.

Description

The instance variable :hitTop contains .T. (true) when a user attempts to navigate the row pointer of the data source before the first row. In all other cases it contains .F. (false). :hitTop is set during a stabilization cycle.

:leftVisible

Numeric position of the leftmost unfrozen column in the browse display.

Data type: N
Default: 1

Description

The instance variable :leftVisible contains a numeric value indicating the ordinal position of the leftmost unfrozen column. Unfrozen columns can be scrolled horizontally. If all columns are frozen, :leftVisible contains zero.

:mColPos

Numeric column position of the mouse cursor within the browse display.

Data type: N
Default: 0

Description

The instance variable :mColPos contains a numeric value indicating the ordinal position of the column where the mouse cursor is currently located. Columns are numbered beginning with one, which is the leftmost column.

:message

Character string to display in the Get system's status line

Data type: C
Default: ""

Description

The instance variable :message can be assigned a character string when a TBrowse object is used with the Get system. The message string is displayed in the status line of the Get system when the READ command or function ReadModal() is executed.

:mRowPos

Numeric row position of the mouse cursor within the browse display.

Data type: N
Default: 0

Description

The instance variable :mRowPos contains a numeric value indicating the ordinal position of the data row where the mouse cursor is currently located. Data rows are numbered from 1 to :rowCount.

:nBottom

Numeric bottom row coordinate of the browse display.

Data type: N
Default: MaxRow()

Description

The instance variable :nBottom contains a numeric value. It is the screen coordinate of the bottom row of the browse window. Assigning a new value to :nBottom invalidates the entire browse display. To stabilize the TBrowse object after changing :nBottom requires a complete stabilization cycle.

:nLeft

Numeric left column coordinate of the browse display

Data type: N
Default: 0

Description

The instance variable :nLeft contains a numeric value. It is the screen coordinate of the left column of the browse window. Assigning a new value to :nLeft invalidates the entire browse display. To stabilize the TBrowse object after changing :nLeft requires a complete stabilization cycle.

:nRight

Numeric right column coordinate of the browse display

Data type: N
Default: MaxCol()

Description

The instance variable :nRight contains a numeric value. It is the screen coordinate of the right column of the browse window. Assigning a new value to :nRight invalidates the entire browse display. To stabilize the TBrowse object after changing :nRight requires a complete stabilization cycle.

:nTop

Numeric top row coordinate of the browse display

Data type: N
Default: 0

Description

The instance variable :nTop contains a numeric value. It is the screen coordinate of the top row of the browse window. Assigning a new value to :nTop invalidates the entire browse display. To stabilize the TBrowse object after changing :nTop requires a complete stabilization cycle.

:rightVisible

Numeric position of the rightmost unfrozen column in the browse display.

Data type: N
Default: 1

Description

The instance variable :rightVisible contains a numeric value indicating the ordinal position of the rightmost unfrozen column that is visible. Unfrozen columns can be scrolled horizontally. If all columns are frozen, :rightVisible contains zero.

:rowCount

Number of data rows in the browse display.

Data type: N
Default: 0

Description

The instance variable :rowCount contains a numeric value representing the number of rows in the data area displayed by a TBrowse object.

:rowPos

Numeric row position of the browse cursor.

Data type: N
Default: 1

Description

The instance variable :rowPos contains a numeric value indicating the current row position of the browse cursor within the browse display. The row position of the browse cursor can be changed by assigning a value between 1 and :rowCount to :rowPos. This causes the TBrowse object to perform a complete stabilization cycle.

:skipBlock

Code block that navigates the row pointer of the data source.

Data type: B
Default: NIL

Description

The instance variable :skipBlock must be assigned a code block which navigates the row pointer of the data source. This code block is evaluated in method :stabilize() or :forceStable(). The code block must accept one numeric parameter which is the number of rows the row pointer must be changed during a single stabilization cycle. The return value of :skipBlock must be numeric as well. It tells the Tbrowse object how far the row pointer of the data source was actually skipped.

When the data source is a database file, a typical code block for :skipBlock is:

```
oTBrowse:skipBlock := {|nSkip| DbSkipper(nSkip) }
```

This code block calls the utility function DbSkipper() which navigates the browse cursor upon request of a TBrowse object.

:stable

Indicates that the browse display is stable.

Data type: L
Default: .F.

Description

The instance variable :stable contains .T. (true) when a TBrowse object has completed a stabilization cycle. Method :stabilize() must be called as long as :stable contains .F. (false) in order to display data for all rows and columns of the browser.

Methods for columns

:addColumn()

Adds a TBColumn object to a TBrowse object.

Syntax

```
:addColumn( <oTBColumn> ) --> oTBColumn
```

Arguments

`<oTBColumn>`

This is the TBColumn object to add to the TBrowse object.

Description

Method :addColumn() accepts a TBColumn() object and adds it to the internal list of column objects. The return value is the added object. The instance variable :colCount reflects the new number of column objects maintained by a TBrowse object. When the TBColumn object is added, the entire TBrowse display is invalidated and all data from the data source is re-read during the next stabilization cycle.

:colWidth()

Returns the width of a column in the browse display.

Syntax

```
:colWidth( <nColPos> ) --> nColumnWidth
```

Arguments

```
<nColPos>
```

This is a numeric value specifying the ordinal position of the column whose width should be returned. Columns are numbered from 1 to :colCount.

Description

Method :colWidth() is used to determine the width of a single column in the TBrowse display. If no column exists at position *<nColPos>*, the return value is zero.

:delColumn()

Removes a TBColumn object from a TBrowse object.

Syntax

```
:delColumn( <nColPos> ) --> oTBColumn
```

Arguments

```
<nColPos>
```

This is a numeric value specifying the ordinal position of the column to remove from the browser. Columns are numbered from 1 to :colCount.

Description

Method :delColumn() is used to remove a single column from a TBrowse object. The method returns the removed object so that it can be preserved and inserted at another position. When the TBColumn object is removed, the entire TBrowse display is invalidated and all data from the data source is re-read during the next stabilization cycle.

:getColumn()

Retrieves a TBColumn object from a TBrowse object.

Syntax

```
:getColumn( <nColPos> ) --> oTBColumn
```

Arguments

```
<nColPos>
```

This is a numeric value specifying the ordinal position of the column to retrieve from the browser. Columns are numbered from 1 to :colCount.

Description

Method :getColumn() retrieves the TBColumn object at position *<nColPos>* from a TBrowse object. If the TBColumn object is changed, the TBrowse object must be notified about such changes by calling one of the methods :configure(), :invalidate() or :refreshAll(). Changes made to the TBColumn object become then visible during the next stabilization cycle.

:insColumn()

Inserts a TBColumn object into a TBrowse object.

Syntax

```
:insColumn( <nColPos>, <oTBColumn> ) --> oTBColumn
```

Arguments

`<nColPos>`

This is a numeric value specifying the ordinal position where to insert a TBColumn object in the browser.

`<oTBColumn>`

This is the TBColumn object to insert into the TBrowse object.

Description

Method :insColumn() accepts a TBColumn() object and inserts it into the internal list of column objects. The return value is the inserted object. The instance variable :colCount reflects the new number of column objects maintained by a TBrowse object. When the TBColumn object is inserted, the entire TBrowse display is invalidated and all data from the data source is re-read during the next stabilization cycle.

:setColumn()

Replaces a TBColumn object within a TBrowse object

Syntax

```
:setColumn( <nColPos>, <oTBColumnNew> ) --> oTBColumnOld
```

Arguments

`<nColPos>`

This is a numeric value specifying the ordinal position of the TBColumn object to replace in the browser.

`<oTBColumnNew>`

This is the TBColumn object that replaces an existing one in the TBrowse object.

Description

Method :setColumn() accepts a TBColumn() object and replaces the existing column object at position *<nColPos>*. The return value is the replaced TBColumn object. When the TBColumn object is replaced, the entire TBrowse display is invalidated and all data from the data source is re-read during the next stabilization cycle.

Display methods

:colorRect()

Changes the color of a group of cells in the browse display.

Syntax

```
:colorRect( <aTLBR>, <aColorIndex> ) --> self
```

Arguments

```
<aTLBR> := { <nTop>, <nLeft>, <nBottom>, <nRight> }
```

This is an array holding four numeric values. They specify the ordinal positions of the data rows and columns of a rectangular area in the browse display whose color should be changed.

```
<aColorIndex> := { <nNormal>, <nHighlight> }
```

This is an array holding two numeric values. They specify the color values of the color string held in :colorSpec to use for display. The first element points to the color value for normal display and the second element indicates the color for highlighting a cell, or the browse cursor, respectively.

Description

Method :colorRect() can be used to change the color of a rectangular area within the data rows of the browse display. This area is defined using ordinal positions of data rows and columns, not screen coordinates. The top and bottom borders of the rectangle must be specified as ordinal row positions in the range from 1 to :rowCount, and the left and right borders are ordinal column positions ranging from 1 to :colCount.

The two array elements of *<aColorIndex>* point to color values of the color string held in :colorSpec. The first element selects the color for normal display while the second determines the color for the browse cursor when it is moved into the rectangular area *<aTLBR>*.

When the colored area is scrolled vertically outside the browse display, the cells in this area do not retain the changed color.

:configure()

Reconfigures the internal data of a TBrowse object.

Syntax

```
:configure() --> self
```

Description

Method :configure() instructs a TBrowse object to recalculate the entire browse display and inspect all TBColumn objects stored in the internal list of columns. All data is re-read from the data source and the browse display is refreshed during the next stabilization cycle. :configure() needs to be called when certain instance variables of TBColumn objects are changed while the browser is visible. For example, when oTBColumn:width or oTBColumn:heading is changed, :configure() must be called to make these changes visible.

:deHilite()

Dehighlights the browse cursor.

Syntax

```
:deHilite() --> self
```

Description

Method :deHilite() changes the color of the browse cursor to the normal color. It is used in conjunction with method :hilite() to control the color of the browse cursor programmatically when instance variable :autoLite is set to .F. (false).

:forceStable()

Performs a complete stabilization of the browse display.

Syntax

```
:forceStable() --> self
```

Description

Method :forceStable() stabilizes the entire browse display. No user input is possible until the method returns.

:hilite()

Highlights the browse cursor.

Syntax

```
:hilite() --> self
```

Description

Method :hilite() changes the color of the browse cursor to the hilighted color. It is used in conjunction with method :deHilite() to control the color of the browse cursor programmatically when instance variable :autoLite is set to .F. (false).

:invalidate()

Invalidates the entire browse display.

Syntax

```
:invalidate() --> self
```

Description

Method :invalidate() instructs a TBrowse object to redraw the entire browse display during the next stabilization cycle. This includes column headings and footings and all data rows. Note, however, that the data is not re-read from the data source. Call :refreshAll() to update the data rows from the data source.

:refreshAll()

Invalidates all data rows of the browse display.

Syntax

```
:refreshAll() --> self
```

Description

Method :refreshAll() causes a TBrowse object to update the display of all data rows by re-reading :rowCount rows of the data source in the next stabilization cycle.

:refreshCurrent()

Invalidates the current data row of the browse display.

Syntax

```
:refreshCurrent() --> self
```

Description

Method :refreshCurrent() causes a TBrowse object to update the display of the current data row by re-reading the current row of the data source in the next stabilization cycle.

:setStyle()

Retrieves or changes a browse style.

Syntax

```
:setStyle( <nStyle>, [<lNewOnOff>] ) --> lOldOnOff
```

Arguments

`<nStyle>`

This is a numeric value indicating the style to query or change. #define constants are listed in the file TBrowse.ch which can be used for <*nStyle*>.

Constants for TBrowse styles

Constant	Value	Description
TBR_APPEND	1	Is the user allowed to add data to the data source?
TBR_APPENDING	2	Is the user adding data to the data source?
TBR_MODIFY	3	Is the user allowed to edit cells in the browser?
TBR_MOVE	4	Is the user allowed to move columns in the browser?
TBR_SIZE	5	Is the user allowed to resize columns in the browser?
TBR_CUSTOM	6	Minimum value for use-defined custom styles.

`<lNewOnOff>`

This is an optional logical value. When .T. (true) is passed, the style <*nStyle*> is switched on, .F. (false) switches it off.

Description

A TBrowse object maintains a set of styles that can be switched on and off. The styles can be used as a convenient way of implementing specific behaviour for a TBrowse object. A TBrowse object itself does not use this information.

:stabilize()

Performs one incremental stabilization cycle.

Syntax

```
:stabilize() --> lIsStable
```

Description

Method :stabilize() performs one incremental stabilization cycle and returns .T. (true) when the browse display is stable. In a stable display, all heading, footing and data rows are displayed, the browse cursor corresponds with the current row in the data source and is highlighted, and no keystrokes are pending

for processing. As long as the method returns .F. (false), :stabilize() must be called repeatedly to complete the stabilization cycle.

It is possible to interrupt the incremental stabilization cycle and call cursor navigation methods as a response to user input. This can speed up the entire browse display when a user holds a navigation key pressed for a longer period of time, for example.

Cursor navigation

:down()

Navigates the cursor to the next row.

Syntax

```
:down() --> self
```

Description

Method :down() instructs the TBrowse object to move the browse cursor to the next data row. If the browse cursor is already located in the last data row, which is at position :rowCount, the data area of the browse display is scrolled up one line and a new data row is read from the data source. If the current row of the data source is the last row already, :down() does not move the browse cursor but sets the instance variable :hitBottom to .T. (true).

:end()

Navigates the cursor to the rightmost visible data column.

Syntax

```
:end() --> self
```

Description

Method :end() moves the browse cursor to the rightmost visible data column. It does not scroll the browse display.

:goBottom()

Navigates the cursor to the last row in the data source.

Syntax

```
:goBottom() --> self
```

Description

Method :goBottom() evaluates the code block stored in the instance variable :goBottomBlock. The code block receives no argument and must navigate the row pointer of the data source to the last row. A TBrowse object then rebuilds the data area of the browse display in the next stabilization cycle and positions the browse cursor on the last visible data row.

:goTop()

Navigates the cursor to the first row in the data source.

Syntax

```
:goTop() --> self
```

Description

Method :goTop() evaluates the code block stored in the instance variable :goTopBlock. The code block receives no argument and must navigate the row pointer of the data source to the first row. A TBrowse object then rebuilds the data area of the browse display in the next stabilization cycle and positions the browse cursor on the first data row.

:home()

Navigates the cursor to the leftmost visible data column.

Syntax

```
:home() --> self
```

Description

Method :home() moves the browse cursor to the leftmost visible data column. It does not scroll the browse display.

:left()

Navigates the cursor left one column.

Syntax

```
:left() --> self
```

Description

Method :left() moves the browse cursor one column to the left. If the next left column is currently not visible, the browse display is scrolled to the right.

:pageDown()

Navigates the cursor to the next page in the data source.

Syntax

```
:pageDown() --> self
```

Description

Method :pageDown() instructs the TBrowse object to display the next :rowCount rows of the data source. This invalidates the entire browse display and :rowCount rows are re-read from the data source in the next stabilization cycle. If the last row of the data source is already visible, the browse cursor is positioned on the last visible data row. When :pageDown() is called and the current row of the data source is the last row already, :pageDown() does not move the browse cursor but sets the instance variable :hitBottom to .T. (true).

:pageUp()

Navigates the cursor to the previous page in the data source.

Syntax

```
:pageUp() --> self
```

Description

Method :pageUp() instructs the TBrowse object to display the previous :rowCount rows of the data source. This invalidates the entire browse display and :rowCount rows are re-read from the data source in the next stabilization cycle. If the first row of the data source is already visible, the browse cursor is positioned on the first visible data row. When :pageUp() is called and the current row of the data source is the first row already, :pageUp() does not move the browse cursor but sets the instance variable :hitTop to .T. (true).

:panEnd()

Navigates the cursor to the rightmost data column.

Syntax

```
:panEnd() --> self
```

Description

Method :panEnd() moves the browse cursor to the last data column. The browse display is scrolled to the left until the rightmost column becomes visible.

:panHome()

Navigates the cursor to the leftmost visible data column.

Syntax

```
:panHome() --> self
```

Description

Method :panHome() moves the browse cursor to the first data column. The browse display is scrolled to the right until the leftmost column becomes visible.

:panLeft()

Pans the browse display left without changing the cursor position.

Syntax

```
:panLeft() --> self
```

Description

Method :panLeft() scrolls the browse display one column to the right so that the next unvisible column becomes visible on the left. The browse cursor remains in its current column, if possible.

:panRight()

Pans the browse display right without changing the cursor position.

Syntax

```
:panRight() --> self
```

Description

Method :panRight() scrolls the browse display one column to the left so that the next unvisible column becomes visible on the right. The browse cursor remains in its current column, if possible.

:right()

Navigates the cursor right one column.

Syntax

```
:right() --> self
```

Description

Method :right() moves the browse cursor one column to the right. If the next right column is currently not visible, the browse display is scrolled to the left.

:up()

Navigates the cursor to the previous row.

Syntax

```
:up() --> self
```

Description

Method :up() instructs the TBrowse object to move the browse cursor to the previous data row. If the browse cursor is already located in the first data row, the data area of the browse display is scrolled down one line and a new data row is read from the data source. If the current row of the data source is the first row already, :up() does not move the browse cursor but sets the instance variable :hitTop to .T. (true).

Event handling

:applyKey()

Evaluates a code block associated with a navigation key.

Syntax

```
:applyKey( <nInkey> ) --> nReturnCode
```

Arguments

<nInkey>

This is a numeric Inkey() code to be processed by a TBrowse object.

Description

Method :applyKey() instructs a TBrowse object to process user input obtained from function Inkey(). The return value is a numeric code that indicates whether or not the stabilization loop must be terminated. #define constants are listed in the file TBrowse.ch that can be used to test possible return values of :applyKey().

Return values of :applyKey()

Constant	Value	Description
TBR_EXIT	-1	User request for the browse to lose input focus
TBR_CONTINUE	0	Code block associated with <*nInkey*> was evaluated
TBR_EXCEPTION	1	<*nInkey*> is unknown, key was not processed

A TBrowse object maintains a :setKey() dictionary of Inkey() codes and associated code blocks that perform default navigation. The default key processing is as follows:

Default key processing

Inkey code	Method or function	Return code
K_DOWN	:down()	TBR_CONTINUE
K_UP	:up()	TBR_CONTINUE
K_RIGHT	:right()	TBR_CONTINUE
K_LEFT	:left()	TBR_CONTINUE
K_CTRL_LEFT	:panLeft()	TBR_CONTINUE
K_CTRL_RIGHT	:panRight()	TBR_CONTINUE
K_END	:end()	TBR_CONTINUE
K_HOME	:home()	TBR_CONTINUE
K_CTRL_END	:panEnd()	TBR_CONTINUE
K_CTRL_HOME	:panHome()	TBR_CONTINUE
K_PGDN	:pageDown()	TBR_CONTINUE
K_PGUP	:pageUp()	TBR_CONTINUE
K_CTRL_PGDN	:goBottom()	TBR_CONTINUE
K_CTRL_PGUP	:goTop()	TBR_CONTINUE
K_ESC	None	TBR_EXIT
K_LBUTTONDOWN	TBMouse()	see below
other codes		TBR_EXCEPTION

When the browse display is clicked with the mouse and a data cell is hit, :applyKey() returns TBR_CONTINUE. If no data is hit, the return code is TBR_EXCEPTION.

:hitTest()

Tests if a TBrowse is hit by the mouse cursor.

Syntax

```
:hitTest( <nMouseRow>, <nMouseCol> ) --> nHitCode
```

Arguments

<nMouseRow>

This is the numeric row position of the mouse cursor. It can be queried using function MRow().

<nMouseCol>

This is the numeric column position of the mouse cursor. It can be queried using function MCol().

Description

Method :hitTest() accepts the numeric row and column position of the mouse cursor and returns a numeric value indicating whether or not the mouse cursor is located within the TBrowse object. #define constants are available in the BUTTON.CH file identifying the return value of :hitTest().

Return values of oTBrowse:hitTest()

Constant	Value	Description
HTNOWHERE	0	Mouse cursor is outside the browse display
HTCELL	-5121	Mouse cursor is inside a data cell
HTHEADING	-5122	Mouse cursor is inside a column heading

:setKey()

Sets or retrieves a code block associated with a navigation key.

Syntax

```
:setKey( <nInkey>, [<bNewBlock>] ) --> bOldBlock
```

Arguments

<nInkey>

This is the numeric Inkey() code of the key to be associated with a code block.

<bNewBlock>

This is an optional code block to be associated with the inkey code *<nInkey>*.

Description

Method :setKey() maintains a dictionary of inkey codes and associated code blocks that are processed in method :applyKey(). If *<bNewBlock>* is omitted, the method returns the current code block associated with *<nInkey>*. If *<bNewBlock>* is a code block, the return value is the replaced code block if an association exists for *<nInkey>*. An existing inkey/code block association is removed from the :setKey() dictionary when NIL is explicitly passed for *<bNewBlock>*.

When a new code block is added to the :setKey() dictionary, it should accept two parameters since the TBrowse object and the inkey code are passed to it in method :applyKey(). Also, the return value of the code block should be one of the #define constants TBR_EXIT, TBR_CONTINUE or TBR_EXCEPTION. Refer to method :applyKey() for a discussion of these constants.

Info

See also: DbSkipper(), SetColor(), TBColumn(), TBrowseDb(), TBrowseNew(), TBMouse()
Category: Object functions
Header: tbrowse.ch
Source: rtl\tbrowse.prg
LIB: xhb.lib
DLL: xhbdll.dll

Examples

```
// The example demonstrates the steps required for creating a
// browse view for a database file.

   #include "TBrowse.ch"

   PROCEDURE Main
      LOCAL oTBrowse, oTBColumn
```

```
      LOCAL bFieldBlock, cFieldName, i, nKey

      USE Customer ALIAS Cust

      // create TBrowse object
      oTBrowse := TBrowse():new( 2,2, MaxRow()-2, MaxCol()-2 )
      oTBrowse:headSep   := "-"
      oTBrowse:colorSpec := "N/BG,W+/R"

      // add code blocks for navigating the record pointer
      oTBrowse:goTopBlock    := {|| DbGoTop() }
      oTBrowse:goBottomBlock := {|| DbGoBottom() }
      oTBrowse:skipBlock     := {|nSkip| DbSkipper(nSkip) }

      // create TBColumn objects and add them to TBrowse object
      FOR i:=1 TO FCount()
         cFieldName  := FieldName( i )
         bFieldBlock := FieldBlock( cFieldName )
         oTBColumn   := TBColumn():new( cFieldName, bFieldBlock )
         oTBrowse:addColumn( oTBColumn )
      NEXT

      // display browser and process user input
      DO WHILE .T.
         oTBrowse:forceStable()
         nKey := Inkey(0)

         IF oTBrowse:applyKey( nKey ) == TBR_EXIT
            EXIT
         ENDIF
      ENDDO

      USE
   RETURN

// The example demonstrates the steps required for creating a
// browse view for a two dimensional array. Note that the data
// source and row pointer of the data source are stored in
// oTBrowse:cargo. The pseudo instance variables :data and :recno
// are translated by the preprocessor.

   #include "TBrowse.ch"

   #xtrans  :data   =>   :cargo\[1]
   #xtrans  :recno  =>   :cargo\[2]

   PROCEDURE Main
      LOCAL i, nKey, bBlock, oTBrowse, oTBColumn
      LOCAL aHeading := { "File Name", ;
                          "File Size", ;
                          "File Date", ;
                          "File Time", ;
                          "File Attr"  }
      LOCAL aWidth := { 20, 10, 9, 9, 9 }

      // Create TBrowse object
      // data source is the Directory() array
      oTBrowse := TBrowse():new( 2, 2, MaxRow()-2, MaxCol()-2 )
      oTBrowse:cargo         := { Directory( "*.*" ), 1 }

      oTBrowse:headSep       := "-"
      oTBrowse:colorSpec     := "N/BG,W+/R"
```

```
   // Navigation code blocks for array
   oTBrowse:goTopBlock    := {|| oTBrowse:recno := 1 }
   oTBrowse:goBottomBlock := {|| oTBrowse:recno := Len( oTBrowse:data ) }
   oTBrowse:skipBlock     := {|nSkip| ArraySkipper( nSkip, oTBrowse ) }

   // create TBColumn objects and add them to TBrowse object
   FOR i:=1 TO Len( aHeading )

      // code block for individual columns of the array
      bBlock    := ArrayBlock( oTBrowse, i )

      oTBColumn := TBColumn():new( aHeading[i], bBlock )
      oTBColumn:width := aWidth[i]

      oTBrowse:addColumn( oTBColumn )
   NEXT

   // display browser and process user input
   DO WHILE .T.
      oTBrowse:forceStable()
      nKey := Inkey(0)

      IF oTBrowse:applyKey( nKey ) == TBR_EXIT
         EXIT
      ENDIF
   ENDDO

RETURN

// This code block uses detached LOCAL variables to
// access single elements of a two-dimensional array.
FUNCTION ArrayBlock( oTBrowse, nSubScript )
RETURN {|| oTBrowse:data[ oTBrowse:recno, nSubScript ] }

// This function navigates the row pointer of the
// the data source (array)
FUNCTION ArraySkipper( nSkipRequest, oTBrowse )
   LOCAL nSkipped
   LOCAL nLastRec := Len( oTBrowse:data ) // Length of array

   IF oTBrowse:recno + nSkipRequest < 1
      // skip requested that navigates past first array element
      nSkipped := 1 - oTBrowse:recno

   ELSEIF oTBrowse:recno + nSkipRequest > nLastRec
      // skip requested that navigates past last array element
      nSkipped := nLastRec - oTBrowse:recno

   ELSE
      // skip requested that navigates within array
      nSkipped := nSkipRequest
   ENDIF

   // adjust row pointer
   oTBrowse:recno += nSkipped

// tell TBrowse how many rows are actually skipped.
RETURN nSkipped
```

TopBarMenu()

Creates a new TopBarMenu object.

Syntax

```
TopBarMenu():new( <nRow>, <nLeft>, <nRight> ) --> oTopBarMenu
```

Arguments

`<nRow>`

This is a numeric value specifying the screen row for displaying the menu bar. It is assigned to the instance variable :row. The range for *<nRow>* is between 0 and MaxRow().

`<nLeft>`

This is a numeric value specifying the left screen coordinate of the menu bar. It is assigned to the instance variable :left. Usually, *<nLeft>* is set to the value 0.

`<nRight>`

This is a numeric value specifying the right screen coordinate of the menu bar. It is assigned to the instance variable :right. Usually, *<nRight>* is set to the value of MaxCol().

Return

Function TopBarMenu() returns a TobParMenu object and method :new() initializes it.

Description

Objects of the TopBarMenu class are used to build a text-mode menu system. A TopBarMenu object generally serves as the main menu and displays its menu items horizontally in one row on the screen. Each menu item is provided as a MenuItem() object and added using method :addItem(). The creation of submenus is accomplished with the aid of Popup() menu objects.

When the menu is completely build, it is activated using method :modal() or by passing the TopBarMenu object to function MenuModal().

Instance variables

:cargo

Instance variable for user-defined purposes.

Data type: ANY
Default: NIL

Description

This instance variable is not used by a TopBarMenu object. It is available for user-defined purposes when an arbitrary value should be attached to a TopBarMenu object.

:colorSpec

Color string for the TopBarMenu display.

Data type: C
Default: "N/W,W/N,W+/W,W+/N,N+/W,W/N"

Description

The instance variable :colorSpec holds a color string with six color values. It is initialized with the string "N/W,W/N,W+/W,W+/N,N+/W,W/N". The individual color values are used for the following purposes:

Colors for TopBarMenu

Color value	Used for display of
1	unselected menu items
2	selected menu item
3	accelerator key of unselected menu items
4	accelerator key of selected menu item
5	disabled menu items
6	menu border

Note: the sixth color value is actually not used by a TobParMenu object but by Popup() menu objects for displaying the menu border.

:current

Currently selected menu item.

Data type: N
Default: 0

Description

The instance variable :current contains a numeric value indicating the ordinal position of the currently selected menu item. Items are numbered from 1 to :itemCount. If no menu item is selected, :current contains zero.

:itemCount

Number of menu items.

Data type: N
Default: 0

Description

The instance variable :itemCount contains a numeric value indicating the number of menu items, or MenuItem() objects, maintained by a TopBarMenu object.

:left

Numeric left screen column for display.

Data type: N
Default: NIL

Description

The instance variable :left contains a numeric value. It is the screen coordinate of the left column of the menu bar.

:right

Numeric right screen column for display.

Data type: N
Default: NIL

Description

The instance variable :right contains a numeric value. It is the screen coordinate of the right column of the menu bar.

:row

Numeric screen row for display.

Data type: N
Default: NIL

Description

The instance variable :row contains a numeric value. It is the screen row of the menu bar.

Menu item methods

:addItem()

Adds a MenuItem object to a TopBarMenu object.

Syntax

```
:addItem( <oMenuItem> ) --> self
```

Arguments

`<oMenuItem>`

This is the MenuItem() object to add to the TopBarMenu object.

Description

Method :addItem() accepts a MenuItem() object and adds it to the internal list of menu items. The return value is the *self* object. The instance variable :itemCount reflects the new number of MenuItem() objects maintained by a TopBarMenu object.

:delItem()

Deletes a MenuItem object from a TopBarMenu object.

Syntax

```
:delItem( <nItemPos> ) --> self
```

Arguments

```
<nItemPos>
```

This is a numeric value specifying the ordinal position of the menu item to delete from the TopBarMenu object. Menu items are numbered from 1 to :itemCount.

Description

Method :delItem() can be used to remove a single menu item from a TopBarMenu object. This can be useful when the number of selectable menu items changes dynamically at runtime, depending on a condition. Alternatively, single menu items can be disabled or enabled.

:getFirst()

Retrieves the ordinal position of the first selectable menu item.

Syntax

```
:getFirst() --> nFirstMenuItemPos
```

Description

Method :getFirst() returns a numeric value indicating the ordinal position of the first, or leftmost, selectable menu item. When a menu item is disabled, it cannot be accessed by the user. When no menu item can be selected, the return value is zero.

:getItem()

Retrieves a MenuItem object from a TopBarMenu object.

Syntax

```
:getItem( <nItemPos> ) --> oMenuItem
```

Arguments

```
<nItemPos>
```

This is a numeric value specifying the ordinal position of the menu item to retrieve from the TopBarMenu object. Menu items are numbered from 1 to :itemCount.

Description

Method :getItem() is used to retrieve the MenuItem() object at position *<nItemPos>* from a TopBarMenu object.

:getLast()

Retrieves the ordinal position of the last selectable menu item.

Syntax

```
:getLast() --> nLastMenuItemPos
```

Description

Method :getLast() returns a numeric value indicating the ordinal position of the last, or rightmost, selectable menu item. When a menu item is disabled, it cannot be accessed by the user. When no menu item can be selected, the return value is zero.

:getNext()

Retrieves the ordinal position of the next selectable menu item.

Syntax

```
:getNext() --> nNextMenuItemPos
```

Description

Method :getNext() returns a numeric value indicating the ordinal position of the selectable menu item to the right of the currently selected menu item. When a menu item is disabled, it cannot be accessed by the user. If the current menu item is the last one, or when no menu item can be selected, the return value is zero.

:getPrev()

Retrieves the ordinal position of the previous selectable menu item.

Syntax

```
:getPrev() --> nPrevMenuItemPos
```

Description

Method :getPrev() returns a numeric value indicating the ordinal position of the selectable menu item to the left of the currently selected menu item. When a menu item is disabled, it cannot be accessed by the user. If the current menu item is the first one, or when no menu item can be selected, the return value is zero.

:insItem()

Inserts a MenuItem object into a TopBarMenu object.

Syntax

```
:insItem( <nItemPos>, <oMenuItem> ) --> self
```

Arguments

`<nItemPos>`

This is a numeric value specifying the ordinal position where to insert the menu item. The value for *<nItemPos>* must be in the range from 1 to :itemCount. Otherwise, it is ignored.

`<oMenuItem>`

This is the MenuItem() object to insert into the TopBarMenu object.

Description

Method :insItem() accepts a MenuItem() object and inserts it into the internal list of menu items at position *<nItemPos>*. The return value is the *self* object. The instance variable :itemCount reflects the new number of MenuItem() objects maintained by a TopBarMenu object. If *<nItemPos>* is outside the valid range, the number of menu items remains unchanged.

:setItem()

Replaces a MenuItem object in a TopBarMenu object.

Syntax

```
:setItem( <nItemPos>, <oMenuItem> ) --> self
```

Arguments

`<nItemPos>`

This is a numeric value specifying the ordinal position of the menu item to replace. The value for *<nItemPos>* must be in the range from 1 to :itemCount. Otherwise, it is ignored.

`<oMenuItem>`

This is the MenuItem() object that replaces an existing one in the TopBarMenu object.

Description

Method :setItem() accepts a MenuItem() object and replaces the existing MenuItem object at position *<nItemPos>*. The return value is the *self* object. If *<nItemPos>* is outside the valid range, no menu item is replaced.

Menu display and selection

:display()

Displays the entire TopBarMenu.

Syntax

```
:display() --> self
```

Description

Method :display() iterates through the list of MenuItem() objects and displays their captions on the screen.

:getAccel()

Determines if a key code identifies an accelerator key.

Syntax

```
:getAccel( <nKey> ) --> nMenuItemPos
```

Arguments

`<nKey>`

This is a numeric Inkey() code to check.

Description

Method :getAccel() is used to check if a numeric inkey code is an accelerator key that should trigger a menu action. Accelerator keys are key combinations of the Alt key an a letter of a menu item caption prefixed with an ampersand (&). If the key code *<nKey>* is an accelerator key, the method returns the ordinal position of the corresponding menu item as a numeric value. If it is not an accelerator, the return value is zero.

:hitTest()

Checks if a menu item is clicked with the mouse.

Syntax

```
:hitTest( <nMouseRow>, <nMouseCol> ) --> nMenuItemPos
```

Arguments

<nMouseRow>

This is the numeric row position of the mouse cursor. It can be queried using function MRow().

<nMouseCol>

This is the numeric column position of the mouse cursor. It can be queried using function MCol().

Description

Method :hitTest() accepts the numeric row and column position of the mouse cursor and returns a numeric value indicating the ordinal position of the menu item that is underneath the mouse cursor. If the mouse is outside the menu bar, the return value is zero.

:modal()

Activates the menu.

Syntax

```
:modal(  <nStartItem>, ;
        [<nMsgRow>]  , ;
        [<nMsgLeft>] , ;
        [<nMsgRight>], ;
        [<cMsgColor>], ;
        [<GetList>]    ) --> nMenuItemID
```

Arguments

<nStartItem>

This is a numeric value indicating the ordinal position of the first menu item to be selected when the menu is activated.

<nMsgRow>

This is a numeric value specifying the screen row for displaying messages assigned to the instance variable :message of menu items. The range for *<MsgnRow>* is between 0 and MaxRow().

<nMsgLeft>

This is a numeric value specifying the left screen coordinate for displaying menu messages. Usually, *<nMsgLeft>* is set to the value 0.

<nMsgRight>

This is a numeric value specifying the right screen coordinate for displaying menu messages. Usually, *<nMsgRight>* is set to the value of MaxCol().

<cMsgColor>

The parameter *<cMsgColor>* is an optional character string defining the color for the message to display. It defaults to SetColor().

<GetList>

Optionally, a *GetList* array can be passed when the menu is activated during READ.

Description

Method :modal() activates the top bar menu and processes user input with the cursor keys. To react to mouse events, SET EVENTMASK must be set accordingly. When the user selects a menu item, the corresponding :data code block is evaluated, if present. The method returns the ordinal position of the selected menu item. If the user cancels menu selection with the Esc key, the return value is zero.

:select()

Changes the currently selected menu item.

Syntax

```
:select( <nItemPos> ) --> nMenuItemPos
```

Arguments

<nItemPos>

This is a numeric value specifying the ordinal position of the menu item to select as current. The value for *<nItemPos>* must be in the range from 1 to :itemCount. Otherwise, it is ignored.

Description

Method :select() defines the menu item at position *<nItemPos>* as currently selected item. The selected menu item is displayed highlighted when method :display() is executed.

Info

See also:	@...GET, MenuItem(), MenuModal(), Popup(), READ, SetColor()
Category:	Get system, Object functions
Header:	Button.ch
Source:	rtl\ttopbar.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example outlines the construction of a text-mode
// menu system with user defined routines. CreateMainMenu()
// returns a TopBarMenu object, CreateSubMenu() builds
// pull-down menus, and MenuSelect() branches to subroutines of
// the program. The menu is activated in a DO WHILE .T. loop.
// This requires a separate Exit routine for program termination.

   #include "Button.ch"
   #include "Inkey.ch"

   PROCEDURE Main
      LOCAL oTopBar := CreateMainMenu()

      CLS

      DO WHILE .T.
         MenuModal( oTopBar , 1, ;
                    MaxRow(), 0, MaxCol(), ;
                    oTopBar:colorSpec )
      ENDDO
```

```
RETURN


FUNCTION CreateMainMenu()
   LOCAL oMainMenu  := TopBarMenu():new( 0, 0, MaxCol() )
   LOCAL bMenuBlock := {|o| MenuSelect(o) }
   LOCAL cMenuColor := "N/BG,W+/R,GR+/BG,GR+/R,N+/BG,N/BG"
   LOCAL aItems

   oMainMenu:colorSpec := cMenuColor

   aItems := { ;
      { " &Open "     , K_ALT_O    , "Open routine"   , 11 }, ;
      { " &Save "     , K_ALT_S    , "Save routine"   , 12 }, ;
      { MENU_SEPARATOR,            ,                  , 13 }, ;
      { " E&xit "     , K_ALT_X    , "Exit program"   , 14 }  ;
   }

   CreateSubMenu( oMainMenu, " &File ", bMenuBlock, aItems )

   aItems := { ;
      { " Cop&y   "   , K_CTRL_INS, "Copy routine"   , 21 }, ;
      { " &Paste  "   , K_SH_INS   , "Paste routine" , 22 }, ;
      { MENU_SEPARATOR,            ,                  , 23 }, ;
      { " C&ut    "   , K_SH_DEL   , "Cut routine"    , 24 }, ;
      { " &Delete "   , K_DEL      , "Delete routine", 25 }  ;
   }

   CreateSubMenu( oMainMenu, " &Edit ", bMenuBlock, aItems )

   aItems := { ;
      { " &Info   "   , K_F1       , "Help routine"   , 31 }, ;
      { " &About  "   ,            , "About program" , 32 }  ;
   }

   CreateSubMenu( oMainMenu, " &Help ", bMenuBlock, aItems )

RETURN oMainMenu


FUNCTION CreateSubMenu( oMenu, cMenuItem, bBlock, aItems )
   LOCAL aItem, oItem, oSubMenu

   oSubMenu           := PopUp():new()
   oSubMenu:colorSpec := oMenu:colorSpec

   FOR EACH aItem IN aItems
      oItem := MenuItem():new( aItem[1], ;
                               bBlock  , ;
                               aItem[2], ;
                               aItem[3], ;
                               aitem[4]  )
      oSubMenu:addItem ( oItem )
   NEXT

   oItem := MenuItem():new( cMenuItem, oSubMenu )
   oMenu:addItem( oItem )
RETURN


PROCEDURE MenuSelect( oMenuItem )
   @ 1, 0 CLEAR TO MaxRow(), MaxCol()
```

```
   SWITCH oMenuItem:ID
   CASE 14
      ExitRoutine() ; EXIT
   DEFAULT
      Alert( oMenuItem:message )
   END
RETURN


PROCEDURE ExitRoutine
   IF Alert( "Exit program?", { "Yes", "No" } ) == 1
      QUIT
   ENDIF
RETURN
```

THtmlDocument()

Creates a new THtmlDocument object.

Syntax

```
THtmlDocument():new( [<cHtmlDocument>] ) --> oTHtmlDocument
```

Arguments

<cHtmlDocument>

This is an optional HTML formatted character string as returned from TIpClientHTTP():readAll(). If *<cHtmlDocument>* does not contain the HTML tags <html>, <head> or <body>, the missing tag is inserted into the HTML document.

Return

Function THtmlDocument() creates the object and method :new() initializes it.

Description

The THtmlDocument() class provides objects for reading and creating HTML files and streams. HTML stands for **H**yper **T**ext **M**arkup **L**anguage which is the standard file format for documents published in the internet. To learn more about HTML itself, the internet provides very good free online tutorials. The website www.w3schools.com is is a good place to quickly learn the basics on HTML.

A THtmlDocument object maintains an entire HTML document and builds from it a tree of THtmlNode() objects which contain the actual HTML data. The first HTML node is stored in the :root instance variable, which is the root node of the HTML tree. Beginning with the root node, an HTML document can be traversed or searched for particular data. The classes THtmlIteratorScan() and THtmlIteratorRegEx() are available to find a particular HTML node, based on its tag name, attribute or textual content.

Besides the root node, a THtmlDocument object has two standard nodes :head and :body.

Instance variables

:body

<body> node of the HTML document.

Data type: O (READONLY)
Default: oTHtmlNode

Description

The instance variable :body contains a THtmlNode() object representing the *<body>* node of the HTML document. This node follows the *<head>* node and contains the textual content of the HTML document.

:changed

Changed flag.

Data type: L
Default: .F.

Description

The instance variable :changed contains .T. (true) when a node within the tree of THtmlNode objects is modified or deleted. Method ::collect() sets :changed to .F. (false).

:head

<head> node of the HTML document.

Data type: O (READONLY)
Default: oTHtmlNode

Description

The instance variable :head contains a THtmlNode() object representing the *<head>* node of the HTML document. This node follows the *<html>* node

:root

Root node of the HTML document.

Data type: O (READONLY)
Default: oTHtmlNode

Description

The instance variable :root contains a THtmlNode() object representing the root node of the HTML document. It can be used as starting point for searching the node tree with iterator objects like THtmlIteratorScan().

Methods for files and streams

:readFile()

Reads a HTML file.

Syntax

```
:readFile( <cFileName> ) --> lSuccess
```

Arguments

`<cFileName>`

This is a character string holding the name of the HTML file to read.

Description

Method :readFile() reads a HTML file from disk and builds a new tree of THtmlNode() objects. If the file does not exist, the return value is .F. (false) and the THtmlDocument object remains unchanged.

:toString()

Creates a HTML formatted string.

Syntax

```
:toString() --> cHtmlDocument
```

Description

Method :toString() creates an HTML formatted character string containing all nodes and data currently held in the HTML document.

:writeFile()

Creates a HTML file.

Syntax

```
:writeFile( <cFileName> ) --> lSuccess
```

Arguments

`<cFileName>`

This is a character string holding the name of the HTML file to create.

Description

Method :writeFile() writes the return value of :toString() to the file *<cFileName>*. The return value is .T. (true) when the file is successfully created, otherwise .F. (false) is returned.

Methods for searching and navigating

:collect()

Returns all nodes of the HTML document.

Syntax

```
:collect() --> aTHtmlNodes
```

Description

Method :collect() returns a one dimensional array holding all nodes of the HTML document in sequential order.

:findFirst()

Locates the first HTML node containing particular data.

Syntax

```
:findFirst( [<cTagName>]        , ;
            [<cAttributeName>]  , ;
            [<cAttributeValue>] , ;
            [<cText>]           ) --> oTHtmlNode | NIL
```

Arguments

<cTagName>

This is a character string holding the name of the HTML node to find.

<cAttributeName>

This is a character string holding the name of an attribute of the HTML node to find.

<cAttributeValue>

This is a character string holding the value of the attribute of the HTML node to find.

<cText>

This is a character string holding the textual content of the HTML node to find.

Description

Method :findFirst() locates the first HTML node in the HTML document that matches the search criteria. If no parameter is passed, the first HTML node in the tree is returned. Search criteria can be any combination of name, attribute and textual content. If no matching node is found, the return value is NIL.

If a matching node is found, the next node matching the same search criteria is returned from method :findNext().

:findFirstRegex()

Locates the first HTML node containing particular data using regular expressions.

Syntax

```
:findFirstRegEx( [<cTagNameRegEx>]        , ;
                 [<cAttributeNameRegEx>]  , ;
                 [<cAttributeValueRegEx>] , ;
                 [<cDataRegEx>]             ) --> oTHtmlNode | NIL
```

Arguments

<cTagNameRegEx>

This is a character string holding the regular expression to match with name of the HTML node to find.

<cAttributeNameRegEx>

This is a character string holding the regular expression to match with the name of an attribute of the HTML node to find.

<cAttributeValueRegEx>

This is a character string holding the regular expression to match with the value of the attribute of the HTML node to find.

<cDataRegEx>

This is a character string holding the regular expression to match with the textual content of the HTML node to find.

Description

Method :findFirstRegEx() locates the first HTML node in the HTML document that matches the regular expressions passed. If no parameter is passed, the first HTML node in the tree is returned. Search criteria can be any combination of name, attribute and textual content. If no matching node is found, the return value is NIL.

If a matching node is found, the next node matching the same search criteria is returned from method :findNext().

:findNext()

Finds the next HTML node matching a search criteria.

Syntax

```
:findNext() --> oTHtmlNode|NIL
```

Description

Method :findNext() continues the search for a matching HTML node and returns the corresponding THtmlNode object. The search criteria defined with a previous call to :findFirst() or :findFirstRegEx() is used. If no further HTML node is found, the return value is NIL.

:getNode()

Returns the first node matching a tag name.

Syntax

```
:getNode( <cTagName> ) --> oTHtmlNode|NIL
```

Arguments

<cTagName>

This is a character string holding the tag name of the HTML node to find.

Description

Method :getNode() returns the first THtmlNode() object having the tag name *<cTagName>*. If no HTML node with this tag name exists, the return value is NIL.

:getNodes()

Syntax

```
:getNodes( <cTagName> ) --> aTHtmlNodes
```

Arguments

<cTagName>

This is a character string holding the tag name of the HTML nodes to find.

Description

Method :getNodes() returns an array of all THtmlNode() objects having the tag name *<cTagName>*. If no HTML node with this tag name exists, the return value is an empty array.

Info

See also: THtmlCleanup(), THtmlInit(), THtmlNode(), TIpClientHttp()
Category: HTML functions, Object functions, xHarbour extensions
Source: tip\thtml.prg
LIB: xhb.lib
DLL: xhbdll.dll

Examples

Creating a simple HTML page

```
// The example creates a HTML document from a simple HTML string

   PROCEDURE Main
      LOCAL cString  := "<p>Hello <p>world"
      LOCAL oHtmlDoc := THtmlDocument():new( cString )

      ? oHtmlDoc:toString()

      ** output
      // <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
      // <html>
      //  <head>
      //  </head>
      //  <body>
      //   <p>Hello
      //   <p>world
      //  </body>
      // </html>

      THtmlCleanUp()
   RETURN
```

Loading a HTML page from the internet

```
// The example loads a HTML page from Google and lists
// the links contained in the HTML response.

   PROCEDURE Main
      LOCAL oHttp, cHtml, hQuery, oHtmlDoc, oNode, aLink

      oHttp:= TIpClientHttp():new( "http://www.google.de/search" )

      // build the Google query
      hQUery := Hash()
      hSetCaseMatch( hQuery, .F. )

      hQuery["q"]    := "xHarbour"
      hQuery["hl"]   := "en"
      hQuery["btnG"] := "Google+Search"

      // add query data to the TUrl object
      oHttp:oUrl:addGetForm( hQuery )

      // Connect to the HTTP server
      IF .NOT. oHttp:open()
         ? "Connection error:", oHttp:lastErrorMessage()
         QUIT
      ENDIF
```

```
   // downlowad the Google response
   cHtml   := oHttp:readAll()
   oHttp:close()
   ? Len(cHtml), "bytes received "

   oHtmlDoc := THtmlDocument():new( cHtml )

   oHtmlDoc:writeFile( "Google.html" )

   // ":a" retrieves the first <a href="url"> text </a> tag
   oNode := oHtmlDoc:body:a
   ? oNode:getText(""), oNode:href

   // ":divs(5)" returns the 5th <div> tag
   oNode := oHtmlDoc:body:divs(5)

   // "aS" is the plural of "a" and returns all <a href="url"> tags
   aLink := oNode:aS

   FOR EACH oNode IN aLink
      ? HtmlToOem( oNode:getText("") ), oNode:href
   NEXT
RETURN
```

THtmlIterator()

Creates a new THtmlIterator object.

Syntax

```
THtmlIterator():new( <oTHtmlNode> ) --> oTHtmlIterator
```

Arguments

<oTHtmlNode>

This is a THtmlNode() object to create the iterator object for.

Return

The function returns a THtmlIterator object and method :new() initializes it.

Description

The THtmlIterator class provides objects for iterating nodes in an HTML document and its sub-nodes. An HTML document is managed by an object of the THtmlDocument() class.

The creation of a THtmlIterator object requires a THtmlNode() object wich serves as starting point for the iterator. The iterator is restricted to the branch in the HTML tree represented by the initial HTML node.

Methods

:clone()

Clones the THtmlIterator object.

Syntax

```
:clone() --> oClone
```

Description

Method :clone() creates a duplicate of the self object. The returned THtmlIterator object uses the same search criteria and the same THtmlNode object for a search.

:getNode()

Retrieves the current HTML node matching the search criteria.

Syntax

```
:getNode() --> oTHtmlNode | NIL
```

Description

Method :getNode() returns the current THtmlNode() object as it is determined by a previous call to :next(). If no further HTML node exists, the return value is NIL.

:next()

Retrieves the next HTML node in the tree.

Syntax

```
:next() --> oTHtmlNode | NIL
```

Description

Method :next() returns the next THtmlNode() object. If no further HTML node is found, the return value is NIL.

:rewind()

Goes back to the first HTML node matching the search criteria.

Syntax

```
:rewind() --> oTHtmlNode | NIL
```

Description

Method :rewind() sets the HTML node passed to the :new() method as the current node, thus rewinding the entire iteration.

:setContext()

Defines the currently found HTML node as first node.

Syntax

```
:setContext() --> self
```

Description

Method :setContext() defines the currently found HTML node as the first node. An iteration can then be restarted with this node when method :rewind() is called.

Info

See also:	THtmlDocument(), THtmlNode(), THtmlIteratorScan(), THtmlIteratorRegEx()
Category:	HTML functions, Object functions, xHarbour extensions
Source:	tip\thtml.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example use the Google.html file as created with the
// THtmlDocument() example, extracts from it the <form>.
// tag and displays contained <input> variables.

   PROCEDURE Main
      LOCAL oHtmlDoc := THtmlDocument():new()
      LOCAL oHtmlNode, oHtmlIter

      oHtmlDoc:readFile( "Google.html" )

      oHtmlNode := oHtmlDoc:findFirst( "form" )
      oHtmlIter := THtmlIterator():new( oHtmlNode )
```

```
      DO WHILE .T.
         IF Lower( oHtmlNode:htmlTagName ) == "input"
            ? oHtmlNode:name , ;
              oHtmlNode:value, ;
              oHtmlNode:type
         ENDIF

         oHtmlNode := oHtmlIter:next()

         IF oHtmlNode == NIL
            EXIT
         ENDIF
      ENDDO
   RETURN
```

THtmlIteratorRegEx()

Creates a new THtmlIteratorRegEx object.

Syntax

```
THtmlIteratorRegEx():new( <oTHtmlNode> ) --> oTHtmlIteratorRegEx
```

Arguments

<oTHtmlNode>

This is a THtmlNode() object to create the iterator object for. It serves as starting node for iterating.

Return

The function returns a THtmlIteratorRegEx object and method :new() initializes it.

Description

The THtmlIteratorRegEx class is derived from the THtmlIterator() class and has the same methods. The only difference is that regular expressions are used to define search criteria for finding a particular HTML node in an HTML tree. The regular expressions are defined once with the :find() method, which searches for the first matching HTML node. Subsequent HTML nodes matching the regular expressions are then searched with the :next() method.

The end of a search is indicated when either :find() or :next() return NIL instead of a THtmlNode object matching the regular expressions.

Search methods

:find()

Searches the first HTML node matching the regular expressions.

Syntax

```
:find( [<cRegExTagName>]  , ;
       [<cRegExAttribute>], ;
       [<cRegExValue>]    , ;
       [<cRegExData>]       ) --> oTHtmlNode | NIL
```

Arguments

<cRegExTagName>

This is a character string holding the regular expression for matching the tag name of HTML nodes.

<cRegExAttribute>

This is a character string holding the regular expression for matching the name of an HTML attribute.

<cRegExValue>

The a regular expression to match an attribute value optionally defined with the character string *<cRegExValue>*.

`<cRegExData>`

This is an optional character string holding the regular expression for matching the textual data of HTML nodes.

Description

Method :find() defines the regular expressions for a THtmlIteratorRegEx object and performs the initial search. Four different regular expressions can be defined to match tag name, attribute name, attribute value or textual data. If a matching HTML node exists, the corresponding THtmlNode() object is returned, otherwise the return value is NIL. The search can be continued using the same regular expressions with method :next().

:next()

Searches the next HTML node matching the regular expressions.

Syntax

```
:next() --> oTHtmlNode | NIL
```

Description

Method :next() returns the next THtmlNode() object matching the regular expressions defined with the :find() method. If no further HTML node is found, the return value is NIL.

Info

See also:	THtmlDocument(), THtmlNode(), THtmlIterator(), THtmlIteratorScan()
Category:	HTML functions, Object functions, xHarbour extensions
Source:	tip\thtml.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

THtmlIteratorScan()

Creates a new THtmlIteratorScan object.

Syntax

```
THtmlIteratorScan():new( <oTHtmlNode> ) --> oTHtmlIteratorScan
```

Arguments

<oTHtmlNode>

This is a THtmlNode() object to create the iterator object for. It is the starting point for searching nodes in the HTML tree.

Return

The function returns a THtmlIteratorScan object and method :new() initializes it.

Description

The THtmlIteratorScan class is derived from the THtmlIterator() class and has the same methods. The only difference is that search criteria can be defined to find a particular HTML node in an HTML tree. The search criteria is defined once with the :find() method, which searches for the first matching HTML node. Subsequent HTML nodes matching the search criteria are then searched with the :next() method.

The end of a search is indicated when either :find() or :next() return NIL instead of a THtmlNode object matching the search criteria.

Search methods

:find()

Searches the first HTML node matching the search criteria.

Syntax

```
:find( [<cTagName>]  , ;
       [<cAttribute>], ;
       [<cValue>]    , ;
       [<cData>]       ) --> oTHtmlNode | NIL
```

Arguments

<cTagName>

This is a character string holding the tag name of the HTML node to search for.

<cAttribute>

This is a character string holding the name of an HTML attribute to search for.

<cValue>

The attribute value to search is optionally defined with the character string *<cValue>*.

<cData>

This is an optional character string holding the textual data of an HTML node to search.

Description

Method :find() defines the search criteria for a THtmlIteratorScan object and performs the initial search. The search criteria can be defined as any combination of tag name, attribute name, attribute value or textual data. If a matching HTML node exists, the corresponding THtmlNode() object is returned, otherwise the return value is NIL. The search can be continued using the same search criteria with method :next().

:next()

Searches the next HTML node matching the search criteria.

Syntax

```
:next() --> oTHtmlNode | NIL
```

Description

Method :next() returns the next THtmlNode() object matching the search criteria defined with the :find() method. If no further HTML node is found, the return value is NIL.

Info

See also:	THtmlDocument(), THtmlNode(), THtmlIterator(), THtmlIteratorRegEx()
Category:	HTML functions, Object functions, xHarbour extensions
Source:	tip\thtml.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example use the Google.html file as created with the
// THtmlDocument() example. A HtmlIteratorScan object is used
// to scan the <form> node for <input> nodes.

   PROCEDURE Main
      LOCAL oHtmlDoc := THtmlDocument():new()
      LOCAL oHtmlNode, oHtmlIter

      oHtmlDoc:readFile( "Google.html" )

      oHtmlIter := THtmlIteratorScan():new( oHtmlDoc:body:form )
      oHtmlNode := oHtmlIter:find( "input" )

      DO WHILE oHtmlNode <> NIL

         ? oHtmlNode:name, oHtmlNode:value, oHtmlNode:type

         oHtmlNode := oHtmlIter:next()
      ENDDO
   RETURN
```

THtmlNode()

Creates a new THtmlNode object.

Syntax

```
THtmlNode():new( <cHtmlString> ) --> oTHtmlNode

or

THtmlNode():new( <oTHtmlNode>   , ;
                 <cTagName>     , ;
                [<hAttributes>], ;
                [<cText>]        ) --> oTHtmlNode
```

Arguments

`<cHtmlString>`

When a character string is passed to method :new(), the string is parsed into a tree of THtmlNode objects. The returned THtmlNode object represents the root node.

`<oTHtmlNode>`

When a THtmlNode object is passed as first parameter, it serves as parent node for the new object created.

`<cTagName>`

This is a character string holding the tag name of the THtmlNode object.

`<hAttributes>`

Optionally, the attributes of of the HTML node to create can be passed as a Hash. The Hash key is the attribute name and the Hash value is the attribute value. Note that both, Hash values and keys, must be of data type Character.

Alternatively, a HTML compliant character string holding the attributes of the THtmlNode object can be passed (e.g. 'border="0" cellspacing="0" cellpadding="0"' are attributes for a *<table>* node).

`<cText>`

This is a character string holding the textual content of the THtmlNode object. If this parameter is used, the THtmlNode object represents a leaf, not a node. *<cText>* is the text appearing between the opening and closing HTML *<cTagName>*.

Return

The function returns a THtmlNode object and method :new() initializes the object.

Description

Objects of the THtmlNode class represent a single node, or a leaf, in an HTML document. HTML documents are maintained by the THtmlDocument() class which builds a tree of HTML nodes when reading an HTML document. A THtmlDocument object creates this tree and fills it with THtmlNode objects.

The THtmlNode class is designed for comfortable HTML processing from PRG source code. The creation of HTML documents is supported by operator overloading and built in lookup tables representing all HTML 4.0 tag names and associated attributes. Refer to **Overloaded operators** below.

Note: when no THtmlNode object exists anymore, the memory resources for the HTML lookup tables should be released with function THtmlCleanup().

Html Attribute data

:attr

Attributes of an HTML node.

Data type: H
Default: NIL

Description

:attr is a generic Access/Assign variable returning all attributes of the THtmlNode as a Hash, or collecting one or more attributes upon assignment. The following code shows valid usage examples:

```
? oNode:htmlTagName                    // result: table

// assigning a single attribute
oNode:attr := 'border="0"'
? oNode:attrToString()                 // result: border="0"

// assigning multiple attributes as character string
oNode:attr := 'border=1 cellspacing=1'
? oNode:attrToString()                 // result: border="1" cellspacing="1"

// assigning multiple attributes as Hash
oNode:attr := { "border" => "0", "cellpadding" => "0" }
? oNode:attrToString()                 // result: border="0" cellpadding="0"
```

Html Node data

:childNodes

Child nodes of an HTML node.

Data type: A (READONLY)
Default: NIL

Description

The instance variable :childNodes contains a one dimensional array of THtmlNode objects representing the child nodes of the *self* object. When the object is a leaf, :childNodes contains NIL.

:document

Document of an HTML node.

Data type: O (READONLY)
Default: NIL

Description

The instance variable :document contains the THtmlDocument() object that has created the *self* object in the course of building a tree of THtmlNode objects representing the HTML document. When the *self* object is not created via a THtmlDocument object, :document contains NIL.

:nextNode

Next HTML node.

Data type: O (READONLY)
Default: NIL

Description

The instance variable :nextNode contains the next THtmlNode object in the tree of objects regardless of the nesting level. If the *self* object is the last node, ::nextNode contains NIL.

:parentNode

Parent node

Data type: O (READONLY)
Default: NIL

Description

The instance variable :parentNode contains the parent object of the *self* object in the object tree. If the *self* object is the root object, ::parentNode contains NIL.

:prevNode

Previous HTML node.

Data type: O (READONLY)
Default: NIL

Description

The instance variable :prevNode contains the previous THtmlNode object in the tree of objects regardless of the nesting level. If the *self* object is the first node, ::prevNode contains NIL.

:siblingNodes

Sibling nodes of an HTML node.

Data type: A (READONLY)
Default: NIL

Description

The instance variable :siblingNodes contains a one dimensional array of THtmlNode objects representing the sibling nodes of the *self* object. Sibling nodes have the same nesting level in the tree of objects as the *self* object. When the object is a leaf or the root object, :siblingNodes contains NIL.

Note: if :siblingNodes contains an array, the *self* object is contained in this array as well.

Html Tag data

:htmlEndTagName

Character string with closing tag name.

Data type: C (READONLY)
Default: NIL

Description

The instance variable :htmlEndTagName contains a character string holding the closing tag name of the HTML node without angled brackets. (e.g. "/table", "/html"). If the HTML node has no closing tag, :htmlEndTagName contains NIL.

:htmlTagName

Character string with opening tag name.

Data type: C (READONLY)

Description

The instance variable :htmlEndTagName contains a character string holding the opening tag name of the HTML node without angled brackets. (e.g. "table", "html").

Note: two non HTML compliant tag names are used in the THtmlNode class: "_root_" is the tag name for the root node, and "_text_" is the tag name for leaf nodes that contain nothing but textual data. Both tag names never appear in HTML output, but may be useful when traversing an HTML object tree.

Another special case is a "<!-- HTML comment -->" node. The tag name of a THtmlNode object representing an HTML comment is "!--".

:htmlTagType

Array specifying the type of an HTML tag.

Data type: A (READONLY)

Description

The instance variable :htmlTagType contains an array with two elements. It is used for internal purposes and usually not required in daily programming with the THtmlNode class.

:htmlTagType[1]

If the first element is NIL, the HTML tag has no attributes. Otherwise, :htmlTagType[1] contains a function pointer. Passing this pointer to HB_Exec() yields a two dimensional array. The first column holds character strings describing valid attribute names for the HTML tag, and the second column identifies the type of the value valid for the attribute. Types for attribute values are identified with #define constants listed in Thtml.ch. They begin with the prefix HTML_ATTR_TYPE_.

The following code shows a usage example that lists all attributes valid for the *<table>* tag:

```
PROCEDURE Main
   LOCAL oDoc   := THtmlDocument():new()
   LOCAL oNode  := oDoc:body + "table"
   LOCAL aAttrs := HB_Exec( oNode:htmlTagType[1] )
```

```
    LOCAL aAttr

    FOR EACH aAttr IN aAttrs
       ? aAttr[1]
    NEXT
 RETURN
```

Note: the only value type that is currently checked in the THtmlNode class is HTML_ATTR_TYPE_BOOL.

:htmlTagType[2]

The second element contains a numeric value which is a combination of one of more #define constants listed in Thtml.ch. They begin with the prefix CM_ and describe the content model of an HTML tag.

:isBlock

Checks if an HTML tag designates a block of text.

Data type: L (READONLY)
Default: .F.

Description

The instance variable :isBlock contains .T. (true) when the HTML node contains a block of text, otherwise .F. (false).

HTML tags for text blocks

HTML tag	
<address>	*<isindex>*
<align>	*<layer>*
<area>	*<listing>*
<blockquote>	*<menu>*
<center>	*<multicol>*
<del>	*<noframes>*
<dir>	*<nolayer>*
<div>	*<nosave>*
<dl>	*<noscript>*
<fieldset>	* *
<form>	*<p>*
<h1>	*<plaintext>*
<h2>	*<pre>*
<h3>	*<script>*
<h4>	*<server>*
<h5>	*<table>*
<h6>	* *
<hr>	*<xmp>*
<ins>	

:isEmpty

Checks if an HTML tag is always empty.

Data type: L (READONLY)
Default: .T.

Description

The instance variable :isEmpty contains .T. (true) when the HTML node has never a child node, otherwise .F. (false).

Empty HTML tags

HTML tag	
<area>	*<input>*
<base>	*<isindex>*
<basefont>	*<keygen>*
<bgsound>	*<link>*
* *	*<meta>*
<col>	*<nextid>*
<embed>	*<param>*
<frame>	*<spacer>*
<hr>	*<wbr>*
<img>	

:isInline

Checks if an HTML tag may appear inline.

Data type: L (READONLY)
Default: .T.

Description

The instance variable :isInline contains .T. (true) when the HTML node may appear within textual content, otherwise .F. (false).

Inline tags

HTML tag		
<a>	*<ilayer>*	*<rt>*
<abbr>	*<img>*	*<rtc>*
<acronym>	*<input>*	*<ruby>*
<applet>	*<ins>*	*<s>*
<b>	*<kbd>*	*<samp>*
<basefont>	*<keygen>*	*<script>*
<bdo>	*<label>*	*<select>*
<big>	*<legend>*	*<server>*
<blink>	*<map>*	*<servlet>*
* *	*<marquee>*	*<small>*
<button>	*<nobr>*	*<spacer>*
<cite>	*<noembed>*	*<span>*
<code>	*<nolayer>*	*<strike>*
<comment>	*<noscript>*	*<strong>*
<del>	*<object>*	*<sub>*
<dfn>	*<param>*	*<sup>*
<em>	*<q>*	*<textarea>*
<embed>	*<rb>*	*<tt>*

<font>	*<rbc>*	*<var>*
<i>	*<u>*	*<wbr>*
<iframe>	*<rp>*	

:isNode

Checks if this is a node.

Data type: L (READONLY)
Default: .F.

Description

The instance variable :isNode contains .T. (true) when the *self* object contains child nodes, otherwise .F. (false). When :isNode is .F. (false) the THtmlNode object contains only textual content, if any.

:isOptional

Checks if an HTML tag has an optional end tag.

Data type: L (READONLY)
Default: .F.

Description

The instance variable :isOptional contains .T. (true) when the HTML node may appear without an end tag, otherwise .F. (false).

HTML tags with optional end tags

HTML tag	
<body>	*<option>*
<colgroup>	*<p>*
<dd>	*<tbody>*
<dt>	*<td>*
<head>	*<tfoot>*
<html>	*<th>*
* *	*<thead>*
<marquee>	*<tr>*
<optgroup>	

Html Text data

:text

Text node.

Data type: O
Default: oTHtmlNode

Description

:text is a generic Access/Assign variable returning a new text node and optionally assigning textual content to it. The following PRG code demonstrates how HTML text can be created using the :text variable:

```
oDoc := THtmlDocument():new()

? oDoc:body:toString()
```

```
   ** output
   //  <body>
   //  </body>

   oDoc:body:text := "Hello"
   oDoc:body:text := "-World"
   oDoc:body:text := "!!"

   ? oDoc:body:toString()
   ** output
   //  <body>Hello-World!!
   //  </body>
!CEND

!P[note=Note:]
use method :getText() to retrieve the entire text contained
in an HTML node.

!! =========================
```

Html Attribute methods

:attrToString()

Returns all HTML attributes as a character string.

Syntax

```
:attrToString() --> cAttributes
```

Description

Method :attrToString() returns a character string holding all attributes set for the THtmlNode object. If no attributes are set, the method returns an empty string ("").

:delAttribute()

Deletes an HTML attribute.

Syntax

```
:delAttribute( <cAttrName> ) --> lSuccess
```

Arguments

<cAttrName>

This is a character string holding the name of the attribute to delete. It is case insensitive.

Description

Method :delAttribute() removes a single HTML attribute from the THtmlNode object and returns a logical value indicating success of the operation. If the attribute was not set, the return value is .F. (false).

:delAttributes()

Removes all HTML attributes.

Syntax

```
:delAttributes() --> lSuccess
```

Description

Method :delAttributes() removes all HTML attributes from the THtmlNode object and returns a logical value indicating success of the operation.

:getAttribute()

Returns the value of an HTML attribute.

Syntax

```
:getAttribute( <cAttrName> ) --> cAttrValue|NIL
```

Arguments

<cAttrName>

This is a character string holding the name of the attribute to query. It is case insensitive.

Description

Method :getAttribute() returns the value of an HTML attribute as a character string, or NIL if the attribute is not present.

:getAttributes()

Returns all HTML attributes.

Syntax

```
:getAttributes() --> hHash
```

Description

Method :getAttributes() returns all HTML attributes as a Hash. The hash keys are the attribute names present in the THtmlNode object and the hash values are the corresponding attribute values. If the HTML tag has never an attribute, the return value is NIL (e.g. *<layer>* or <!-- comment -->). If the HTML tag may have attributes, but no attribute is set, an empty hash is returned.

:isAttribute()

Checks if an attribute is present.

Syntax

```
:isAttribute( <cAttrName> ) --> lExists
```

Arguments

<cAttrName>

This is a character string holding the name of the attribute to query. It is case insensitive.

Description

Method :isAttribute() returns .T. (true) when the attribute with the name *<cAttrName>* is present in the THtmlNode object, otherwise .F. (false).

:setAttribute()

Sets an attribute and its value.

Syntax

```
:setAttribute( <cAttrName>, <cAttrValue> ) --> cValue
```

Arguments

`<cAttrName>`

This is a character string holding the name of the attribute to set. It is case insensitive.

`<cAttrValue>`

This is a character string holding the value of the attribute to set.

Description

Method :setAttribute() assigns the value *<cAttrValue>* to the attribute named *<cAttrName>*. If the attribute is not present, it is created. If the attribute is not valid for the HTML tag, a runtime error is raised. Use function THtmlIsValid() to verify if the attribute is valid.

Note: attributes that have only a name but no value receive always an empty string ("") as their value. This applies to the following HTML attributes:

Attributes with no value

Attribute	
checked	noresize
compact	noshade
declare	nowrap
defer	readonly
disabled	selected
ismap	showgrid
multiple	showgridx
nohref	showgridy

:setAttributes()

Sets multiple attributes and their values.

Syntax

```
:setAttributes( <cHtmlAttr> ) --> hHash
```

Arguments

`<cHtmlAttr>`

This is a character string holding the names and values of the attributes to set. Attribute names are case insensitive.

Description

Method :setAttributes() accepts a character string holding one or more attribute names and values and assigns these attributes to the THtmlNode object. An attribute name must be separated with an equal

sign from its value. Name/value pairs are separated with blank spaces. If an attribute value contains blank spaces, it must be embedded in quotes. For example:

```
oDoc  := THtmlDocument():new()
oNode := oDoc:head + "meta"
oNode:setAttributes( 'name=Generator content="xHarbour THtmlDocument class"' )
```

Html Node methods

:addNode()

Adds a child node.

Syntax

```
:addNode( <oTHtmlNode> ) --> oTHtmlNode
```

Arguments

<oTHtmlNode>

This is a THtmlNode object to add as child node to the HTML tree.

Description

Method :addNode() adds the passed THtmlNode object as child node to the *self!E object.*

:collect()

Returns all nodes including this node.

Syntax

```
:collect() --> aTHtmlNodes
```

Description

Method :collect() returns a one dimensional array holding all nodes of the THtmlNode object in sequential order. The first element of the array contains the *self* object.

:delete()

Removes an HTML node from the HTML tree.

Syntax

```
:delete() --> self
```

Description

Method :delete() removes the THtmlNode object executing the method from the HTML tree. The method removes this object and its sub-nodes from the tree.

:firstNode()

Returns the first node below this node.

Syntax

```
:firstNode( [<lRoot>] ) --> oTHtmlNode
```

Arguments

`<lRoot>`

This is a logical value. It defaults to .F. (false).

Description

Method :firstNode() returns the first node in the HTML tree below this THtmlNode object. If *<lRoot>* is set to .T. (true), the return value is the first node in the entire HTML tree.

:insertAfter()

Inserts an HTML node after the current node.

Syntax

```
:insertAfter( <oTHtmlNode> ) --> oTHtmlNode
```

Arguments

`<oTHtmlNode>`

This ia a THtmlNode object to insert into the HTML tree after the current HTML node.

Description

Method :insertAfter() inserts a THtmlNode object at the same nesting level into the HTML tree. The inserted HTML node appears immediately after the self object.

:insertBefore()

Inserts an HTML node before the current node.

Syntax

```
:insertBefore( <oTHtmlNode> ) --> oTHtmlNode
```

Arguments

`<oTHtmlNode>`

This ia a THtmlNode object to insert into the HTML tree before the current HTML node.

Description

Method :insertBefore() inserts a THtmlNode object at the same nesting level into the HTML tree. The inserted HTML node appears immediately before the self object. The return value is the previous HTML node.

:insertBelow()

Inserts a new HTML node below the current HTML node.

Syntax

```
:insertBelow( <oTHtmlNode> ) --> oTHtmlNode
```

Arguments

`<oTHtmlNode>`

This ia a THtmlNode object to insert into the HTML tree below the current HTML node.

Description

Method :insertBelow() adds a new HTML node to the HTML tree, one level below the current node.

:lastNode()

Returns the last node below this node.

Syntax

```
:lastNode() --> oTHtmlNode
```

Arguments

`<lRoot>`

This is a logical value. It defaults to .F. (false).

Description

Method :lastNode() returns the last node in the HTML branch below this THtmlNode object. If *<lRoot>* is set to .T. (true), the return value is the last node in the entire HTML tree.

:unlink()

Synonym for :delete()

Syntax

```
:unlink() --> self
```

Description

Method :unlink() is an alternative message for the :delete() method.

Html Tag methods

:isType()

Checks the content model type of an HTML tag.

Syntax

```
:isType( <nCM_TYPE> ) --> lIsType
```

Arguments

`<nCM_Type>`

This is a numeric value which is a combination of one of more #define constants listed in Thtml.ch. They begin with the prefix CM_ and describe the content model of an HTML tag.

Description

The method returns .T. (true) when the HTML tag is of the specified type *<nCM_TYpe>*, otherwise .F. (false) is returned.

Html Text methods

:getText()

Returns the textual content of an HTML node.

Syntax

```
:getText( [<cCRLF>] ) --> cHtmlText
```

Arguments

`<cCRLF>`

This parameter defaults to Chr(13)+Chr(10) so that textual content of nested HTML tags appears in separate text lines.

Description

Method :getText() retrieves the textual content of this HTML node and all subnodes as text string without HTML tags. The text of sub-nodes is separated with *<cCRLF>*. The expetion are table cells (*<td>* tags) whose textual content is separated by Chr(9) (TAB).

The returned string may need to be passed to HtmlToAnsi() or HtmlToOem() in order to convert HTML character entities to regular ANSI or OEM characters (e.g. "&" may appear in the returned string which must be translated to "&").

:toString()

Returns a HTML formatted character string.

Syntax

```
:toString() --> cHtmlString
```

Description

Method :toString() returns a character string including all HTML tags, attributes and textual content of this HTML node and its sub-nodes.

Overloaded operators

+ <cTagName>()

Opens a new HTML node

Syntax

```
+ <cTagName>
```

Arguments

`<cTagName>`

This is a character string holding a valid HTML tag name.

Description

The overloaded plus operator accepts a THtmlNode object as left operand, and a character string beginning with a valid HTML tag name as right operand. If the right operand does not start with a valid HTML tag name, or if the left operand is an HTML node that cannot have child nodes, a runtime error occurs. Otherwise, a new THtmlNode object is added as child node to the *self* object and returned. For example:

```
oDoc  := THtmlDocument():new()
oNode := oDoc:body + "h1"      // opening tag

? oNode:htmlTagName            // result: h1
oNode:text := "Heading"

oNode := oNode - "h1"          // closing tag
? oNode:htmlTagName            // result: body

? oDoc:body:toString()
** output
// <body>
// <h1>Heading</h1>
// </body>
```

The right operand may include HTML attributes valid for the HTML tag. In this case, a new HTML node with attributes is created and returned. For example:

```
oDoc  := THtmlDocument():new()

oNode := oDoc:body + "table border=0 cellspacing=0"
oNode := oNode     + "tr"
oNode := oNode     + "td nowrap"

oNode:text         := "Table cell content"

? oDoc:body:toString()
** output
// <body>
// <table border=0 cellspacing=0>
//  <tr>
//   <td nowrap>Table cell content
// </table>
// </body>
```

- <cTagName>()

Closes a new HTML node

Syntax

```
- <cTagName>
```

Arguments

<cTagName>

This is a character string holding the HTML tag name of the *self* object.

Description

The overloaded minus operator accepts a THtmlNode object as left operand, and a character string holding the HTML tag name of the THtmlNode as right operand. If the right operand is not the HTML tag name, a runtime error occurs. Otherwise, the HTML node is closed and the parent node is returned. For example:

```
oDoc       := THtmlDocument():new()
oNode      := oDoc:body + "p"
oNode:text := "This is "

oNode      := oNode + "i"
oNode:text := "italic "

oNode      := oNode + "b"
oNode:text := "and bold "

oNode      := oNode - "b"
oNode      := oNode - "i"

oNode:text := "Text."

? oDoc:body:toString()
** output
// <body>
// <p>This is <i>italic <b>and bold </b></i>Text.
// </body>
```

:<tagName> --> oTHtmlNode()

Returns the first HTML node with the specified tag name.

Syntax

```
:<tagName> --> oTHtmlNode
```

Arguments

<tagName>

This is the tag name of the HTML node to retrieve, or the attribute name of the attribute to retrieve or set.

Description

The overloaded send operator accepts as message the tag name of an HTML node or the name of an HTML attribute. If *<tagName>* is neither a HTML tag name nor a valid attribute name, a runtime error is raised.

HTML tag name

If *<tagName>* is a valid HTML tag name, the operator returns the first THtmlNode object having this tag name. If no THtmlNode object exists with this tag name, it is created and added as child node. For example:

```
oDoc       := THtmlDocument():new()
oTable     := oDoc:body:table
oRow       := oTable:tr
oCell      := oRow:td
oCell:text := "A"
oCell      := oRow + "td"
oCell:text := "B"

? oDoc:body:table:tr:toString()
** output
// <tr>
// <td>A
// <td>B
```

HTML attribute name

If *<tagName>* is a valid HTML sttribute name, the operator returns the value of this attribute. If the attribute is not set, the return value is NIL.

```
oDoc := THtmlDocument():new()
? oDoc:head:meta:name                   // result: NIL (attribute is not set)

oDoc:head:meta:name    := "Generator"
oDoc:head:meta:content := "THtmlDocument class"

? oDoc:head:meta:name                   // result: Generator

? oDoc:head:toString()
** output
// <head>
// <meta content="THtmlDocument class" name="Generator">
// </head>
```

:<tagName>S[()

Returns all HTML nodes with the specified tag name.

Syntax

```
:<tagName>S[(<nOrdinal>)] --> aTHtmlNodes
```

Arguments

`<tagName>S`

This is the tag name of the HTML nodes to retrieve followed by an "s" (the plural form of a tag name). They are returned in a one dimensional array.

`<tagName>S( <nOrdinal> )`

If *<tagName>*S is followed by a numeric value enclosed in parentheses, the *<nOrdinal>*th THtmlNode object having this tag name is returned.

Description

Sending a message consisting of the plural form of a HTML tag name (the tag name plus "s") instructs a THtmlNode object to collect all THtmlNode objects having this tag name in an array, which is returned.

Optionally, the ordinal position of the object with *<tagName>* can be specified. In this case, this THtmlNode object is returned. If *<nOrdinal>* is larger than the number of matching HTML nodes, the return value is NIL.

Info

See also:	AnsiToHtml(), THtmlCleanup(), THtmlInit(), THtmlDocument(), TIpClientHttp()
Category:	HTML functions, Object functions, xHarbour extensions
Header:	thtml.ch
Source:	tip\thtml.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// This example demonstrates operator overloading for
// creating an HTML document from a database.
```

```
PROCEDURE Main
   LOCAL oDoc, oNode, oTable, oRow, oCell
   LOCAL nDll, pApi
   CLS

   TRY
      USE Test.dbf
   CATCH
      ? "Error: Database not found TEST.DBF"
      QUIT
   END

   oDoc          := THtmlDocument():new()

   // Operator "+" creates a new node
   oNode         := oDoc:head + "meta"
   oNode:name    := "Generator"
   oNode:content := "THtmlDocument"

   // Operator ":" returns first "h1" from body (creates if not existent)
   oNode         := oDoc:body:h1
   oNode:text    := "My address book"

   // Operator "+" creates a new <p> node
   oNode         := oDoc:body + "p"

   // Operator "+" creates a new <font> node with attribute
   oNode         := oNode   + 'font size="5"'
   oNode:text    := "This is a "

   // Operator "+" creates a new <b> node
   oNode         := oNode   + "b"

   // Operator "+" creates a new <font> node with attribute
   oNode         := oNode   + "font color='blue'"
   oNode:text    := "sample "

   // Operator "-" closes 2nd <font>, result is <b> node
   oNode         := oNode   - "font"

   // Operator "-" closes <b> node, result is 1st <font> node
   oNode         := oNode   - "b"

   oNode:text    := "database!"

   // Operator "-" closes 1st <font> node, result is <p> node
   oNode         := oNode   - "font"

   oNode         := oNode   + "hr"

   // Operator ":" returns first "table" from body (creates if not existent)
   oTable        := oDoc:body:table
   oTable:attr   := 'border="0" width="100%" cellspacing="0" cellpadding="0"'

   oRow          := oTable  + 'tr bgcolor="lightcyan"'
   FOR i:=1 TO FCount()
       oCell     := oRow + "th"
       oCell:text:= FieldName(i)
   NEXT

   FOR i:=1 TO 10
      oRow         := oTable + "tr"
```

```
      oRow:bgColor := IIf( Recno() % 2 == 0, "lightgrey", "white" )

      FOR j:=1 TO FCount()
         oCell      := oRow + "td"
         oCell:text := FieldGet(j)
      NEXT

      SKIP
   NEXT

   oNode := oDoc:body  + "hr"
   oNode := oDoc:body  + "p"

   oNode:text := "10 records from database " + Alias() + ".dbf"

   DbCloseArea()

   IF oDoc:writeFile( "Address.html" )
      ? "File created: Address.html"
   ELSE
      ? "Error: ", FError()
   ENDIF

   WAIT
   ? HtmlToOem( oDoc:body:getText() )

   nDll := DllLoad( "Shell32.dll" )
   pApi := GetProcAddress( nDll, "ShellExecute" )

   CallDll( pApi, 0, "open", "Address.html", NIL, "", 1 )

   DllUnload( nDll )
RETURN
```

TIpClient()

Abstract class for internet communication.

```
Sub-classes:

TIpClientFtp()
TIpClientHttp()
TIpClientPop()
TIpClientSmtp()
```

Description

TIpClient() is an abstract class providing instance variables and methods for four sub-classes. These sub-classes are used for internet communication with the FTP, HTTP, POP and SMTP protocols. TIpClient() objects are never instantiated directly. The class defines generic instance variables and methods required for all four internet protocols.

Instance variables

:cReply

Reply message of server.

Data type: C
Default: NIL

Description

The instance variable :cReply contains a character string holding the server's reply message of the last internet communication.

:exGauge

Codeblock or function pointer for progress information.

Data type: C|P
Default: NIL

Description

The instance variable :exGauge can be assigned a code block or a function pointer. It is executed in methods :readToFile() or :writeFromFile() and receives three parameters which can be used to display progress information. The parameters are 1) numeric number of bytes already processed, 2) numeric number total bytes to process and 3) the *self* object

:lTrace

Logical value indicating trace logging.

Data type: L
Default: .F.

Description

When :lTrace is set to .T. (true), the *self* object logs information during internet communications to a file. The log file name depends on the (sub) class.

:nConnTimeout

Numeric timeout value in milliseconds.

Data type: N
Default: NIL

Description

The instance variable :nConnTimeOut holds a numeric value specifying the connection timout period in milliseconds. Sub-classes of the TIpClient() class define their own connection timeout values.

:nDefaultPort

Numeric port number.

Data type: N
Default: NIL

Description

The instance variable :nDefaultPort holds a numeric value specifying the port number to use for the internet connection. Sub-classes of the TIpClient() class define their own post number.

Note: if the TUrl() object stored in :oUrl has its own port number, it overrides the port number :nDefaultPort.

:nLastRead

Number of bytes read.

Data type: N
Default: NIL

Description

The instance variable :nLastRead contains a numeric value specifying the number of bytes transmitted with the last read method.

:nLastWrite

Number of bytes written.

Data type: N
Default: NIL

Description

The instance variable :nLastWrite contains a numeric value specifying the number of bytes transmitted with the last write method.

:oUrl

TUrl object.

Data type: TIpUrl()
Default: NIL

Description

The instance variable :oUrl contains a TUrl() object maintaining the data of the URL used for the internet connection.

Communication

:close()

Closes the internet connection.

Syntax

```
:close() --> nErrorCode
```

Description

Method :close() closes an open internet connection and returns a numeric error code. If the operation is successful, the return value is zero.

:commit()

Commits data.

Syntax

```
:commit() --> lSuccess
```

Description

Method :commit() commits data to the server when required by the internet protocol.

:open()

Opens the internet connection.

Syntax

```
:open( [<cUrl>] ) --> lSuccess
```

Arguments

<cUrl>

Optionally, a new URL can be specified as a character string. If the parameter is omitted, the URL passed to the :new() method is used.

Description

Method :open() establishes a connection to the internet server specified with the URL. The function returns .T. (true) when the connection is successfully established, otherwise .F. (false) is returned. When the connection is opened, data can be transmitted using read and write methods.

:read()

Reads data from the internet connection.

Syntax

```
:read( <nBytes> ) --> cData
```

Arguments

`<nBytes>`

This is a numeric value specifying the number of bytes to read from the internet connection.

Description

Method :read() transmits the specified number of bytes from the server to the local station and returns the data as a character string. Note that reading data requires a successful call to the :open() method for establishing the internet connection.

:reset()

Resets internal state variables.

Syntax

```
:reset() --> lSuccess
```

Description

Method :reset() resets internal state variables used during data transmission. The method should be called before a read or write method is executed.

:write()

Writes data to the internet connection.

Syntax

```
:write( <cData>, [<nBytes>], [<lCommit>] ) --> nBytesWritten
```

Arguments

`<cData>`

This is a character string holding the data to transmit to the server.

`<nBytes>`

Optionally, the number of bytes to transmit can be specified as a numeric value. *<nBytes>* defaults to Len(*<cData>*).

`<lCommit>`

If this parameter is set to .T. (true), method :commit() is called when all data is transferred. The default is .F. (false).

Description

Method :write() transmits the specified character string *<cData>* from the local station to the server and returns the number of bytes transmitted as a numeric value. Note that writing data requires a successful call to the :open() method for establishing the internet connection.

Error info

:lastErrorCode()

Returns the last error code.

Syntax

```
:lastErrorCode() --> nErrorCode
```

Description

Method :lastErrorCode() returns the numeric error code of the last read or write method. When no error occurred, the return value is zero.

:lastErrorMessage()

Returns the last error message.

Syntax

```
:lastErrorMessage() --> cErrorMessage
```

Description

Method :lastErrorMessage() returns a character string describing the error of the last read or write method. When no error occurred, the return value is a null string ("").

File handling

:readToFile()

Reads data from the internet connection to a local file.

Syntax

```
:readToFile( <cFileName>, [<nFileAttr>], <nBytes> ) --> lSuccess
```

Arguments

`<cFileName>`

This is a character string holding the name of the file to create. It must include path and file extension. If the path is omitted from <*cFileName*>, the file is created in the current directory.

`<nFileAttr>`

A numeric value specifying one or more attributes associated with the new file. #define constants from the FILEIO.CH file can be used for <*nFileAttr*> as listed in the table below:

Attributes for binary file creation

Constant	Value	Attribute	Description
FC_NORMAL *)	0	Normal	Creates a normal read/write file
FC_READONLY	1	Read-only	Creates a read-only file
FC_HIDDEN	2	Hidden	Creates a hidden file
FC_SYSTEM	4	System	Creates a system file

**) default attribute*

`<nBytes>`

The number of bytes to retrieve from the server must be specified as a numeric value.

Description

Method :readToFile() transmits the specified number of bytes from the server to the local station and writes them into the file <*cFileName*>. Note that reading data requires a successful call to the :open() method for establishing the internet connection.

The method returns a logical value indicating success of the operation.

:writeFromFile()

Writes data from a local file to the internet connection.

Syntax

```
:writeFromFile( <cFileName> ) --> lSuccess
```

Arguments

`<cFileName>`

This is a character string holding the name of the file to open on the local station. It must include path and file extension. If the path is omitted from <*cFileName*>, the file is searched in the current directory.

Description

Method :writeFromFile() transmits the contents of an entire file <*cFileName*> from the local station to the server. Note that writing data requires a successful call to the :open() method for establishing the internet connection.

The method returns a logical value indicating success of the operation.

Info

See also: TIpClientFtp(), TIpClientHttp(), TIpClientPop(), TIpClientSmtp(), TUrl()
Category: Internet functions, Object functions, xHarbour extensions
Source: tip\client.prg
LIB: lib\xhb.lib
DLL: dll\xhb.dll

TIpClientFtp()

Creates a new TIpClientFtp object.

Syntax

```
TIpClientFtp():new( <cUrl>, [<lTrace>] ) --> oTIpClientFtp
```

Arguments

<cUrl>

This is a character string holding the URL of the FTP server used to upload or download files to/from an FTP server. It must be coded in this form:

```
ftp://<userID>:<password>@<server.com>
```

Alternatively, a TUrl() object can be passed that is initialized with the FTP server URL.

<lTrace>

This parameter defaults to .F. (false). When .T. (true) is passed, the communication with a mail server is logged into the file Ftp<*nn*>.log, where <*nn*> is the number of the log file storing information of the internet communictaion with this object.

Return

The function returns a new TIpClientFtp object and method :new() initializes the object.

Description

Objects of the TIpClientFtp() class inherit from the generic internet client class TIpClient(). TIpClientFtp objects are used to communicate with an FTP server. They use the File Transfer Protocol (FTP, RFC0821.TXT) for exchanging files between a local and a remote station. The address of the FTP server must be provided as a URL string with method :new(). The URL is maintained by a TUrl() object, which is stored in the :oUrl instance variable.

The internet connection to the FTP server must be established with the :open() method.

Once the internet connection is established (opened), files can be uploaded or downloaded using the :uploadFile() or :uploadFile() methods.

When all files are sent or received, the internet connection must be closed with the :close() method.

Instance variables

:nConnTimeout

Numeric timeout value in milliseconds.

Data type: N
Default: 3000

Description

The instance variable :nConnTimeOut holds a numeric value specifying the connection timout period in milliseconds. The default is 3 seconds (3000 milliseconds).

:nDefaultPort

Numeric port number.

Data type: N
Default: 21

Description

The instance variable :nDefaultPort holds a numeric value specifying the port number to use for the internet connection. The port number for the FTP protocol is 21.

Directory methods

:cwd()

Changes the working directory on the FTP server.

Syntax

```
:cwd( <cPath> ) --> lSuccess
```

Arguments

<cPath>

This is a character string holding the name of the new working directory.

Description

Method :cwd() changes the current directory on the FTP server and returns a logical value indicating success of the operation.

:mkd()

Creates a directory on the FTP server.

Syntax

```
:mkd( <cPath> ) --> lSuccess
```

Arguments

<cPath>

This is a character string holding the name of the directory to create.

Description

Method :mkd() creates a new directory on the FTP server and returns a logical value indicating success of the operation.

:pwd()

Queries the current directory on the FTP server.

Syntax

```
:pwd() --> lSuccess
```

Description

Method :pwd() sends the PWD command to the Ftp server and returns a logical value indicating success of the operation. If successful, the current directory is stored as character string in the :cReply instance variable.

:rmd()

Removes a directory on the FTP server.

Syntax

```
:rmd( <cPath> ) --> lSuccess
```

Arguments

```
<cPath>
```

This is a character string holding the name of the directory to remove.

Description

Method :rmd() deletes a directory on the FTP server and returns a logical value indicating success of the operation.

File methods

:dele()

Deletes a file on the FTP server

Syntax

```
:dele( <cFileName> ) --> lSuccess
```

Arguments

```
<cFileName>
```

This is a character string holding the name of the file to delete.

Description

Method :dele() deletes a file on the FTP server and returns a logical value indicating success of the operation.

:list()

Retrieves file information from the FTP server as character string.

Syntax

```
:list( [<cFileSpec>] ) --> cFileList
```

Arguments

```
<cFileSpec>
```

This is a character string holding the drive file specification to retrieve information for. It defaults to an empty string ("") which retrieves all available file information from the FTP server.

Description

Method :list() sends the LIST command along with an optional file specification to the FTP server and returns the file information as a charcter string. Information of individual files is delimited with INetCrlf().

Note: method :listFiles() can be used to retrieve the file information in form of an array.

:listFiles()

Retrieves file information from the FTP server as array.

Syntax

```
:listFiles( [<cFileSpec>] ) --> aFiles
```

Arguments

This is a character string holding the drive file specification to retrieve information for. It defaults to an empty string ("") which retrieves all available file information from the FTP server.

Description

Method :listFiles() sends the LIST command along with an optional file specification to the FTP server and returns the file information as a two-dimensional array. The following #define constants from the Directry.ch file can be used to access individual columns of the file information array:

Constants for the file information array

Constant	Position	Description	Data type
F_NAME	1	File name	Character
F_SIZE	2	File size in bytes	Numeric
F_DATE	3	Creation date	Date
F_TIME	4	Creation time	Character
F_ATTR	5	File permissions (attributes)	Character
F_LEN+1	6	Number of links to file	Numeric
F_LEN+2	7	Owner name	Character
F_LEN+3	8	Group name	Character

The column F_ATTR contains a character string of 10 bytes. It encodes the file type in its first byte and file permissions for the owner, group and public in the following nine bytes.

File types (1st byte of F_ATTR)

Character	Description
-	Regular file
b	Block special file
c	Character special file
d	Directory
l	Symbolic link
n	Network file
p	FIFO
s	Socket

The next nine characters are in three groups of three bytes. They describe the permissions on the file. The first group of three describes owner permissions, the second describes group permissions, and the

third describes other (or public) permissions. Because Windows systems do not support group and other permissions, these are copies of the owner's permissions. Characters that may appear are:

File permissions (byte 2-10 of F_ATTR)

Character	Description
r	Permission to read file
w	Permission to write to file
x	Permission to execute file
t	Temporary file

:downloadFile()

Downloads a file.

Syntax

```
:downloadFile( <cLocalFile>, [<cRemoteFile>] ) --> lSuccess
```

Arguments

<cLocalFile>

This is a character string holding the name of the file on the local station where downloaded data is written to. It must include path and file extension. If the path is omitted from *<cLocalFile>*, the file is created in the current directory.

<cRemoteFile>

This is an optional character string holding the name of the file on the FTP server. It must be specified without path information. If omitted, the file name from *<cLocalFile>* is used.

Description

Method :downloadFile() transfers a file from the FTP server to the local station and returns a logical value indicating success of the operation. The file must be located in the current directory of the FTP server. Use method :mget() to download multiple files with one method call.

:mget()

Downloads multiple files from the FTP server.

Syntax

```
:mget( <cFileSpec>, <cLocalDir> ) --> cFileList
```

Arguments

<cFileSpec>

This is a character string holding the specification of files to retrieve from the FTP server.

<cLocalDir>

This is a character string holding the name of th directory on the local station where the files should be stored.

Description

Method :mget() downloads all files meeting the file specification *<cFileSpec>* from the FTP server and stores them in the directory *<cLocalDir>*. This directory must exist on the local station. If a file exists on the local station having the same name as on the FTP server, the local file is overwritten. The

method returns a character string including the names of all downloaded files, separated with INetCrlf().

:mput()

Uploads multiple files to the FTP server.

Syntax

```
:mput( <cFileSpec>, [<cAttr>] ) --> nFileCount
```

Arguments

```
<cFileSpec>
```

This is a character string holding the drive, directory and file specification to upload.

```
<cAttr>
```

Optionally, a character string holding file attributes can be specified. One or more characters of the table below can be included in *<cAttr>*.

Attributes for :mput()

Attribute	Meaning
H	Include hidden files
S	Include system files

Description

Method :mput() uploads all files meeting the file specification *<cFileSpec>* to the FTP server and stores them in the current directory of the server. If a file exists on the FTP server having the same name as on the loacl station, the remote file is overwritten. The method returns a character string including the names of all uploaded files, separated with INetCrlf().

:rename()

Renames a file on the FTP server.

Syntax

```
:rename( <cFrom>, <cTo> )  --> lSuccess
```

Arguments

```
<cFrom>
```

This is a character string holding the name of the file to rename.

```
<cTo>
```

This is a character string with the new file name including file extension.

Description

Method :rename() renames a file on the FTP server and returns a logical value indicating success of the operation.

:uploadFile()

Downloads a file.

Syntax

```
:uploadFile( <cLocalFile>, [<cRemoteFile>] ) --> lSuccess
```

Arguments

`<cLocalFile>`

This is a character string holding the name of the file on the local station to transfer to the FTP server. It must include path and file extension. If the path is omitted from <*cLocalName*>, the file is searched in the current directory.

`<cRemoteFile>`

This is an optional character string holding the name of the file to create on the FTP server. It must be specified without path information. If omitted, the file name from <*cLocalFile*> is used.

Description

Method :uploadFile() transfers a file from the local station to the FTP server and returns a logical value indicating success of the operation. The file is stored in the current directory of the FTP server. Use method :mput() to upload multiple files with one method call.

Info

See also:	TIpClient(), TIpClientHttp(), TUrl()
Category:	Internet functions, Object functions, xHarbour extensions
Source:	tip\ftpcln.prg
LIB:	lib\xhb.lib
DLL:	dll\xhb.dll

Example

```
// The example demonstrates file transfers using an USB memory stick connected
// to a WLAN server. The internet address of the WLAN server is 192.168.178.1.

   #include "Directry.ch"

   PROCEDURE Main
      LOCAL oFtp, cRoot
      LOCAL cUSBStick := "ftp://anonymous:guest@192.168.178.1"

      CLS

      oFtp := TIpClientFtp():new( cUSBStick )

      // function for displaying progress bar
      oFtp:exGauge := ( @FtpProgress() )

      IF .NOT. oFtp:open()
         ? oFtp:lastErrorMessage()
         QUIT
      ENDIF

      aFiles := oFtp:listFiles()
      cRoot  := aFiles[1,1]

      ? oFtp:cwd( cRoot    ), oFtp:cReply
      ? oFtp:mkd( "newdir" ), oFtp:cReply
```

```
   ? oFtp:cwd( "newdir" ), oFtp:cReply

   ? oFtp:uploadFile( "testftp.prg" )  , oFtp:cReply
   ? oFtp:uploadFile( "testftp.exe" )  , oFtp:cReply

   oFtp:cwd( ".." )
   Dirlist( oFtp, aFiles, "" )
   oFtp:cwd( cRoot + "/newdir"  )

   ? oFtp:downloadFile( "testftp.prg" ), oFtp:cReply

   ? oFtp:dele( "testftp.exe" ), oFtp:cReply
   ? oFtp:dele( "testftp.prg" ), oFtp:cReply

   ? oFtp:cwd( ".." ), oFtp:cReply

   ? oFtp:rmd( "newdir" ), oFtp:cReply

   oFtp:close()

RETURN

// Recurses through all directories and lists files
FUNCTION DirList( oFtp, aFiles, cIndent )
   LOCAL aFile

   FOR EACH aFile IN aFiles
      IF aFile[F_NAME] == "." .OR. aFile[F_NAME] == ".."
         LOOP
      ENDIF

      ? cIndent, aFile[F_NAME]

      IF aFile[F_ATTR][1] == "d"
         oFtp:cwd( aFile[F_NAME] )
         DirList( oFtp, oFtp:listFiles(), cIndent+"   " )
         oFtp:cwd( ".." )
      ENDIF
   NEXT
RETURN .T.

// Displays a progress bar during file upload/download
FUNCTION FtpProgress( nSent, nTotal, oFtp )
   LOCAL cProgress
   LOCAL nRow := Row(), nCol := Col()

   cProgress := Replicate( Chr(178), Int( MaxCol()*nSent/nTotal ) )
   SetPos( MaxRow()-1, 0 )

   ?? oFtp:oUrl:cFile
   ?  PadR( cProgress, MaxCol() )

   SetPos( nRow, nCol )
RETURN .T.
```

TIpClientHttp()

Creates a new TIpClientHttp object.

Syntax

```
TIpClientHttp():new( <cUrl> ) --> oTIpClientHttp
```

Arguments

<cUrl>

This is a character string holding the URL of the HTTP server used to access files in the World Wide Web (WWW). It must be coded in this form:

```
http://[<userID>:<password>@]<www.server.com>
```

Alternatively, a TUrl() object can be passed that is initialized with the HTTP server URL.

Return

The function returns a new TIpClientHttp object and method :new() initializes the object.

Description

Objects of the TIpClientHttp() class inherit from the generic internet client class TIpClient(). TIpClientHttp objects are used to communicate with an HTTP server. They use the Hyper Text Transfer Protocol (HTTP, RFC2616.TXT) for exchanging data between a local and a remote station on the World Wide Web (WWW). The address of the HTTP server must be provided as a URL string with method :new(). The URL is maintained by a TUrl() object, which is stored in the :oUrl instance variable.

The internet connection to the HTTP server must be established with the :open() method.

Once the internet connection is established (opened), data can be retrieved using the :readAll() method.

When all data is received, the internet connection must be closed with the :close() method.

Instance variables

:nConnTimeout

Numeric timeout value in milliseconds.

Data type: N
Default: 5000

Description

The instance variable :nConnTimeOut holds a numeric value specifying the connection timout period in milliseconds. The default is 5 seconds (5000 milliseconds).

:nDefaultPort

Numeric port number.

Data type: N
Default: 80

Description

The instance variable :nDefaultPort holds a numeric value specifying the port number to use for the internet connection. The default port number for the HTTP protocol is 80.

Methods

:readAll()

Retrieves all data from a URL.

Syntax

```
:readAll() --> cURLContent
```

Description

Method :readAll() retrieves all data from a URL and returns it as a character string. Normally, a URL points to an HTML file stored on an HTTP server. The method returns the contents of the file, which may need to be analyzed for retrieving linked files, such as image files.

Info

See also: THtmlDocument(), TIpClient(), TIpClientFtp(), TUrl()
Category: HTML functions, Internet functions, Object functions, xHarbour extensions
Source: tip\httpcln.prg
LIB: lib\xhb.lib
DLL: dll\xhb.dll

Example

```
// The example loads the Google search page and enters "xHarbour" as
// query data. The response from Google for the query is stored in a local
// HTML file.

    PROCEDURE Main
       LOCAL oHttp, cHtml, hQuery

       oHttp:= TIpClientHttp():new( "http://www.google.de/search" )

       // build the Google query
       hQUery := Hash()
       hSetCaseMatch( hQuery, .F. )

       hQuery["q"]    := "xHarbour"
       hQuery["hl"]   := "en"
       hQuery["btnG"] := "Google+Search"

       // add query data to the TUrl object
       oHttp:oUrl:addGetForm( hQuery )

       // Connect to the HTTP server
       IF oHttp:open()
```

```
      // downlowad the Google response
      cHtml   := oHttp:readAll()
      Memowrit( "Google_xHarbour.html", cHtml )

      oHttp:close()
      ? Len(cHtml), "bytes received "
      ?? "and written to file Google_xHarbour.html"
   ELSE
      ? "Connection error:", oHttp:lastErrorMessage()
   ENDIF
RETURN
```

TIpClientPop()

Creates a new TIpClientPop object.

Syntax

```
TIpClientPop():new( <cUrl>, [<lTrace>] ) --> oTIpClientPop
```

Arguments

`<cUrl>`

This is a character string holding the URL of the mail server to retrieve eMails from. It must be coded in this form:

```
pop3://<mailAccount>:<password>@<mail.server.com>
```

Alternatively, a TUrl() object can be passed that is initialized with the mail server URL.

`<lTrace>`

This parameter defaults to .F. (false). When .T. (true) is passed, the communication with a mail server is logged into the file Pop3*<nn>*.log, where *<nn>* is the number of the log file storing information of the internet communictaion with this object.

Return

The function returns a new TIpClientPop object and method :new() initializes the object.

Description

Objects of the TIpClientPop() class inherit from the generic internet client class TIpClient(). TIpClientPop objects are used to communicate with a POP3 mail server. They use the Post Office Protocol (POP) for downloading, or reading, eMails. The address of the mail server must be provided with a URL string with method :new(). The URL is maintained by a TUrl() object, which is stored in the :oUrl instance variable.

The internet connection to the mail server must be established with the :open() method, after which a list of pending eMails can be queried from the mail server with the :list() method.

Individual eMails can then be retrieved with :retrieve(). This method returns the entire contents of a single eMail as a character string which can be decoded using a TIpMail() object and its :fromString() method.

Once an eMail is retrieved and stored locally, it can be deleted from the mail server using the :delete() method. If a mail is retrieved but not deleted, it remains on the mail server and can be retrieved again.

When all eMails are read, the internet connection must be closed with the :close() method.

Instance variables

:nConnTimeout

Numeric timeout value in milliseconds.

Data type: N
Default: 10000

Description

The instance variable :nConnTimeOut holds a numeric value specifying the connection timout period in milliseconds. The default is 10 seconds (10000 milliseconds).

:nDefaultPort

Numeric port number.

Data type: N
Default: 110

Description

The instance variable :nDefaultPort holds a numeric value specifying the port number to use for the internet connection. The port number for the POP protocol is 110.

Methods

:delete()

Deletes an eMail on the mail server.

Syntax

```
:delete( <nMail> ) --> lSuccess
```

Arguments

`<nMail>`

This is a numeric value specifying the ordinal number of the mail to delete from the mail server.

Description

Method :delete() instructs the mail server to remove the mail with the ordinal number *<nMail>* from the storage. When a mail is deleted on the server, it cannot be retrieved again.

The method returns a logical value indicating success of the operation.

:list()

Lists eMails pending on the mail server.

Syntax

```
:list() --> cMailIDs
```

Description

Method :list() returns a character string including the ordinal numbers and unique IDs of mail messages pending on the mail server for download. Data for individual mail messages are separated with INetCRLF(). If no messages are available on the server, an empty string is returned.

:noOp()

Can be called repeatedly to keep-alive the connection.

Syntax

```
:noOp() --> lSuccess
```

Description

Method :noOp() can be used to keep the connection to the mail server alive. Alternatively, the :connTimeout value can be increased.

The method returns a logical value indicating success of sending a NOOP instruction to the mail server.

:retrieve()

Retrieves an eMail from the mail server

Syntax

```
:retrieve( <nMail>, [<nBytes>] ) --> cMail
```

Arguments

`<nMail>`

This is a numeric value specifying the ordinal number of the mail to retrieve from the mail server.

`<nBytes>`

Optionally, a numeric value can be passed specifying the number of bytes to retrieve from the mail. If not specified, the entiere mail message s retrieved.

Description

Method :retrieve() reads the mail with number *<nMail>* from the mail server and downloads it to the local station. The mail message is returned as a character string holding the raw data of the entire mail messaged. The message can be decomposed into its individual components by passing the returned string to oTIpMail():fromString().

:stat()

Retrieves a status message from the mail server.

Syntax

```
:stat() --> cStatusMsg
```

Description

Method :stat() sends a STAT command to the mail server and returns the server response as a character string. It is mainly of informational purpose.

:top()

Retrieves the headers of an eMail (no body).

Syntax

```
:top( <nMail> ) --> cMailHeaders
```

Arguments

`<nMail>`

This is a numeric value specifying the ordinal number of the mail to get header information from.

Description

Methop :top() reads the headers of the mail with number *<nMail>* from the mail server and downloads them. The mail body and file attachments, if any, are not downloaded. This allows for a quick lookup for mail subject and sender. The headers can be decomposed by passing the returned string to

oTIpMail():fromString(), and querying individual mail header fields with method oTIpMail():getFieldPart() and/or oTIpMail():getFieldOption()

:uidl()

Returns unique ID of an eMail.

Syntax

```
:uidl( <nMail> ) --> cMailID
```

Arguments

<nMail>

This is a numeric value specifying the ordinal number of the mail to get the unique ID from.

Description

Method :uidl() returns the unique message ID of the mail with number *<nMail>* from the mail server. It is of informational purpose only since mail messages are retrieved from a server by their ordinal number.

Info

See also:	TIpClient(), TIpMail(), TUrl()
Category:	Internet functions, Object functions, xHarbour extensions
Source:	tip\popcln.prg
LIB:	lib\xhb.lib
DLL:	dll\xhb.dll

Example

```
// The example outlines the steps required for retrieving all
// eMails from a POP mail server and how to decompose
// incoming mail messages.

   PROCEDURE Main
      LOCAL oPop, oPart, aParts, oTIpMail, aEmails, i

      oPop := TIpClientPop():new( "pop://mailaccount:password@pop.server.com" )

      IF .NOT. oPop:open()
         ? "Connection error:", oPop:lastErrorMessage()
         QUIT
      ELSE
         aEMails := oPop:retrieveAll()
         oPop:close()
      ENDIF

      FOR i:=1 TO Len( aEMails )
         oTIpMail := aEmails[i]
         ? oTIpMail:getFieldPart( "From" )
         ? oTIpMail:getFieldPart( "Subject" )

         IF oTIpMail:isMultiPart()
            // Retrieve all parts of a multipart message
            aParts := oTIpMail:getMultiParts()

            FOR EACH oPart IN aParts
               IF .NOT. Empty( oPart:getFileName() )
                  // This is a file attachment. Store it in the TMP folder.
```

```
               IF oPart:detachFile( "C:\tmp\" )
                  ? "File written: C:\tmp\" + oPart:getFileName()
               ENDIF
            ELSE
               ? oPart:getBody()
            ENDIF
         NEXT
      ELSE
         // simple mail message
         ? oTIpMail:getBody()
      ENDIF
   NEXT

RETURN
```

TIpClientSmtp()

Creates a new TIpClientSmtp object.

Syntax

```
TIpClientSmtp():new( <cUrl>, [<lTrace>] ) --> oTIpClientSmtp
```

Arguments

`<cUrl>`

This is a character string holding the URL of the mail server used to send eMails. It must be coded in this form:

```
smtp://<mailAccount>:<password>@<mail.server.com>
```

Alternatively, a TUrl() object can be passed that is initialized with the SMTP mail server URL.

`<lTrace>`

This parameter defaults to .F. (false). When .T. (true) is passed, the communication with a mail server is logged into the file Sendmail<*nn*>.log, where <*nn*> is the number of the log file storing information of the internet communictaion with this object.

Return

The function returns a new TIpClientSmtp object and method :new() initializes the object.

Description

Objects of the TIpClientSmtp() class inherit from the generic internet client class TIpClient(). TIpClientSmtp objects are used to communicate with an SMTP mail server. They use the Simple Mail Transfer Protocol (SMTP, RFC0821.TXT) for sending, or writing, eMails. The address of the mail server must be provided as a URL string with method :new(). The URL is maintained by a TUrl() object, which is stored in the :oUrl instance variable.

The internet connection to the mail server must be established with the :open() method.

Once the internet connection is established (opened), eMails can be sent using the :sendMail() method, which accepts an eMail in form of a TIpMail() object.

When all eMails are sent, the internet connection must be closed with the :close() method.

Note: methods of the TIpClientSmtp class are divided into "High level" and "Low level" methods. "High level" methods are designed to process TIpMail() objects, while "Low level" methods communicate with an SMTP server directly using commands of the SMTP protocol.

Instance variables

:nConnTimeout

Numeric timeout value in milliseconds.

Data type: N
Default: 5000

Description

The instance variable :nConnTimeOut holds a numeric value specifying the connection timout period in milliseconds. The default is 5 seconds (5000 milliseconds).

:nDefaultPort

Numeric port number.

Data type: N
Default: 25

Description

The instance variable :nDefaultPort holds a numeric value specifying the port number to use for the internet connection. The port number for the SMTP protocol is 25.

High level methods

:sendMail()

Sends an eMail.

Syntax

```
:sendMail( <oTIpMail> ) --> lSuccess
```

Arguments

`<oTIpMail>`

This must be a TIpMail() object holding all data of the eMail to send.

Description

Method :sendMail() provides the most convenient way of sending an eMail. It accepts a TIpMail() object holding all data of an eMail. :sendMail() authenticates a user from the user and password information held in the TURl() object stored in :oURL Note that sending a mail message requires a successful call to the :open() method for establishing the internet connection.

Low level methods

:auth()

Sends the AUTH LOGIN command to the SMTP server.

Syntax

```
:auth( <cUserID>, <cPassword> ) --> lSuccess
```

Arguments

`<cUserID>`

This is a character string holding the user ID for logging into the SMTP server.

`<Password>`

This is a character string holding the password for logging into the SMTP server.

Description

Metod :auth() logs into the the SMTP server and authenticates the user with the AUTH LOGIN command. The return value is .T. (true) when the log-in procedure is successful. Otherwise, .F. (false)

is returned. Note that the log-in procedure requires a successful call to the :open() method for establishing the internet connection.

:authPlain()

Sends the AUTH PLAIN command to the SMTP server.

Syntax

```
:authPlain( <cUserID>, <cPassword> ) --> lSuccess
```

Arguments

`<cUserID>`

This is a character string holding the user ID for logging into the SMTP server.

`<Password>`

This is a character string holding the password for logging into the SMTP server.

Description

Metod :auth() logs into the the SMTP server and authenticates the user with the AUTH PLAIN command. The return value is .T. (true) when the log-in procedure is successful. Otherwise, .F. (false) is returned. Note that the log-in procedure requires a successful call to the :open() method for establishing the internet connection.

:data()

Sends the DATA command to the SMTP server.

Syntax

```
:data( <cData> ) --> lSuccess
```

Arguments

`<cData>`

This is a character string holding the mail message to send.

Description

Method :data() sends the DATA command along with the mail message to the SMTP server and returns a logical value indicating success of the operation. Prior to :data(), the methods :mail() and the methods :mail() and :rcpt() must be called in order to specify sender and recipient(s) of the mail message.

:mail()

Sends the MAIL command to the SMTP server.

Syntax

```
:mail( <cFrom> ) --> lSuccess
```

Arguments

`<cFrom>`

This is a character string holding the sender's eMail address.

Description

Method :mail() sends the MAIL command along with the sender's mail address to the SMTP server and returns a logical value indicating success of the operation. :mail() is the first method initiating an eMail transfer, it must be followed by :rcpt() and :data().

:quit()

Sends the QUIT command to the SMTP server.

Syntax

```
:quit() --> lSuccess
```

Description

Method :quit() sends the QUIT command to the SMTP server which indicates the end of eMail transmission. It is implicitely called in the :close() method.

:rcpt()

Sends the RCPT command to the SMTP server.

Syntax

```
:rcpt( <cRecipient> ) --> lSuccess
```

Arguments

<cRecipient>

This is a character string holding the recipient's eMail address.

Description

Method :rcpt() sends the RCPT command along with the recipient's mail address to the SMTP server and returns a logical value indicating success of the operation. :rcpt() is the second method in an eMail transfer. It can be called repeatedly for specifying multiple recipients and must be followed by the :data() method.

Info

See also:	TIpClientFtp(), TIpClientHttp(), TIpClientPop(), TIpMail(), TUrl()
Category:	Internet functions, Object functions, xHarbour extensions
Source:	tip\smtpcln.prg
LIB:	lib\xhb.lib
DLL:	dll\xhb.dll

Example

```
// The example outlines the steps and data required to send an eMail
// including file attachment to an SMTP mail server (it forwards the eMail)

   PROCEDURE Main
      LOCAL oSmtp, oEMail
      LOCAL cSmtpUrl
      LOCAL cSubject, cFrom, cTo, cBody, cFile

      // preparing data for eMail
      cSmtpUrl := "smtp://mailaccount:password@smtp.server.com"
      cSubject := "Testing eMail"
      cFrom    := "MyName@Mail.server.com"
```

```
   cTo      := "YourName@another.server.com"
   cFile    := "File_Attachment.zip"
   cBody    := "This is a test mail sent at: " + DtoC(date()) + " " + Time()

   // preparing eMail object
   oEMail   := TIpMail():new()
   oEMail:setHeader( cSubject, cFrom, cTo )
   oEMail:setBody( cBody )
   oEMail:attachFile( cFile )

   // preparing SMTP object
   oSmtp := TIpClientSmtp():new( cSmtpUrl )

   // sending data via internet connection
   IF oSmtp:open()
      oSmtp:sendMail( oEMail )
      oSmtp:close()
      ? "Mail sent"
   ELSE
      ? "Error:", oSmtp:lastErrorMessage()
   ENDIF
RETURN
```

TIpMail()

Creates a new TIpMail object.

Syntax

```
TIpMail():new( [<cMailBody>], [<cEncode>] ) --> oTIpMail
```

Arguments

`<cMailBody>`

This is an optional character string holding the mail body of an eMail. The mail body is the text transmitted with the eMail.

`<cEncode>`

This is an optional character string holding the name of the encoding method to use for the body of the eMail. The following encoding methods are supported:

Encoding methods for eMail bodies

Name	Description
as-is *)	Not encoded (unchanged)
base64	Base64 encoded
quoted-printable	Encoded as quoted printable.
url	Url encoded
urlencoded	Url encoded
7bit	7bit encoded
8bit	8bit encoded

**)default*

Return

Function TIpMail() returns a new TIpMail object and method :new() initializes the object.

Description

Objects of class TIpMail are used to manage a complete eMail. The minimum data an eMail must contain are the sender address (From), the recipient address (To), a subject line (Subject) and the text of the eMail (Body). These data can be assigned to a TIpMail object with its :setBody() and :setHeader() methods.

Note: methods of the TIptMail class are divided into "High level" and "Low level" methods. "High level" methods are designed to compose an email easily, while "Low level" methods manipulate individual entries of an eMail header. You should be familiar with eMail headers when using "Low level" methods.

Instance variables

:hHeaders

Hash holding the mail header entries.

Data type: H
Default: {=>}

Description

The instance variable :hHeaders contains a Hash value. The keys of this hash are the names of the mail header fields while its values are the values of the mail header fields. The methods :getFieldPart(), :getFieldOption(), :setFieldPart() and :setFieldOption() are available to query/assign mail header fields.

High level methods

:attachFile()

Attaches a file to an eMail.

Syntax

```
:attachFile( <cFileName> ) --> lSuccess
```

Arguments

<cFileName>

This is a character string holding the name of the file to attach to the eMail. It must include path and file extension. If the path is omitted from <*cFileName*>, the file is searched in the current directory.

Description

Method :attachFile() attaches a local file to an outgoing eMail. The return value is .T. (true) when the file is attached, otherwise .F. (false) is returned.

The method determines the MIME type of the file and creates a multipart mail message with the file contents encoded base64.

:detachFile()

Stores an attached a file to disk.

Syntax

```
:detachFile( [<cPath>] ) --> lSuccess
```

Arguments

<cPath>

This is a character string holding the directory of the local station where to store the attached file. If <*cPath*> is omitted, the file is stored in the currect directory.

Description

Method :detachFile() retrieves the contents of a file attached to an incoming mail and stores it in the specified directory on the local station. The file name is retrieved with method :getFileName() which extracts the file name from the mail header field of a multipart mail message.

:getBody()

Returns the mail body.

Syntax

```
:getBody() --> cEmailBody
```

Description

Method :getBody() returns the body of a mail message as a character string. When the mail message is a simple mail, the mail body is the text sent wit the eMail. If the mail is a multipart mail, :getBody() returns the contents of the current part. Indivitual parts of a multipart mail message are retrieved with methods :getMultiParts(), :getAttachment() or :nextAttachment().

:getFileName()

Retrieves the name of an attached file.

Syntax

```
:getFileName() --> cFileName
```

Description

Method :getFileName() retrieves the name of a file attached to an incoming eMail and returns it as a character string. If a file is attached, the mail message is a multipart message, of which one or more parts hold the file contents. If the mail message part is no file attachment, a null string is returned ("").

:getMultiParts()

Retrieves all parts of a multipart mail message.

Syntax

```
:getMultiParts() --> aParts
```

Description

Method :getMultiParts() retrieves all parts of a multipart mail message and returns them in a one dimensional array. Each array element holds a TIpMail object holding the contents of an individual part of the multipart message. When the mail message is not a multipart message, an empty array is returned. Use method :isMultiPart() to determine if the incoming mail message is a multipart message.

:isMultipart()

Determines if an eMail is a miltipart mail message.

Syntax

```
:isMultipart() --> lIsMultiPartMail
```

Description

Method :isMultiPart() returns .T. (true) when the mail message is composed of multiple parts. The return value is .F. (false) when the message is not a multipart mail message.

:setBody()

Assigns the contents of the mail body

Syntax

```
:setBody( <cMailBody> ) --> lSuccess
```

Arguments

```
<cMailBody>
```

This is a character string holding the mail body of an eMail. The mail body is the text transmitted with the eMail.

Description

Method :setBody() can be used to assign the mail body of a mail message after the TIpMail object is created. Normally, the contents of the mail body is passed with the :new() method upon object instantiation

:setHeader()

Syntax

```
:setHeader( <cSubject>, ;
            <cFrom>   , ;
            <xTo>     , ;
           [<xCC>]    , ;
           [<xBCC>]     ) --> lSuccess
```

Arguments

```
<cSubject>
```

This is a character string holding the subject line of the mail message.

```
<cFrom>
```

This is a character string holding the sender address.

```
<xTo>
```

This is either a character string holding the address of a single recipient, or a one dimensional array holding addresses of multiple recipients as character strings.

```
<xCC>
```

This is either a character string holding the address of a single CC recipient (Carbon Copy), or a one dimensional array holding addresses of multiple CC recipients as character strings.

```
<xBCC>
```

This is either a character string holding the address of a single BCC recipient (Blind Carbon Copy), or a one dimensional array holding addresses of multiple BCC recipients as character strings.

Description

Method :setHeader() sets the most common header fields of a mail message required for sending eMails. This includes a subject line, and the eMail addresses of the sender and recipient(s).

Low level methods

:attach()

Composes a multipart mail message.

Syntax

```
:attach( <oTIpMail> ) --> lSuccess
```

Arguments

`<oTIpMail>`

This is a TIpMail object holding the contents of a single part of a multipart mail message.

Description

Method :attach() can be used to compose a multipart mail message. Each part of a multipart message is managed by a single TIpMail object. Note that :attach() adds a TIpMail object, it does not attach a file. Use method :attachFile() for file attachments.

The method returns a logical value indicating success of the operation.

:countAttachments()

Returns the number of parts in a multipart mail message.

Syntax

```
:countAttachments() --> nCount
```

Description

Method :countAttachments() returns the number of parts, or TIpMail objects, attached to a multipart mail message as a numeric value. When the mail message is not a multipart message, the return value is zero.

:fromString()

Decomposes a downloaded mail message.

Syntax

```
:fromString( <cEMail> ) --> lSuccess
```

Arguments

`<cEMail>`

This is a character string holding the entire raw mail messge as downloaded from a POP server.

Description

Method :fromString() accepts the raw mail message downloaded from a POP server and decomposes the character string into individual components of the mail message. A mail message is downloaded using a TIpClientPop() object.

The method returns a logical value indicating success of the operation.

:getAttachment()

Returns the current part of a multipart mail message.

Syntax

```
:getAttachment() --> oTIpMail
```

Description

Method :getAttachment() returns a TIpMail object holding the current part of a multipart mail message, or NIL when the message is not a multipart mail message. Use method :resetAttachment() to reset the internal part counter to the first part of a multipart mail message. Method :nextAttachment() increments the internal part counter.

:getCharEncoding()

Returns character encoding information.

Syntax

```
:getCharEncoding() --> cEncoding
```

Description

Method :getCharEncoding() returns a character string holding encoding information. This is included in the mail header field "Content-Type", option "encoding".

:getContentType()

Returns the content type of a mail message.

Syntax

```
:getContentType() --> cContentType
```

Description

Method :getContentType() returns a character string holding the content type of a mail message. This is included in the mail header field "Content-Type".

:getFieldOption()

Returns an optional value of a mail header field.

Syntax

```
:getFieldOption( <cField>, <cOption> ) --> cValue
```

Arguments

<cField>

This is a character string holding the name of the mail header field to query.

<cOption>

This is a character string holding the name of the option of a mail header field to query.

Description

Method :getFieldOption() queries an optional value in a mail header field and returns it as a character strng. When the option, or the mail header field, is not present, a null string ("") is returned.

:getFieldPart()

Returns the value of a mail header field.

Syntax

```
:getFieldPart( <cField> ) --> cValue
```

Arguments

<cField>

This is a character string holding the name of the mail header field to query.

Description

Method :getFieldPart() queries the value of a mail header field and returns it as a character strng. When the mail header field is not present, a null string ("") is returned.

:nextAttachment()

Returns the next part of a multipart mail message.

Syntax

```
:nextAttachment() --> oTIpMail
```

Description

Method :nextAttachment() returns a TIpMail object holding the next part of a multipart mail message, or NIL when no more parts are available. Use method :resetAttachment() to reset the internal part counter to the first part of a multipart mail message. Method :getMultiParts() retrieves all parts is available to retrieve all parts of a multipart message with one method call.

:resetAttachment()

Resets the internal part counter to 1.

Syntax

```
:resetAttachment() --> NIL
```

Description

Method :resetAttachment() resets the internal part counter of a multipart mail message to the first part. This counter is incremented with method :nextAttachment().

:setFieldOption()

Assigns an optional value to a mail header field.

Syntax

```
:setFieldOption( <cField>, <cOption>, <cValue> ) --> lSuccess
```

Arguments

`<cField>`

This is a character string holding the name of the mail header field.

`<cOption>`

This is a character string holding the name of the optional value of a mail header field.

`<cValue>`

This is a character string holding the optional value of a mail header field.

Description

Method :setFieldOption() assigns an optional value to a mail header field and returns .T. (true) when the value is assigned, otherwise .F. (false).

:setFieldPart()

Assigns a value to a mail header field.

Syntax

```
:setFieldPart( <cField>, <cValue> ) --> lSuccess
```

Arguments

`<cField>`

This is a character string holding the name of the mail header field.

`<cValue>`

This is a character string holding the value of a mail header field.

Description

Method :setFieldPart() assigns a value to a mail header field and returns .T. (true) when the value is assigned, otherwise .F. (false).

:toString()

Collects all information of a mail message in a character string.

Syntax

```
:toString() --> cEmail
```

Description

Method :toString() returns a character string holding the entire information of the mail message. This character string can be sent to an SMTP server using a TIpClientSmtp() object and its :data() method.

Info

See also: TIpClientPop(), TIpClientSmtp(), TUrl()
Category: Internet functions, Object functions, xHarbour extensions
Source: tip\mail.prg
LIB: lib\xhb.lib
DLL: dll\xhb.dll

Examples

Incoming eMails

```
// The example outlines the steps required for retrieving all
// eMails from a POP mail server and how to decompose
// incoming mail messages.

   PROCEDURE Main
      LOCAL oPop, oPart, aParts, oTIpMail, aEmails, i

      oPop := TIpClientPop():new( "pop://mailaccount:password@pop.server.com" )

      IF .NOT. oPop:open()
         ? "Connection error:", oPop:lastErrorMessage()
         QUIT
      ELSE
         aEMails := oPop:retrieveAll()
         oPop:close()
      ENDIF

      FOR i:=1 TO Len( aEMails )
         oTIpMail := aEmails[i]
         ? oTIpMail:getFieldPart( "From" )
         ? oTIpMail:getFieldPart( "Subject" )

         IF oTIpMail:isMultiPart()
            // Retrieve all parts of a multipart message
            aParts := oTIpMail:getMultiParts()

            FOR EACH oPart IN aParts
               IF .NOT. Empty( oPart:getFileName() )
                  // This is a file attachment. Store it in the TMP folder.
                  IF oPart:detachFile( "C:\tmp\" )
                     ? "File written: C:\tmp\" + oPart:getFileName()
                  ENDIF
               ELSE
                  ? oPart:getBody()
               ENDIF
            NEXT
         ELSE
            // simple mail message
            ? oTIpMail:getBody()
         ENDIF
      NEXT

   RETURN
```

Outgoing eMails

```
// The example outlines the steps required for composing an eMail
// and sending it to an SMTP mail server.
```

```
PROCEDURE Main
   LOCAL oSmtp, oEMail
   LOCAL cSmtpUrl
   LOCAL cSubject, cFrom, cTo, cBody, cFile

   // preparing data for eMail
   cSmtpUrl := "smtp://mailaccount:password@smtp.server.com"
   cSubject := "Testing eMail"
   cFrom    := "MyName@Mail.server.com"
   cTo      := "YourName@another.server.com"
   cFile    := "File_Attachment.zip"
   cBody    := "This is a test mail sent at: " + DtoC(Date()) + " " + Time()

   // preparing eMail object
   oEMail   := TIpMail():new()
   oEMail:setHeader( cSubject, cFrom, cTo )
   oEMail:setBody( cBody )
   oEMail:attachFile( cFile )

   // preparing SMTP object
   oSmtp := TIpClientSmtp():new( cSmtpUrl )

   // sending data via internet connection
   IF oSmtp:open()
      oSmtp:sendMail( oEMail )
      oSmtp:close()
      ? "Mail sent"
   ELSE
      ? "Error:", oSmtp:lastErrorMessage()
   ENDIF
RETURN
```

TStream()

Abstract class for low-level file data streams.

Sub-classes:

TStreamFileReader()
TStreamFileWriter()

Description

TStream() is an abstract class providing instance variables and methods for the two sub-classes TStreamFileReader() and TStreamFileWriter(). Both classes treat files as low-level streams of data that can be read or written to. The current position of the file pointer is maintained within the TStream object when stream data is read or written to.

TStream() objects are never instantiated directly but have instance variables and methods used in both sub-classes.

Instance variables

:lCanRead

Indicates if stream data can be read.

Data type: L
Default: .F.

Description

The instance variable :lCanRead contains .T. (true) when the *self* object is capable of reading the low-level file stream data, otherwise .F. (false).

:lCanWrite

Indicates if stream data can be written to.

Data type: L
Default: .F.

Description

The instance variable :lCanWrite contains .T. (true) when the *self* object is capable of writing data to the low-level file stream, otherwise .F. (false).

:nLength

Numeric length of the stream in bytes.

Data type: N
Default: 0

Description

The instance variable :nLenght holds a numeric value indicating the total number of bytes, or length, of the low-level file data stream.

:nPosition

Numeric position of file pointer.

Data type: N
Default: 0

Description

The instance variable :nPosition holds a numeric value indicating the current position in the low-level file data stream. :nPosition is updated with each read or write operation occurring in sub-classes.

Methods

:copyTo()

Copies a stream of data to a TStreamFileWriter object.

Syntax

```
:copyTo( <oTStreamFileWriter> ) --> self
```

Arguments

<oTStreamFileWriter>

This must be an object capable of writing to a stream of data, such as a TStreamFileWriter() object.

Description

Method :copyTo() is used to copy the entire stream of data maintained by the *self* object to the target object. The target object's instance variable :lCanRead must contain .T. (true) for the method to work. If this is not the case, the target object is considered to be not writeable and a runtime error is raised.

Info

See also:	FileReader(), FileWriter(), TStreamFileReader(), TStreamFileWriter()
Category:	Object functions, xHarbour extensions
Header:	fileio.ch
Source:	rtl\stream.prg
LIB:	lib\xhb.lib
DLL:	dll\xhbdll.dll

TStreamFileReader()

Creates a new TStreamFileReader object.

Syntax

```
TStreamFileReader():new( <cFileName>, ;
                         [<nOpenMode>] ) --> oTStreamFileReader
```

Arguments

<cFileName>

This is a character string holding the name of the file to open for reading. It must include path and file extension. If the path is omitted from *<cFileName>*, the file is searched in the current directory.

<nOpenMode>

A numeric value specifying the open mode and access rights for the file. #define constants from the FILEIO.CH file can be used for *<nOpenMode>* as listed in the table below:

File open modes

Constant	Value	Description
FO_READ	0	Open file for reading
FO_WRITE	1	Open file for writing
FO_READWRITE *)	2	Open file for reading and writing

**) default*

Constants that define the access or file sharing rights can be added to an FO_* constant. They specify how file access is granted to other applications in a network environment.

File sharing modes

Constant	Value	Description
FO_COMPAT *)	0	Compatibility mode
FO_EXCLUSIVE	16	Exclusive use
FO_DENYWRITE	32	Prevent other applications from writing
FO_DENYREAD	48	Prevent other applications from reading
FO_DENYNONE	64	Allow others to read or write
FO_SHARED	64	Same as FO_DENYNONE

**) default*

Return

The function returns a new TStreamFileReader object and method :new() initializes the object.

Description

TStreamFileReader objects encapsulate low-level file functions for reading a file as a stream of data. The class inherits from the abstract class TStream(), which maintains information about the stream size (file size) and current position within the stream data (file pointer).

Instance variables

:cFile

Name of the file being read.

Data type: C
Default: NIL

Description

The instance variable :cFile stores the first parameter passed to :new(), which is the name of the file being read by a TStreamFileReader object.

:handle

Numeric low-level file handle

Data type: N
Default: 0

Description

This is the numeric low-level file handle of the file being read by a TStreamFileReader object.

Methods

:close()

Closes the data stream, or file.

Syntax

```
:close() --> self
```

Description

Method :close() closes the file opened by a TStreamFileReader object. Refer to function FClose() on closing low-level files.

:finalize()

Destructor method.

Syntax

```
:finalize() --> self
```

Description

This is the destructor method which is automatically called when a TStreamFileReader object gets out of scope. The method makes sure, that the file :cFile is closed.

:read()

Reads data from the stream into memory.

Syntax

```
:read( @<cBuffer>, [<nOffset>], <nBytes> ) --> nBytesRead
```

Arguments

`@<cBuffer>`

A memory variable holding a character string must be passed by reference. It must have at least *<nBytes>* characters.

`<nBytes>`

This is a numeric value specifying the number of bytes to transfer from the data stream into the memory variable *<cBuffer>*, beginning at the current file pointer position.

`<nOffset>`

This is a numeric value specifying the number of bytes to skip at the beginning of *<cBuffer>* where the result is copied to. This allows to copy the data from the file into the middle of a string buffer. The default value is zero. Note that the sum of *<nBytes>*+*<nOffset>* must be smaller than or equal Len(*<cBuffer>*).

Description

Method :read() reads *<nBytes>* bytes from the current position ::nPosition in the data stream and copies them to the memory variable *<cBuffer>*, beginning at the position *<nOffset>*. The file pointer is advanced by *<nBytes>* bytes until the end-of-file is reached. The return value is numeric indicating the number of bytes read from the stream.

Refer to function FRead() for more information on low-level file reading.

:readByte()

Reads a single byte from the data stream.

Syntax

```
:readByte() --> cByte
```

Description

Method :readByte() reads a single byte from the current position in data stream and returns it. When the end-of-file is reached, or when the file is empty, the return value is a null string ("").

:seek()

Changes the position of the file pointer.

Syntax

```
:seek( <nBytes>, <nOrigin> ) --> nPosition
```

Arguments

`<nBytes>`

This is a numeric value specifying the number of bytes to move the file pointer. It can be a positive or negative number. Negative numbers move the file pointer backwards (towards the

beginning of the file), positive values move it forwards. The value zero is used to position the file pointer exactly at the location specified with <*nOrigin*>.

<nOrigin>

Optionally, the starting position from where to move the file pointer can be specified. #define constants are available in the FILEIO.CH file that can be used for <*nOrigin*>.

Start positions for moving the file pointer

Constant	Value	Description
FS_SET *)	0	Start at the beginning of the file
FS_RELATIVE	1	Start at the current file pointer position
FS_END	2	Start at the end of the file

**) default*

Description

Method :seek() is used to change the current position of the low-level file pointer. It returns the new position of the file pointer as a numeric value. Refer to function FSeek() for more information.

Info

See also: FileReader(), FileWriter(), TStream(), TStreamFileWriter()
Category: Object functions, xHarbour extensions
Header: fileio.ch
Source: rtl\stream.prg
LIB: lib\xhb.lib
DLL: dll\xhbdll.dll

Example

```
// The example reads an entire file in blocks of 4096 bytes and fills
// them into an array

   #define BLOCK_SIZE     4096

   PROCEDURE Main
      LOCAL oReader, cFile, cBuffer, nBytes, aBlocks

      aBlocks := {}
      cBuffer := SPace( BLOCK_SIZE )
      cFile   := "myfile.log"
      oReader := TStreamFileReader():new( cFile )

      IF oReader:handle < 0
         ? "Error opening file:", FError()
         QUIT
      ENDIF

      DO WHILE .T.
         nBytes := oReader:read( @cBuffer, , BLOCK_SIZE )

         IF nBytes == BLOCK_SIZE
            AAdd( aBlocks, cBuffer )
         ELSE
            // end of file reached
            AAdd( aBlocks, SubStr( cBuffer, nBytes ) )
            EXIT
         ENDIF
      ENDDO
```

```
   oReader:close()

   ? Len( aBlocks ), "Blocks of", BLOCK_SIZE, "bytes read"
RETURN
```

TStreamFileWriter()

Creates a new TStreamFileWriter object.

Syntax

```
TStreamFileWriter():new( <cFileName>, ;
                         [<nFileAttr>] ) --> oTStreamFileWriter
```

Arguments

<cFilename>

This is a character string holding the name of the file to create and/or write to. It must include path and file extension. If the path is omitted from *<cFileName>*, the file is created or opened in the current directory.

<nFileAttr>

A numeric value specifying one or more attributes associated with the file to create or open. #define constants from the FILEIO.CH file can be used for *<nFileAttr>* as listed in the table below:

Attributes for binary file creation

Constant	Value	Attribute	Description
FC_NORMAL *)	0	Normal	Creates a normal read/write file
FC_READONLY	1	Read-only	Creates a read-only file
FC_HIDDEN	2	Hidden	Creates a hidden file
FC_SYSTEM	4	System	Creates a system file

**) default attribute*

Return

The function returns a new TStreamFileWriter object and method :new() initializes the object.

Description

TStreamFileWriter objects encapsulate low-level file functions for writing streams of data into a file. The class inherits from the abstract class TStream(), which maintains information about the stream size (file size) and current position within the stream data (file pointer).

when the file *<cFileName>* does not exist, it is created during object initialization. when it exists it is opened and the file pointer is positioned at the end-of-file, so that new data is appended to the existing file with method :write() or :writeByte(). To overwrite an existing file, position the file pointer at the beginning with method :seek().

Instance variables

:cFile

Name of the file being written.

Data type: C
Default: NIL

Description

The instance variable :cFile stores the first parameter passed to :new(), which is the name of the file a TStreamFileWriter object writes streams of data to.

:handle

Numeric low-level file handle.

Data type: N
Default: 0

Description

This is the numeric low-level file handle of the file TStreamFileReader object writes data to.

Methods

:close()

Closes the data stream, or file.

Syntax

```
:close() --> self
```

Description

Method :close() closes the file opened by a TStreamFileWriter object. Refer to function FClose() on closing low-level files.

:finalize()

Destructor method.

Syntax

```
:finalize() --> self
```

Description

This is the destructor method which is automatically called when a TStreamFileWriter object gets out of scope. The method makes sure, that the file :cFile is closed.

:seek()

Changes the position of the file pointer.

Syntax

```
:seek( <nBytes>, <nOrigin> ) --> nPosition
```

Arguments

<nBytes>

This is a numeric value specifying the number of bytes to move the file pointer. It can be a positive or negative number. Negative numbers move the file pointer backwards (towards the beginning of the file), positive values move it forwards. The value zero is used to position the file pointer exactly at the location specified with *<nOrigin>*.

<nOrigin>

Optionally, the starting position from where to move the file pointer can be specified. #define constants are available in the FILEIO.CH file that can be used for *<nOrigin>*.

Start positions for moving the file pointer

Constant	Value	Description
FS_SET *)	0	Start at the beginning of the file
FS_RELATIVE	1	Start at the current file pointer position
FS_END	2	Start at the end of the file

**) default*

Description

Method :seek() is used to change the current position of the low-level file pointer. It returns the new position of the file pointer as a numeric value. Refer to function FSeek() for more information.

:write()

Writes data from memory into the stream.

Syntax

```
:write( <cBuffer>, [<nOffset>], [<nBytes>] ) --> nBytesWritten
```

Arguments

`<cBuffer>`

This is a character string to write to the stream.

`<nBytes>`

A numeric value specifying the number of bytes to write to the stream, beginning with the first character of <*cBuffer*>. It defaults to Len(<*cBuffer*>), i.e. all bytes of <*cBuffer*>.

`<nOffset>`

This is a numeric value specifying the number of bytes to skip at the beginning of <*cBuffer*>. This allows to write data from the middle of a string buffer into the stream. The default value is zero. Note that the sum of <*nBytes*>+<*nOffset*> must be smaller than or equal Len(<*cBuffer*>).

Description

Method :write() writes <*nBytes*> minus <*nOffset*> bytes from the string buffer <*cBuffer*> to the data stream, beginning at the current position ::nPosition. The file pointer is advanced by <*nBytes*>-<*nOffset*> bytes and the number of bytes written is returned as a numeric value.

Refer to function FWrite() for more information on writing to a low-level file.

:writeByte()

Writes a single byte into the data stream.

Syntax

```
:writeByte( <cByte> ) --> self
```

Arguments

`<cByte>`

This is a single character to write into the data stream.

Description

Method :writeByte() writes a single byte to the current position in data stream and advances the file pointer by 1. When the byte cannot be written, a runtime error is raised.

Info

See also:	FileReader(), FileWriter(), TStream(), TStreamFileReader()
Category:	Object functions, xHarbour extensions
Header:	fileio.ch
Source:	rtl\stream.prg
LIB:	lib\xhb.lib
DLL:	dll\xhbdll.dll

Example

```
// The example lists all file names of the current directory
// to a text file.

   PROCEDURE Main
      LOCAL aFiles := Directory()
      LOCAL cFile  := "Directory.txt"
      LOCAL oWriter:= TStreamFileWriter():new( cFile )
      LOCAL aFile

      FOR EACH aFile IN aFiles
         oWriter:write( aFile[1],, Len(aFile[1]) )
         oWriter:writeByte( Chr(13) )
         oWriter:writeByte( Chr(10) )
      NEXT

      oWriter:close()
   RETURN
```

TUrl()

Creates a new TUrl object.

Syntax

```
TUrl():new( <cURL> ) --> oTUrl
```

Arguments

```
<cUrl>
```

This is a character string holding a Uniform Resource Locator (URL).

Return

Function TUrl() returns a new TUrl object and method :new() initializes the object.

Description

Objects of class TUrl are used to manage a complete URL and query its individual components. A URL is generally composed of the following parts, most of which are optional:

```
http://user:pass@www.xHarbour.com:1080/xhdn/index.html?avar=0&avar1=1
^__^   ^__^ ^__^ ^_______________^ ^__^ ^______________^ ^_____________^
Proto  UID  PWD       Server       Port   Path              Query
                                         ^__^ ^_________^
                                         Dir     File
                                              ^___^ ^__^
                                               Name  Ext
```

The minimum requirement of a URL is the communication protocol to use and the server to connect to.

TUrl objects are mainly used by other TIP classes available for the various internet protocols.

Instance variables

:cAddress

Character string holding the entire URL.

Data type: C
Default: ""

Description

The instance variable :cAddress holds the entire URL as a character string after method :buildAddress() is called.

:cFile

CHaracter string holding the name of the file to access.

Data type: C
Default: ""

Description

The instance variable :cFile holds the name of the file to access as a character string. It must include the file extension.

:cPassword

Optional password for server login.

Data type: C
Default: ""

Description

The instance variable :cPassword holds a character string with the login password for the server.

:cPath

CHaracter string holding the path of the file to access.

Data type: C
Default: ""

Description

The instance variable :cFile holds the path of the file to access as a character string.

:cProto

CHaracter string holding the internet protocol to use for server access.

Data type: C
Default: ""

Description

The instance variable :cProto holds a character string describing the internet protocol to use for server communications. The following protocols are supported with the TIp classes:

Internet protocols

:cProto	TIp class	Description
"ftp"	TIpClientFtp()	File Transfer Protocol
"http"	TIpClientHttp()	HyperText Transfer Protocol
"pop"	TIpClientPop()	Post Office Protocol
"smtp"	TIpClientSmtp()	Simple Mail Transfer Protocol

:cQuery

Optional query string.

Data type: C
Default: ""

Description

The instance variable :cQuery holds a character string with query data. They are easily added to a TUrl object with its :addGetForm() method.

:cServer

Character string holding the server address.

Data type: C
Default: ""

Description

The instance variable :cServer holds a character string with the server address. "www.xharbour.com" is an example for a server address.

:cUserId

Optional user ID for server login.

Data type: C
Default: ""

Description

The instance variable :cUserId holds a character string with the login user ID for the server.

:nPort

Numeric port number.

Data type: N
Default: -1

Description

The instance variable :nPort holds the port number to use for server connection.

Methods

:addGetForm()

Adds data to post to a server with the URL.

Syntax

```
:addGetForm( <xPostData> ) --> cPostData
```

Arguments

<xPostData>

This parameter can be of data type Hash, Array or Character. It defines the form data to send to the server with the URL.

When *<xPostData>* is a Hash, its hash keys define the form fields and the hash values define the field values for a Html form. The hash values must be character strings.

When an array is passed, it must be a two dimensional array with two columns. The first column defined the field names and the second column the field values of a form.

When a character string is passed, it must bein the form "name1=value1&name2=value2"

Description

Method :addGetForm() provides a comfortable way of creating a URL encoded character string used to transmit name/value pairs of data to a server. The method returns a string holding the URL encoded data.

:buildAddress()

Returns the entire URL string.

Syntax

```
:buildAddress() --> cUrl
```

Description

Method :buildAddress() creates the entire URL from all components held in instance variables of a TUrl object and returns the URL as a character string.

:buildQuery()

Returns the entire URL query string.

Syntax

```
:buildQuery() --> cUrlQuery
```

Description

Method :buildQuery() creates the URL query from corresponding components held in instance variables of a TUrl object and returns the URL query as a character string.

:setAddress()

Changes the URL maintained by the object.

Syntax

```
:setAddress( <cUrl> ) --> lSuccess
```

Arguments

This is a character string holding a new Uniform Resource Locator (URL).

Description

Method :setAddress() accepts a complete URL string and assigns it to the appropriate instance variables of a TUrl object. This allows to re-use an existing TUrl object with a new URL.

Info

See also: TIpClientFtp(), TIpClientHttp(), TIpClientPop(), TIpClientSmtp()
Category: Internet functions, Object functions, xHarbour extensions
Source: tip\url.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example creates a TUrl object and displays the various
// components of a URL.

   PROCEDURE Main
      LOCAL cUrl, oUrl, hFormData

      cUrl := "http://user:pass@www.xHarbour.com:1080/xhdn/index.html?var=0&var1=1"
      oUrl := TUrl():new( cUrl )

      ? oUrl:cAddress        // result: (empty string)
      ? oUrl:cFile           // result: index.html
      ? oUrl:cPassword       // result: pass
      ? oUrl:cPath           // result: /xhdn/
      ? oUrl:cProto          // result: http
      ? oUrl:cQuery          // result: avar=0&avar1=1
      ? oUrl:cServer         // result: www.xHarbour.com
      ? oUrl:cUserid         // result: user
      ? oUrl:nPort           // result: 1080

      ? "--- oUrl:buildQuery() ----------------------"
      ? oUrl:buildQuery()

      ? "--- oUrl:buildAddress() --------------------"
      ? oUrl:buildAddress()

      hFOrmData := {=>}
      hFormData["NAME"] := "Smith"
      hFormData["CITY"] := "Los Angeles"

      ? "---- oUrl:addGetForm( hFormData ) ---------"
      ? oUrl:addGetForm( hFormData )

      ? "--- oUrl:buildQuery() ----------------------"
      ? oUrl:buildQuery()

      ? "--- oUrl:buildAddress() --------------------"
      ? oUrl:buildAddress()

   RETURN
```

TXmlDocument()

Creates a new TXmlDocument object.

Syntax

```
TXmlDocument():new( [<nFileHandle>|<cXmlString>], [<nStyle>] ) --> oTXmlDocument
```

Arguments

`<nFileHandle>`

This is a file handle of an XML file to read. It is returned from function FOpen().

`<cXmlString>`

Instead of a file handle, an XML formatted character string can be passed. If the first parameter is omitted, the object has no XML data, but can be used to add XML nodes and create a new XML file (see example).

`<nStyle>`

This parameter instructs the TXMlDocument object how to read an XML file and/or how to write XML nodes into a file. #define constants listed in Hbxml.ch are used to specify *<nStyle>*:

Constants for XML object creation

Constant	Value	Description
HBXML_STYLE_INDENT	1	Indents XML nodes with one space
HBXML_STYLE_TAB	2	Indents XML nodes with tabs
HBXML_STYLE_THREESPACES	4	Indents XML nodes with three spaces
HBXML_STYLE_NOESCAPE	8	Reads and creates unescaped characters in data sections

Note: when the style HBXML_STYLE_NOESCAPE is set, the textual content enclosed in an opening and closing XML tag is scanned for characters that normally must be escaped in XML. This can lead to a considerable longer time for reading the XML data.

The characters to be escaped are single and double quotes ('"), ampersand (&), and angled brackets (<>). If such characters exist in textual content and are not escaped, a parsing error is generated, unless HBXML_STYLE_NOESCAPE is used.

Return

Function TXmlDocument() creates the object and method :new() initializes it.

Description

The TXmlDocument() class provides objects for reading and creating XML files. XML stands for e**X**tensible **M**arkup **L**anguage which is similar to HTML, but designed to describe data rather to display it. To learn more about XML itself, the internet provides very good free online tutorials. The website www.w3schools.com is is a good place to quickly learn the basics on XML.

A TXmlDocument object maintains an entire XML document and builds from it a tree of TXmlNode() objects which contain the actual XML data. The first XML node is stored in the :oRoot instance variable, which is the root node of the XML tree. Beginning with the root node, an XML document can be traversed or searched for particular data. The classes TXmlIteratorScan() and TXmlIteratorRegEx() are available to find a particular XML node, based on its tag name, attribute or data it contains.

Instance variables

:oRoot

Root node of the XML tree.

Data type: O
Default: NIL

Description

The instance variable :oRoot contains a TXmlNode() object representing the root node of the XML tree. It can be used as starting point for searching the tree with iterator objects like TXmlIteratorScan().

:nStatus

Status information on XML parsing.

Data type: N
Default: HBXML_STATUS_OK

Description

The instance variable :nStatus contains a numeric value indicating the current status while a TXmlObject is reading a file or XML string and builds the tree of TXmlNode() objects. #define constants are listed in Hbxml.ch that can be used to check the status. They begin with the prefix HBXML_STATUS_.

:nError

Error code for XML parsing.

Data type: N
Default: HBXML_ERROR_NONE

Description

When the XML file or string cannot be parsed, the instance variable :nError contains a numeric value <> 0, indicating the reason for failure. #define constants are listed in Hbxml.ch that can be used to check the error. They begin with the prefix HBXML_ERROR_. A textual description of the error can be obtained by passing :nError to function HB_XmlErrorDesc().

:nLine

Current line number being parsed.

Data type: N
Default: 1

Description

The instance variable :nLine contains the line number currently being parsed. Line numbering begins with 1. If an error occurs during parsing, :nLine contains the line number of the offending line.

:oErrorNode

TXmlNode object containing illegal XML code.

Data type: O
Default: NIL

Description

The instance variable :oErrorNode is assigned a TXmlNode() object when the XML parser detects illegal XML code. The object in :oErrorNode contains the illegal XML data and is available for inspection. If no error is detected, :oErrorNode contains NIL.

:nNodeCount

Number of nodes in the XML tree.

Data type: N
Default: 0

Description

The instance variable :nNodeCount contains a numeric value indicating the total number of nodes in the XML tree.

Methods for XML data manipulation

:read()

Reads an XML file or string.

Syntax

```
:read( [<nFileHandle>|<cXmlString>], [<nStyle>] ) --> self
```

Arguments

<nFileHandle>

This is a file handle of an XML file to read. It is returned from function FOpen().

<cXmlString>

Instead of a file handle, an XML formatted character string can be passed. If the first parameter is omitted, the object has no XML data, but can be used to add XML nodes and create a new XML file.

<nStyle>

This parameter instructs the TXMlDocument object how to read an XML file and/or how to write XML nodes into a file. #define constants listed in Hbxml.ch are used to specify *<nStyle>*:

Constants for XML object creation

Constant	Value	Description
HBXML_STYLE_INDENT	1	Indents XML nodes with one space
HBXML_STYLE_TAB	2	Indents XML nodes with tabs
HBXML_STYLE_THREESPACES	4	Indents XML nodes with three spaces
HBXML_STYLE_NOESCAPE	8	Reads and creates unescaped characters in data sections

Description

Method :read() receives the same parameters as the :new() method and instructs an existing TXmlDocument object to discard the previously created XML tree and build a new one from the file handle or XML string. When the XML data is sucessfully parsed, instance variable :nError is set to zero.

Note: when the style HBXML_STYLE_NOESCAPE is set, the textual content enclosed in an opening and closing XML tag is scanned for characters that normally must be escaped in XML. This can lead to a considerable longer time for reading the XML data.

The characters to be escaped are single and double quotes ('"), ampersand (&), and angled brackets (<>). If such characters exist in textual content and are not escaped, a parsing error is generated, unless HBXML_STYLE_NOESCAPE is used.

:toString()

Creates an XML formatted character string.

Syntax

```
:toString( <nStyle> ) --> cXmlString
```

Arguments

`<nStyle>`

This parameter instructs the TXMlDocument object how to create the XML code. #define constants listed in Hbxml.ch are used to specify <*nStyle*>:

Constants for XML code creation

Constant	Value	Description
HBXML_STYLE_INDENT	1	Indents XML nodes with one space
HBXML_STYLE_TAB	2	Indents XML nodes with tabs
HBXML_STYLE_THREESPACES	4	Indents XML nodes with three spaces
HBXML_STYLE_NOESCAPE	8	Creates unescaped characters in data sections

Description

Method :toString() creates an XML formatted character string containing all nodes and data currently held in the XML tree. The <*nStyle*> parameter can be used to indent the nodes for readability, or to leave characters in data sections unescaped.

:write()

Creates an XML formatted character string.

Syntax

```
:write( <nFileHandle>, [<nStyle>] ) --> self
```

Arguments

`<nFileHandle>`

This is the handle of a file to write XML data to. It is returned from function FCreate().

<nStyle>

This parameter instructs the TXMlDocument object how to create the XML code. #define constants listed in Hbxml.ch are used to specify *<nStyle>*:

Constants for XML code creation

Constant	Value	Description
HBXML_STYLE_INDENT	1	Indents XML nodes with one space
HBXML_STYLE_TAB	2	Indents XML nodes with tabs
HBXML_STYLE_THREESPACES	4	Indents XML nodes with three spaces
HBXML_STYLE_NOESCAPE	8	Creates unescaped characters in data sections

Description

Method :write() writes all XML nodes and data currently held in the XML tree to the open file *<nFileHandle>*. The *<nStyle>* parameter can be used to indent the nodes for readability, or to leave characters in data sections unescaped.

Methods for searching and navigating

:findFirst()

Locates the first XML node containing particular data.

Syntax

```
:findFirst( [<cTagName>]        , ;
            [<cAttributeName>]  , ;
            [<cAttributeValue>] , ;
            [<cData>]           ) --> oTXmlNode | NIL
```

Arguments

<cTagName>

This is a character string holding the name of the XML node to find.

<cAttributeName>

This is a character string holding the name of an attribute of the XML node to find.

<cAttributeValue>

This is a character string holding the value of the attribute of the XML node to find.

<cData>

This is a character string holding the textual content of the XML node to find.

Description

Method :findFirst() locates the first XML node in the XML tree that matches the search criteria. If no parameter is passed, the first XML node in the tree is returned. Search criteria can be any combination of name, attribute and textual content. If no matching node is found, the return value is NIL.

If a matching node is found, the next node matching the same search criteria is returned from method :findNext().

:findFirstRegEx()

Locates the first XML node containing particular data using regular expressions.

Syntax

```
:findFirstRegEx( [<cTagNameRegEx>]         , ;
                 [<cAttributeNameRegEx>]   , ;
                 [<cAttributeValueRegEx>]  , ;
                 [<cDataRegEx>]              ) --> oTXmlNode | NIL
```

Arguments

<cTagNameRegEx>

This is a character string holding the regular expression to match with name of the XML node to find.

<cAttributeNameRegEx>

This is a character string holding the regular expression to match with the name of an attribute of the XML node to find.

<cAttributeValueRegEx>

This is a character string holding the regular expression to match with the value of the attribute of the XML node to find.

<cDataRegEx>

This is a character string holding the regular expression to match with the textual content of the XML node to find.

Description

Method :findFirstRegEx() locates the first XML node in the XML tree that matches the regular expressions passed. If no parameter is passed, the first XML node in the tree is returned. Search criteria can be any combination of name, attribute and textual content. If no matching node is found, the return value is NIL.

If a matching node is found, the next node matching the same search criteria is returned from method :findNext().

:findNext()

Finds the next XML node matching a search criteria.

Syntax

```
:findNext() --> oTXmlNode | NIL
```

Description

Method :findNext() continues the search for a matching XML node and returns the corresponding TXmlNode object. The search criteria defined with a previous call to :findFirst() or :findFirstRegEx() is used. If no further XML node is found, the return value is NIL.

Info

See also: TXmlIterator(), TXmlIteratorRegEx(), TXmlIteratorScan(), TXmlNode()
Category: Object functions, xHarbour extensions
Header: hbxml.ch
Source: rtl\txml.prg
LIB: xhb.lib
DLL: xhbdll.dll

Examples

Creating an XML file

```
// The example uses the Customer database and creates an XML file
// from it. The database structure and its records are written
// to different nodes in the XML file. The basic XML tree is this:
//
// <database>
//  <structure>
//   <field .../>
//  </structure>
//  <records>
//   <record>
//    <fieldname> data </fieldname>
//   </record>
//  </records>
// </database>

   #include "hbXml.ch"

   PROCEDURE Main
      LOCAL aStruct, aField, nFileHandle
      LOCAL oXmlDoc, oXmlNode, hAttr, cData
      LOCAL OXmlDatabase, oXmlStruct, oXmlRecord, oXmlField

      USE Customer

      aStruct  := DbStruct()

      // Create empty XML document with header
      oXmlDoc  := TXmlDocument():new( '<?xml version="1.0"?>' )

      // Create main XML node
      oXmlDatabase := TXmlNode():new( , "database", { "name" => "CUSTOMER" } )
      oXmlDoc:oRoot:addBelow( oXmlDatabase )

      // copy structure information to XML
      oXmlStruct := TXmlNode():new( , "structure" )
      oXmlDataBase:addBelow( oXmlStruct )

      FOR EACH aField IN aStruct
         // store field information in XML attributes
         hAttr := { "name" => Lower( aField[1] ), ;
                    "type" => aField[2], ;
                    "len"  => LTrim( Str(aField[3]) ), ;
                    "dec"  => LTrim( Str(aField[4]) )  }

         oXmlField := TXmlNode():new(, "field", hAttr )
         oXmlStruct:addBelow( oXmlField )
      NEXT
```

```
      // copy all records to XML
      oXmlNode := TXmlNode():new( , "records" )
      oXmlDataBase:addBelow( oXmlNode )

      DO WHILE .NOT. Eof()
         hAttr      := { "id" => LTrim( Str( Recno() ) ) }
         oXmlRecord := TXmlNode():new( , "record", hAttr )

         FOR EACH aField IN aStruct
            IF aField[2] == "M"
               // Memo fields are written as CDATA
               cData     := FieldGet( Hb_EnumIndex() )
               oXmlField := TXmlNode():new( HBXML_TYPE_CDATA  , ;
                                            Lower( aField[1] ), ;
                                            NIL               , ;
                                            cData               )
            ELSE
               // other fields are written as normal tags
               cData     := FieldGet( Hb_EnumIndex() )
               cData     := Alltrim( CStr( cData ) )
               oXmlField := TXmlNode():new( HBXML_TYPE_TAG    , ;
                                            Lower( aField[1] ), ;
                                            NIL               , ;
                                            cData               )
            ENDIF
            // add field node to record
            oXmlRecord:addBelow( oXmlField )
         NEXT

         // add record node to records
         oXmlNode:addBelow( oXmlRecord )
         SKIP
      ENDDO

      // create XML file
      nFileHandle := FCreate( "Customer.xml" )

      // write the XML tree
      oXmlDoc:write( nFileHandle, HBXML_STYLE_INDENT )

      // close files
      FClose( nFileHandle )
      USE
   RETURN
```

Reading an XML file

```
// This example uses the Customer.xml file created in the
// prevous example and extracts from it the structure definition
// for the Customer.dbf file.

   PROCEDURE Main
      LOCAL oXmlDoc := TXmlDocument():new()
      LOCAL oXmlNode, aStruct := {}

      oXMlDoc:read( Memoread( "customer.xml" ) )

      oXmlNode := oXmlDoc:findFirst()
      ? oXmlNode:cName

      oXmlNode := oXmlDoc:findFirst( "structure" )
      ? oXmlNode:cName
```

```
      oXmlNode := oXmlDoc:findFirst( "field" )

      DO WHILE oXmlNode <> NIL
         // attributes are stored in a hash
         AAdd( aStruct, { oXmlNode:aAttributes[ "name" ]      , ;
                          oXmlNode:aAttributes[ "type" ]      , ;
                          Val( oXmlNode:aAttributes[ "len" ] ), ;
                          Val( oXmlNode:aAttributes[ "dec" ] )  } )

         oXmlNode := oXmlDoc:findNext()
      ENDDO

      AEval( aStruct, {|a| Qout( ValToPrg(a) ) } )
   RETURN
```

Escape characters in XML

```
// The example demonstrates the effect of HBXML_STYLE_NOESCAPE
// when XML code is created.

   #include "hbXml.ch"

   PROCEDURE Main
      LOCAL oXmlDoc, oXmlNode

      oXmlDoc := TXmlDocument():new( '<?xml version="1.0"?>' )
      oXmlNode:= TXmlNode():new( , "text", , [this must be escaped: "'&<>] )

      oXmlDoc:oRoot:addBelow( oXmlNode )

      ? oXmlDoc:toString()

      ** output:
      // <?xml version="1.0"?>
      // <text>this must be escaped: &quot;&apos;&amp;&lt;&gt;</text>

      ? oXmlDoc:toString( HBXML_STYLE_NOESCAPE )

      ** output:
      // <?xml version="1.0"?>
      // <text>this must be escaped: "'&<></text>
   RETURN
```

TXmlIterator()

Creates a new TXmlIterator object.

Syntax

```
TXmlIterator():new( <oTXmlNode> ) --> oTXmlIterator
```

Arguments

<oTXmlNode>

This is a TXmlNode() object to create the iterator object for.

Return

The function returns a TXmlIterator object and method :new() initializes it.

Description

The TXmlIterator class provides objects for iterating nodes in an XML document and its sub-nodes (in the XML tree). An XML document is managed by an object of the TXmlDocument() class.

The creation of a TXmlIterator object requires a TXmlNode() object wich serves as starting point for the iterator. The iterator is restricted to the branch in the XML tree represented by the initial XML node.

Methods

:clone()

Clones the TXmlIterator object.

Syntax

```
:clone() --> oClone
```

Description

Method :clone() creates a duplicate of the self object. The returned TXmlIterator object uses the same search criteria and the same TXmlNode object for a search.

:getNode()

Retrieves the current XML node matching the search criteria.

Syntax

```
:getNode() --> oTXmlNode | NIL
```

Description

Method :getNode() returns the current TXmlNode() object as it is determined by a previous call to :next(). If no further XML node exists, the return value is NIL.

:next()

Retrieves the next XML node in the tree.

Syntax

```
:next() --> oTXmlNode | NIL
```

Description

Method :next() returns the next TXmlNode() object. If no further XML node is found, the return value is NIL.

:rewind()

Goes back to the first XML node matching the search criteria.

Syntax

```
:rewind() --> oTXmlNode | NIL
```

Description

Method :rewind() sets the XML node passed to the :new() method as the current node, thus rewinding the entire iteration.

:setContext()

Defines the currently found XML node as first node.

Syntax

```
:setContext() --> self
```

Description

Method :setContext() defines the currently found XML node as the first node. An iteration can then be restarted with this node when method :rewind() is called.

Info

See also: TXmlDocument(), TXmlNode(), TXmlIteratorScan(), TXmlIteratorRegEx()
Category: Object functions, xHarbour extensions
Source: rtl\txml.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example use the Customer.xml file as created with the
// TXmlDocument() example, and extracts from it the field names
// of the database structure. Note that the iterator is restricted
// to the subnodes of the "structure" node.

   PROCEDURE Main
      LOCAL oXmlDoc := TXmlDocument():new()
      LOCAL oXmlNode, oXmlIter

      oXMlDoc:read( Memoread( "customer.xml" ) )

      oXmlNode := oXmlDoc:findFirst( "structure" )
```

```
   oXmlIter := TXmlIterator():new( oXmlNode )

   DO WHILE .T.
      oXmlNode := oXmlIter:next()
      IF oXmlNode == NIL
         EXIT
      ENDIF

      ? oXmlNode:getAttribute( "name" )
   ENDDO
RETURN
```

TXmlIteratorRegEx()

Creates a new TXmlIteratorRegEx object.

Syntax

```
TXmlIteratorRegEx():new( <oTXmlNode> ) --> oTXmlIteratorRegEx
```

Arguments

<oTXmlNode>

This is a TXmlNode() object to create the iterator object for. It serves as starting node for iterating.

Return

The function returns a TXmlIteratorRegEx object and method :new() initializes it.

Description

The TXmlIteratorRegEx class is derived from the TXmlIterator() class and has the same methods. The only difference is that regular expressions are used to define search criteria for finding a particular XML node in an XML tree. The regular expressions are defined once with the :find() method, which searches for the first matching XML node. Subsequent XML nodes matching the regular expressions then searched with the :next() method.

The end of a search is indicated when either :find() or :next() return NIL instead of a TXmlNode object matching the regular expressions.

Search methods

:find()

Searches the first XML node matching the regular expressions.

Syntax

```
:find( [<cRegExTagName>]  , ;
       [<cRegExAttribute>], ;
       [<cRegExValue>]    , ;
       [<cRegExData>]       ) --> oTXmlNode | NIL
```

Arguments

<cRegExTagName>

This is a character string holding the regular expression for matching the tag name of XML nodes.

<cRegExAttribute>

This is a character string holding the regular expression for matching the name of an XML attribute.

<cRegExValue>

The a regular expression to match an attribute value optionally defined with the character string *<cRegExValue>*.

<cRegExData>

This is an optional character string holding the regular expression for matching the textual data of XML nodes.

Description

Method :find() defines the regular expressions for a TXmlIteratorRegEx object and performs the initial search. Four different regular expressions can be defined to match tag name, attribute name, attribute value or textual data. If a matching XML node exists, the corresponding TXmlNode() object is returned, otherwise the return value is NIL. The search can be continued using the same regular expressions with method :next().

:next()

Searches the next XML node matching the regular expressions.

Syntax

```
:next() --> oTXmlNode | NIL
```

Description

Method :next() returns the next TXmlNode() object matching the regular expressions defined with the :find() method. If no further XML node is found, the return value is NIL.

Info

See also:	TXmlDocument(), TXmlNode(), TXmlIterator(), TXmlIteratorScan()
Category:	Object functions, xHarbour extensions
Source:	rtl\txml.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example use the Customer.xml file as created with the
// TXmlDocument() example. Two XmlIteratorScan objects are used
// to scan the <record> nodes and field nodes below record nodes.
// The example displays the contents of the <lastname> nodes
// whose textual content begin with "R" or "S".

   PROCEDURE Main
      LOCAL oXmlDoc := TXmlDocument():new()
      LOCAL oXmlRecord, oXmlField, oXmlRecScan, oXmlFieldScan
      LOCAL cRegEx := "^[R?]|^[S?]"

      oXMlDoc:read( Memoread( "customer.xml" ) )

      oXmlNode := oXmlDoc:findFirst( "records" )

      // iterator for <record> nodes
      oXmlRecScan := TXmlIteratorScan():new( oXmlNode )
      oXmlRecord  := oXmlRecScan:find( "record" )

      DO WHILE oXmlRecord <> NIL
         // iterator for <fieldname> nodes
         oXmlFieldScan := TXmlIteratorRegEx():new( oXmlRecord )
         oXmlField     := oXmlFieldScan:find( "lastname" ,,, cRegEx )

         IF oXmlField <> NIL
            ? oXmlField:cData
```

```
      ENDIF

      oXmlRecord := oXmlRecScan:next()
   ENDDO

RETURN
```

TXmlIteratorScan()

Creates a new TXmlIteratorScan object.

Syntax

```
TXmlIteratorScan():new( <oTXmlNode> ) --> oTXmlIteratorScan
```

Arguments

`<oTXmlNode>`

This is a TXmlNode() object to create the iterator object for. It is the starting point for searching nodes in the XML tree.

Return

The function returns a TXmlIteratorScan object and method :new() initializes it.

Description

The TXmlIteratorScan class is derived from the TXmlIterator() class and has the same methods. The only difference is that search criteria can be defined to find a particular XML node in an XML tree. The search criteria is defined once with the :find() method, which searches for the first matching XML node. Subsequent XML nodes matching the search criteria are then searched with the :next() method.

The end of a search is indicated when either :find() or :next() return NIL instead of a TXmlNode object matching the search criteria.

Search methods

:find()

Searches the first XML node matching the search criteria.

Syntax

```
:find( [<cTagName>]  , ;
       [<cAttribute>], ;
       [<cValue>]    , ;
       [<cData>]       ) --> oTXmlNode | NIL
```

Arguments

`<cTagName>`

This is a character string holding the tag name of the XML node to search for.

`<cAttribute>`

This is a character string holding the name of an XML attribute to search for.

`<cValue>`

The attribute value to search is optionally defined with the character string *<cValue>*.

`<cData>`

This is an optional character string holding the textual data of an XML node to search.

Description

Method :find() defines the search criteria for a TXmlIteratorScan object and performs the initial search. The search criteria can be defined as any combination of tag name, attribute name, attribute value or textual data. If a matching XML node exists, the corresponding TXmlNode() object is returned, otherwise the return value is NIL. The search can be continued using the same search criteria with method :next().

:next()

Searches the next XML node matching the search criteria.

Syntax

```
:next() --> oTXmlNode | NIL
```

Description

Method :next() returns the next TXmlNode() object matching the search criteria defined with the :find() method. If no further XML node is found, the return value is NIL.

Info

See also:	TXmlDocument(), TXmlNode(), TXmlIterator(), TXmlIteratorRegEx()
Category:	Object functions, xHarbour extensions
Source:	rtl\txml.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example use the Customer.xml file as created with the
// TXmlDocument() example. Two XmlIteratorScan objects are used
// to scan the <record> nodes and field nodes below record nodes.
// The example displays the reecord IDs and the data from the <lastname>
// nodes.

   PROCEDURE Main
      LOCAL oXmlDoc := TXmlDocument():new()
      LOCAL oXmlNode, oXmlRecScan, oXmlFieldScan

      oXMlDoc:read( Memoread( "customer.xml" ) )

      oXmlNode := oXmlDoc:findFirst( "records" )

      // iterator for <record> nodes
      oXmlRecScan := TXmlIteratorScan():new( oXmlNode )
      oXmlNode    := oXmlRecScan:find( "record" )

      DO WHILE oXmlNode <> NIL
         ? oXmlNode:getAttribute( "id" )

         // iterator for <fieldname> nodes
         oXmlFieldScan := TXmlIteratorScan():new( oXmlNode )
         oXmlNode      := oXmlFieldScan:find( "lastname" )
         ?? " ", oXmlNode:cData

         oXmlNode := oXmlRecScan:next()
      ENDDO

   RETURN
```

TXmlNode()

Creates a new TXmlNode object.

Syntax

```
TXmlNode():new( [<nType>]       , ;
                 <cTagName>     , ;
                [<hAttributes>], ;
                [<cData>]        ) --> oTXmlNode
```

Arguments

`<nType>`

This is an optional numeric value indicating the type of the XML node. XML node types are distinguished by the folloing #define constants listed in Hbxml.ch:

Types of XML nodes

Constant	Value	Description
HBXML_TYPE_TAG *)	0	Regular XML node
HBXML_TYPE_COMMENT	1	XML comment
HBXML_TYPE_PI	2	XML processing instruction
HBXML_TYPE_DIRECTIVE	3	XML directive
HBXML_TYPE_DATA	4	Parsed XML data section (PCDATA)
HBXML_TYPE_CDATA	5	Unparsed XML data section (CDATA)
HBXML_TYPE_DOCUMENT	6	XML document

**) default type*

`<cTagName>`

This is a character string holding the tag name of the XML node to create.

`<hAttributes>`

Optionally, the attributes of the XML node to create can be passed as a Hash. The Hash key is the attribute name and the Hash value is the attribute value. Note that both, Hash values and keys, must be of data type Character.

`<cData>`

This is an optional character string holding the textual data of the TXmlNode object. If this parameter is used, the TXmlNode object represents a leaf, not a node. *<cData>* is the text appearing between the opening and closing XML *<cTagName>*.

Return

The function returns a TXmlNode object and method :new() initializes the object.

Description

Objects of the TXmlNode class represent a single node, or a leaf, in an XML document. XML documents are maintained by the TXmlDocument() class which builds a tree of XML nodes when reading an XML document. A TXmlDocument object creates this tree and fills it with TXmlNode objects.

One TXmlNode object contains in its instance variables the XML tag name, an optional list of XML atttributes, and the textual content of an XML node, when it is a leaf node.

The XML node hierarchy of an XML document is reflected in instance variables of an TXmlNode object. These are :oParent and :oChild holding TXmlNode objects of the next higher or lower level in

the XML tree, plus :oNext and :oPrev holding the next or previous TXmlNode object of the same level in the XML tree. The nesting level of a TXmlNode object can be queried with method :depth().

Note: the beforementioned instance variables allow for traversing an XML tree programmatically. Before doing so, the search and scan capabilities of the classes TXmlIterator(), TXmlIteratorRegEx() and TXmlIteratorScan() should be investigated. They are designed to aid in traversing an XML document and extracting information from it.

Instance variables

:aAttributes

Contains attributes of an XML node.

Data type: H
Default: {=>}

Description

The instance variable :aAttributes contains a Hash. The hash keys are the names of the attributes while the Hash values are the attribute values. Attribute values are of data type Character.

:cData

Contains textual data of an XML node.

Data type: C
Default: NIL

Description

The instance variable :cData contains a character string representing the texual content of an XML leaf node. This is the text eclosed in an openening and closing XML tag.

:cName

Contains the tag name of an XML node.

Data type: C
Default: NIL

Description

The instance variable :cName holds the tag name of an XML node as a character string.

:nBeginLine

Contains the line number of an XML node.

Data type: N
Default: 0

Description

The instance variable :nBeginLine is assigned a numeric value when a TXmlDocument() object reads and parses an XML document. :nBeginLine is the line number of the XML node in the XML document.

:nType

Contains the type number of an XML node.

Data type: N
Default: HBXML_TYPE_TAG

Description

The type of an XML node is encoded as a numeric value and stored in :nType. #define constants listed in Hbxml.ch are used for this instance variable

XML node types

Constant	Value	Description
HBXML_TYPE_TAG	0	Regular XML node
HBXML_TYPE_COMMENT	1	XML comment
HBXML_TYPE_PI	2	Processing instruction
HBXML_TYPE_DIRECTIVE	3	XML directive
HBXML_TYPE_DATA	4	Parsed data section
HBXML_TYPE_CDATA	5	Unparsed data section
HBXML_TYPE_DOCUMENT	6	Document node

:oChild

Contains the child node of an XML node.

Data type: O
Default: NIL

Description

The instance variable :oChild contains a TXmlNode object representing the first XML node of the next lower level in an XML tree. If there is no lower level, or if the XML node is a leaf, :oChild contains NIL.

:oNext

Contains the next XML node in the XML document.

Data type: O
Default: NIL

Description

The instance variable :oNext contains a TXmlNode object representing the next XML node of the same nesting level in an XML tree. If there is no further XML tag, this TXMlNode object is the last node of this nesting level in the XML tree, and :oNext contains NIL.

:oParent

Contains the parent XML node.

Data type: O
Default: NIL

Description

The instance variable :oParent contains a TXmlNode object representing the XML node of the next higher level in an XML tree. If there is no higher level, :oParent contains NIL.

:oPrev

Contains the previous XML node in the XML document.

Data type: O
Default: NIL

Description

The instance variable :oPrev contains a TXmlNode object representing the previous XML node of the same nesting level in an XML tree. If there is no further XML tag, this TXMlNode object is the first node of this nesting level in the XML tree, and :oPrev contains NIL.

Methods for attributes

:getAttribute()

Retrieves an attribute from an XML tag by name.

Syntax

```
:getAttribute( <cAttribute> ) --> cValue | NIL
```

Arguments

`<cAttribute>`

This is a character string holding the name of the XML attribute to retrieve the value for.

Description

Method :getAttribute() retrieves the value of an XML attribute as a character string. If no attribute exists having *<cAttribute>* as name, the return value is NIL.

:setAttribute()

Sets a named attribute and its value for an XML tag.

Syntax

```
:setAttribute( <cAttribute>, <cValue> ) --> cValue
```

Arguments

`<cAttribute>`

This is a character string holding the name of the XML attribute to create or assign a value to.

`<cValue>`

The attribute value is set with the character string *<cValue>*.

Description

Method :setAttribute() assigns the value *<cValue>* to the XML attribute *<cAttribute>*. If the attribute does no exist, it is created.

Methods for nodes

:addBelow()

Adds a new XML node below the current XML node.

Syntax

```
:addBelow( <oTXmlNode> ) --> oChildNode
```

Arguments

<oTXmlNode>

This ia a TXmlNode object to add to the XML tree below the current XML node.

Description

Method :addBelow() adds a new XML node to the XML tree, one level below the current node. The return value is the child node of the self object.

:clone()

Clones an XML node.

Syntax

```
:clone() --> oXmlClone
```

Description

Method :clone() creates a duplicate of the self object, including all XML attributes and possible data. Child nodes are not duplicated. Method :cloneTree() is available to clone an XML node including all child nodes.

:cloneTree()

Clones an XML node with all sub-nodes.

Syntax

```
:cloneTree() --> oXmlCloneWithSubNodes
```

Description

Method :cloneTree() creates a duplicate of the self object, including all child nodes.

:insertAfter()

Inserts an XML node after the current node.

Syntax

```
:insertAfter( <oTXmlNode> ) --> oParentNode
```

Arguments

<oTXmlNode>

This ia a TXmlNode object to insert into the XML tree after the current XML node.

Description

Method :insertAfter() inserts a TXMlNode object at the same nesting level into the XML tree. The inserted XML node appears immediately after the self object.

:insertBefore()

Inserts an XML node before the current node.

Syntax

```
:insertBefore( <oTXmlNode> ) --> oPreviousNode
```

Arguments

<oTXmlNode>

This ia a TXmlNode object to insert into the XML tree before the current XML node.

Description

Method :insertBefore() inserts a TXMlNode object at the same nesting level into the XML tree. The inserted XML node appears immediately before the self object. The return value is the previous XML node.

:insertBelow()

Inserts a new XML node below the current XML node.

Syntax

```
:insertBelow( <oTXmlNode> ) --> self
```

Arguments

<oTXmlNode>

This ia a TXmlNode object to insert into the XML tree below the current XML node.

Description

Method :insertBelow() adds a new XML node to the XML tree, one level below the current node.

:nextInTree()

Retrieves the next XML node in the XML tree.

Syntax

```
:nextInTree() --> oNextNode | NIL
```

Description

Method :nextInTree() returns the next TXmlNode object in the XML tree, regardless of its nesting level or :depth(). If the method returns NIL, the TXmlNode object executing the method is the last object in the XML tree and no further TXmlNode objects exist below the current one. If the return value is a TXmlNode object, it can have a higher, lower or the same nesting level.

:unlink()

Removes an XML node from the XML tree.

Syntax

```
:unlink() --> self
```

Description

Method :unlink() removes the XmlNode object executing the method from the XML tree. The method removes this object and its sub-nodes from the tree. As a result, this TXmlNode object, and its sub-nodes, is deleted from the TXmlDocument() object maintaining the tree of TXmlNode objects.

Methods for XML code

:toString()

Creates an XML formatted character string.

Syntax

```
:toString( <nStyle> ) --> cXmlString
```

Description

:write()

Writes an XML formatted character string to a file.

Syntax

```
:write( <nFileHandle>, <nStyle> ) --> lSuccess
```

Arguments

<nFileHandle>

This is the handle of a file to write XML data to. It is returned from function FCreate().

<nStyle>

This parameter instructs the TXmlNode object how to create the XML code. #define constants listed in Hbxml.ch are used to specify *<nStyle>*:

Constants for XML code creation

Constant	Value	Description
HBXML_STYLE_INDENT	1	Indents XML nodes with one space
HBXML_STYLE_TAB	2	Indents XML nodes with tabs
HBXML_STYLE_THREESPACES	4	Indents XML nodes with three spaces
HBXML_STYLE_NOESCAPE	8	Creates unescaped characters in data sections

Description

Method :write() writes all XML nodes and data currently held by the TXmlNode object and all sub-nodes to the open file *<nFileHandle>*. The *<nStyle>* parameter can be used to indent the nodes for readability, or to leave characters in data sections unescaped.

Informational methods

:depth()

Determines the depth of an XML node.

Syntax

```
:depth() --> nDepth
```

Description

Method :depth() returns a numeric value indicating the nesting level, or depth, of the XML node represented by a TXmlNode object within an XML document.

:path()

Determines the complete path of an XML node.

Syntax

```
:path() --> cXmlPath
```

Description

Method :path() returns a character string holding the names of all XML nodes existing **before** this node plus the name of this node. The names of XML nodes are separated by a slash (/).

:toArray()

Creates an array holding all XML information of a TXmlNode object.

Syntax

```
:toArray() --> aXmlNodeInfo
```

Description

Method :toArray() is provided for informational or debugging purposes. The return value is an array with four elements. They contain all data of an XML node as stored in four instance variables:

```
{ ::nType, ::cName, ::aAttributes, ::cData }
```

Info

See also: TXmlDocument(), TXmlIterator()
Category: Object functions, xHarbour extensions
Header: hbxml.ch
Source: rtl\txml.prg
LIB: xhb.lib
DLL: xhbdll.dll

Examples

Creating XML nodes

```
// The example creates an XML document with two nodes of different
// nesting level, or depth. The inner XML node is a leaf node. Both
// XML nodes have the same attribute. Note that the starting point of
```

```
// the node creation is the root node of the XML document.

   #include "HbXml.ch"

   PROCEDURE Main
      LOCAL oXmlDoc  := TXmlDocument():new( '<?xml version="1.0"?>' )
      LOCAL oXmlNode := oXmlDoc:oRoot
      LOCAL oXmlSubNode

      oXmlSubNode := TXmlNode():new( , "AAA", ;
                                   { "id" => "1" },
                                     "NOT A LEAF NODE" )
      oXmlNode:addBelow( oXmlSubNode )

      oXmlNode    := oXmlSubNode
      oXmlSubNode := TXmlNode():new(, "BBB", {"id" => "1"}, "Leaf node" )
      oXmlNode:addBelow( oXmlsubNode )

      ? oXmlDoc:toString( HBXML_STYLE_INDENT )

      ** Output
      // <?xml version="1.0"?>
      // <AAA id="1">
      //  <BBB id="1">Leaf node</BBB>
      //  NOT A LEAF NODE</AAA>
   RETURN
```

Inserting XML nodes

```
// The example creates an XML document and inserts XML nodes
// before and after the current node.

   #include "HbXml.ch"

   PROCEDURE Main
      LOCAL oXmlDoc      := TXmlDocument():new( '<?xml version="1.0"?>'  )
      LOCAL oXmlNode     := oXmlDoc:oRoot
      LOCAL oXmlSubNode1, oXmlSubNode2

      oXmlSubNode1 := TXmlNode():new(, "AAA" )
      oXmlNode:addBelow( oXmlsubNode1 )

      oXmlNode     := oXmlSubNode1
      oXmlSubNode1 := TXmlNode():new(, "BBB", {"id" => "1"}, "Leaf node" )
      oXmlNode:addBelow( oXmlsubNode1 )

      oXmlSubNode2 := TXmlNode():new(, "BBB", {"id" => "2"}   , ;
                                               ":insertBefore()" )
      oXmlSubnode1:insertBefore( oXmlSubNode2 )

      oXmlSubNode2 := TXmlNode():new(, "BBB", {"id" => "3"}   , ;
                                               ":insertAfter()" )
      oXmlSubnode1:insertAfter( oXmlSubNode2 )

      ? oXmlDoc:toString( HBXML_STYLE_INDENT )

      ** Output
      // <?xml version="1.0"?>
      // <AAA>
      //  <BBB id="2">:insertBefore()</BBB>
      //  <BBB id="1">Leaf node</BBB>
      //  <BBB id="3">:insertAfter()</BBB>
```

```
      // </AAA>

   RETURN
```

Deleting XML nodes

```
// The example creates an XML document with a nesting level, or depth,
// of 3. The XML node at nesting level 2 is removed from the XML document.

   #include "HbXml.ch"

   PROCEDURE Main
      LOCAL oXmlDoc      := TXmlDocument():new( '<?xml version="1.0"?>'  )
      LOCAL oXmlNode     := oXmlDoc:oRoot
      LOCAL oXmlSubNode1, oXmlSubNode2

      oXmlSubNode1 := TXmlNode():new( , "AAA", { "id" => "1" }, "1st level" )
      oXmlNode:addBelow( oXmlsubNode1 )

      oXmlNode     := oXmlSubNode1
      oXmlSubNode1 := TXmlNode():new( , "BBB", { "id" => "1" }, "2nd level" )
      oXmlNode:addBelow( oXmlsubNode1 )


      oXmlSubNode2 := TXmlNode():new( , "CCC", { "id" => "1" }, "3rd level" )
      oXmlSubnode1:addBelow( oXmlSubNode2 )

      ? oXmlDoc:toString( HBXML_STYLE_INDENT )
      ** Output
      // <?xml version="1.0"?>
      // <AAA id="1">
      //  <BBB id="1">
      //   <CCC id="1">3rd level</CCC>
      //   2nd level </BBB>
      //  1st level</AAA>

      // delete this node from the XML document
      oXmlSubNode1:unlink()

      ? oXmlDoc:toString( HBXML_STYLE_INDENT )

      ** Output (only one node is left)
      // <?xml version="1.0"?>
      // <AAA id="1">1st level</AAA>

      ? oXmlSubNode1:toString( HBXML_STYLE_INDENT )

      ** Output (deleted node has a sub-node)
      // <BBB id="1">
      //  <CCC id="1">3rd level</CCC>
      // 2nd level</BBB>
   RETURN
```

Win32Bmp()

Creates a new Win32Bmp object.

Syntax

```
Win32Bmp():new()
```

Return

The function returns a new Win32Bmp object and method :new() initializes the object.

Description

Objects of class Win32Bmp are used in conjunction with objects of the Win32Prn class for printing bitmap images using the Windows Graphical Device Interface (GDI). For this reason, an application using Win32Bmp objects must be created as GUI application. Text mode applications, or console applications, cannot use the Win32Bmp class.

A Win32Bmp object is capable of loading a bitmap file into memory with its :loadFile() method. After the bitmap image is loaded, it can be printed with the aid of a Win32Prn() object using method :draw().

Instance variables

:rect

Array with X/Y coordinates where to draw the bitmap on the printer.

Data type: A
Default: {0,0,0,0}

Description

The instance variable :rect holds an array of four numeric elements. :rect stores the second parameter of method :draw().

:bitmap

Character string holding the data of the bitmap image.

Data type: C
Default: ""

Description

The instance variable :bitmap holds the data of a loaded bitmap file as a character string. A bitmap is loaded into memory with method :loadFile().

:fileName

Character string holding the file name of the bitmap image.

Data type: C
Default: ""

Description

The instance variable :fileName holds the name of a loaded bitmap file as a character string. A bitmap is loaded into memory with method :loadFile().

Methods

:draw()

Draws the bitmap image to a printer.

Syntax

```
:draw( <oWin32Prn>, <aRectangle> ) --> lSuccess
```

Arguments

`<oWin32Prn>`

This parameter must be a Win32Prn() object.

`<aRectangle> := { <nX1>, <nY1>, <nX2>, <nY2> }`

This is a one dimensional array of four elements. It holds numeric values specifying the X/Y coordinates of the upper-left (*<nX1>*, *<nY1>*) and lower-right (*<nX2>*, *<nY2>*) position on the printer wher the bitmap image is drawn. The coordinates must be specified in pixel units.

Description

Method :draw() sends a loaded bitmap image to the printer specified with *<oWin32Prn>*. The coordinates passsed with *<aRectangle>* are printer coordinates. i.e. they tell the Win32Prn object, where to print the bitmap on paper.

If the rectangle *<aRectangle>* does not match the size of the bitmap image, the image is automatically scaled and/or transformed to the size of *<aRectangle>*.

:loadFile()

Loads a bitmap image file into memory.

Syntax

```
:loadFile( <cFileName> ) --> lSuccess
```

Arguments

`<cFileName>`

This is a character string holding the name of the bitmap file to load into memory. It must be a full qualified file name including file extension. If *<cFileName>* does not include path information, the bitmap file is searched in the current directory.

Description

Method :loadFile() loads the specified bitmap file into memory and returns a logical value indicating success of the operation. When the file is successfully loaded, the return value is .T. (true). The bitmap can then be output with method :draw().

Info

See also: Win32Prn()
Category: Object functions, Printer functions, xHarbour extensions
Source: rtl\win32prn.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example outlines the steps required for printing
// a bitmap image

  PROCEDURE Main
     LOCAL cFileName, oWin32Bmp, oWin32Prn

      oWin32Prn := Win32Prn():new()  // default printer object
      IF .NOT. oWin32Prn:create()    // create device context
         Alert( "Unable to create device context for printer" )
         QUIT
      ENDIF

      cFileName := "TestImage.bmp"

      oWin32Bmp := Win32Bmp():new()  // load bitmap file into memory
      IF .NOT. oWin32Bmp:loadFile( cFileName )
         Alert( "Unable to load bitmap file: " + cFileName )
         QUIT
      ENDIF

      // print bitmap image
      oWin32Bmp:draw( oWin32Prn,  { 200, 400, 2000, 1500 } )

      // release GDI system resources of printer
      oWin32Prn:destroy()
   RETURN
```

Win32Prn()

Creates a new Win32Prn object.

Syntax

```
Win32Prn():new( [<cPrinterName>] ) --> oWin32Prn
```

Arguments

<cPrinterName>

This is a character string holding the name of the printer to use for printing. It defaults to the return value of GetDefaultPrinter().

Return

The function returns a new Win32Prn object and method :new() initializes the object.

Description

Objects of class Win32Prn provide the means for printing using the Windows Graphical Device Interface (GDI). For this reason, an application using Win32Prn objects must be created as GUI application. Text mode applications, or console applications, cannot use the Win32Prn class.

After a Win32Prn object is initialized with the :new() method, it must request system recources with its :create() method. Before this method is called, a print job can be configured with instance variables of a Win32Prn object. They define the paper format, print orientation or number of copies to print. A Win32Prn object can only be used for printing when its :create() method is called.

A print job is started with the method :startDoc() after which various print output methods can be called. They define the actual output that appears on paper. :startDoc() defines the document name to print for the Windows GDI printer spooler and initiates a print job. The print job must be finished with method :endDoc(). Print output is sent to the printer after :endDoc() is called.

When a print job is finished, a Win32Prn object must release GDI system resources with its :destroy() method. As a result, the following programming pattern must be applied for successful GDI printing:

```
oPrinter := Win32Prn():new()      // creates the object
oPrinter:create()                 // requests system resources
oPrinter:startDoc()               // begins a document
oPrinter:textOut( "Hello World" ) // print output methods
oPrinter:endDoc()                 // ends document definition and prints
oPrinter:destroy()                // releases system resources
```

Coordinate system for GDI printing

The coordinate system has its origin in the upper left corner of a page. X and Y coordinates are measured in pixels. The X coordinate increases from left to right (horizontal coordinate), and the Y coordinate increases from the top to the bottom of a page (vertical coordinate).

To support the DOS Clipper dot-matrix printing style, the Win32Prn class has methods allowing for a row/column oriented positioning of print output. These methods emulate functions of the same name, like :pRow(), :pCol() or :setPrc().

Fonts

The default font is a DOS Clipper compatible fixed font (Courier New). A list of all available fonts can be obtained with method :getFonts(). Text is printed using the currently selected font, which is defined with method :setFont().

Printing text

Text is output at the current print position with method :textOut(), or at a particular X/Y coordinate with method :textOutAt(). The text color can be selected with method :setColor().

Printing lines

A variety of methods are available for drawing basic graphical elements, such as :arc(), :box() or :line(). The color and line style for these graphical elements is defined with method :setPen(). This method creates a pen capable of drawing lines.

Colors

Colors for GDI printing must be provided as numeric RGB color values. The console mode SetColor() strings cannot be used as color definition. RGB color values are calculated from three numeric values between 0 and 255. They define the intensity for Red, Green and Blue. An RGB color value can be calulated as follows:

```
FUNCTION RGB( nRed, nGreen, nBlue )
RETURN ( nRed + ( nGreen * 256 ) + ( nBlue * 65536 ) )
```

Color and text settings

:bitmapsOk

Indicates if bitmap images can be printed.

Data type: L
Default: .F.

Description

The instance variable :bitmapsOk indicates if the Win32Prn object is capable of printing bitmap images. The value of :bitmapOk is only relevant after the :create() method is called. :bitmapsOk contains .T. (true) when bitmaps can be printed, otherwise .F. (false).

:bkColor

Numeric background color value.

Data type: N
Default: NIL

Description

The instance variable :bkColor contains the numeric value for the background color defined with method :setColor(). If :setColor() is not called, the background color is NIL.

:numColors

Number of colors supported by the GDI device (printer).

Data type: N
Default: 1

Description

The instance variable :numColors contains a numeric value indicating the number of colors supported by the GDI printer device. This value is only relevant after the :create() method is called.

:penColor

Numeric color value of current pen.

Data type: N
Default: NIL

Description

The instance variable :penColor contains the numeric color value for the current pen defined with method :setPen(). If :setPen() is not called, the pen color is NIL.

:penStyle

Numeric style of current pen.

Data type: N
Default: NIL

Description

The instance variable :penStyle contains the numeric style for the current pen defined with method :setPen(). If :setPen() is not called, the pen style is NIL.

:penWidth

Numeric width of current pen.

Data type: N
Default: NIL

Description

The instance variable :penWidth contains the numeric width for the current pen defined with method :setPen(). If :setPen() is not called, the pen width is NIL.

:posX

Current position of print head in X direction (horizontal).

Data type: N
Default: 0

Description

The instance variable :posX contains a numeric value indicating the current print position on paper in X direction (horizontal). The value is updated with each print output method that prints text or starts a new line or page. :posX contains the position in device units (pixel). The X position can be changed with method :setPos().

:posY

Current position of print head in Y direction (vertical).

Data type: N
Default: 0

Description

The instance variable :posY contains a numeric value indicating the current print position on paper in Y direction (vertical). The value is updated with each print output method that prints text or starts a

new line or page. :posY contains the position in device units (pixel). The Y position can be changed with method :setPos().

:textAlign

Numeric value for text alignment.

Data type: N
Default: NIL

Description

The instance variable :textAlign contains the numeric value for the text alignment defined with method :setColor(). If :setColor() is not called, :textAlign is NIL.

:textColor

Numeric value for text color (foreground).

Data type: N
Default: NIL

Description

The instance variable :textColor contains the numeric value for the foreground color of text defined with method :setColor(). If :setColor() is not called, the foreground color is NIL.

Font settings

:charHeight

Numeric height of characters for the current font.

Data type: N
Default: 0

Description

The instance variable :charHeight contains the numeric height in pixels for a character of the current font. By default, method :create() selects "Courier New" as current font.

The current font can be changed with method :setFont() after :create() is called.

:charWidth

Numeric width of characters for the current font.

Data type: N
Default: 0

Description

The instance variable :charWidth contains the numeric width in pixels for a character of the current font. By default, method :create() selects "Courier New" as current font.

The current font can be changed with method :setFont() after :create() is called.

Note: the character width is an average width when a proportional font is selected.

:fontName

Character string describing the currently selected font.

Data type: C
Default: ""

Description

The instance variable :fontName contains the name of the current font as a character string. By default, method :create() selects "Courier New" as current font.

The current font can be changed with method :setFont() after :create() is called.

:fontPointSize

Numeric point size of the current font.

Data type: N
Default: 12

Description

The instance variable :fontPointSize contains the numeric point size of the current font. By default, method :create() selects "Courier New" with 12 points as current font.

The current font can be changed with method :setFont() after :create() is called.

:fontWidth

Array describing the width of the current font.

Data type: N
Default: {0,0}

Description

The instance variable :fontWidth contains an array with two elements holding numeric integer values. The font width is defined with method :setFont() after :create() is called. It is calculated in the unit "Characters Per Inch" (CPI) by dividing the second element of :fontWidth by the first element. For example:

```
::fontWidth := { 3, 50 }

nCPI := ::fontWidth[2]/::fontWidth[1]

? nCPI  // 16.67 characters per inch
```

When :fontWidth contains the array {0,0} the default font width is selected.

Note: when the second element contains a negative value, fixed character spacing is enforced, even if the font is a proportional font.

:lineHeight

Numeric line height of current font.

Data type: N
Default: 0

Description

The instance variable :lineHeight contains the numeric height in pixels for a text line of the current font. By default, method :create() selects "Courier New" as current font.

The current font can be changed with method :setFont() after :create() is called.

:pixelsPerInchX

Number of pixels per inch in X direction (horizontal)

Data type: N
Default: NIL

Description

The instance variable :pixelsPerInchX contains a numeric value indicating number of pixels per Inch in X direction (horizontal). This value is only relevant after method :create() is called and can vary between different GDI printers.

:pixelsPerInchY

Number of pixels per inch in Y direction (vertical)

Data type: N
Default: NIL

Description

The instance variable :pixelsPerInchY contains a numeric value indicating number of pixels per Inch in Y direction (vertical). This value is only relevant after method :create() is called and can vary between different GDI printers.

:setFontOk

Indicates success of setting a font.

Data type: L
Default: .F.

Description

The instance variable :setFontOk contains a logical value indicating success of method :setFont(). A font is available for printing when :setFontOk contains .T. (true). By default, method :create() selects "Courier New" as current font.

Page metrics

:bottomMargin

Numeric bottom margin of page.

Data type: N
Default: 0

Description

The instance variable :bottomMargin contains a numeric value indicating the last position on the page that can be printed to. It is measured from the top of the page in pixel units. The difference between bottom margin and :pageHeight is the part at the bottom of a page that cannot be printed to.

:leftMargin

Numeric left margin of a page.

Data type: N
Default: 0

Description

The instance variable :leftMargin contains a numeric value indicating the first position on the left side of a page that can be printed to. It is measured in pixel units. The area between position 0 and :leftMargin cannot be printed to.

:pageHeight

Numeric page dimension in Y direction (vertical or height).

Data type: N
Default: 0

Description

The instance variable :pageHeight contains a numeric value indicating the physical dimension of a sheet of paper in Y direction (vertical). The unit is pixels.

:pageWidth

Numeric page dimension in X direction (horizontal or width).

Data type: N
Default: 0

Description

The instance variable :pageWidth contains a numeric value indicating the physical dimension of a sheet of paper in X direction (horizontal). The unit is pixels.

:rightMargin

Numeric right margin of a page.

Data type: N
Default: 0

Description

The instance variable :rightMargin contains a numeric value indicating the rightmost position on the page that can be printed to. It is measured from the left of the page in pixel units. The difference between right margin and :pageWidth is the part on the right side of a page that cannot be printed to.

:topMargin

Numeric top margin of a page.

Data type: N
Default: 0

Description

The instance variable :topMargin contains a numeric value indicating the first position at the top of a page that can be printed to. It is measured in pixel units. The area between position 0 and :topMargin cannot be printed to.

Print job configuration

:binNumber

Numeric paper bin to use for printing.

Data type: N
Default: 0

Description

A numeric value can be assigned to :binNumber before method :create() is called. It selects the paper bin to use for the print job. #define constants are available in the file WinGdi.ch that can be used for :binNumber. If :binNumber is not set, the default paper bin is used.

Constants for paper bin selection

Constant	Value
DMBIN_FIRST	1
DMBIN_UPPER	1
DMBIN_ONLYONE	1
DMBIN_LOWER	2
DMBIN_MIDDLE	3
DMBIN_MANUAL	4
DMBIN_ENVELOPE	5
DMBIN_ENVMANUAL	6
DMBIN_AUTO	7
DMBIN_TRACTOR	8
DMBIN_SMALLFMT	9
DMBIN_LARGEFMT	10
DMBIN_LARGECAPACITY	11

DMBIN_CASSETTE	14
DMBIN_FORMSOURCE	15

When :binNumber is changed during a print job, method :startPage() must be called so that the new setting becomes valid on a new page.

:copies

Number of copies to print.

Data type: N
Default: 1

Description

A numeric value can be assigned to :copies before method :create() is called. It defines the number of copies to print with the print job.

:formType

Numeric paper format to print on.

Data type: N
Default: 0

Description

A numeric value can be assigned to :formType before method :create() is called. It selects the paper format for the print job. Numerous #define constants are available in the file WinGdi.ch that can be used for :formType. They begin with the prefix DMPAPER_. A selection of common paper formats is listed below. If :formType is not set, the default paper format of the printer is used.

Constants for common paper formats

Constant	Value	Description
DMPAPER_LETTER	1	US Letter format 8 1/2 x 11 inch
DMPAPER_LEGAL	5	US Legal format 8 1/2 x 14 inch
DMPAPER_A3	8	DIN A3 format 297 x 420 mm
DMPAPER_A4	9	DIN A4 format 210 x 297 mm

When :formType is changed during a print job, method :startPage() must be called so that the new setting becomes valid on a new page.

:landscape

Logical print orientation.

Data type: L
Default: .F.

Description

The instance variable :landscape can be set to .T. (true) before method :create() is called. This selects Landscape print orientation. When :landscape is .F. (false), which is the default, print orientation is Portrait.

When :landscape is changed during a print job, method :startPage() must be called so that the new setting becomes valid on a new page.

Printer configuration

:havePrinted

Logical flag indicating if text or bitmap was printed.

Data type: L
Default: .F.

Description

The instance variable :havePrinted is set to .F. (false) when method :create() is called, indicating that no print output has ocurred with the printer yet. Print output methods like :textOut() or :drawBitmap() set :havePrinted to .T. (true).

:hPrinterDc

Numeric handle to the printer device context.

Data type: N
Default: 0

Description

The instance variable :hPrinterDc contains a numeric value which is a handle to the printer device context after method :create() is called. :hPrinterDc can be used with Windows API functions when the device context of the GDI device is required.

:printerName

Character string describing the currently selected printer.

Data type: C
Default: ""

Description

The instance variable :printerName contains the name of the printer as a character string, as it is passed to method :new().

:printing

Logical flag indicating if a print job is in progress.

Data type: L
Default: .F.

Description

The instance variable :printing contains .T. (true) when method :startDoc() is called, indicating that a print job has started. :printing is set to .F. (false) when the print job has ended with method :endDoc().

Clipper DOS compatibility

:inch_To_PosX()

Calculates a horizontal X position from inches.

Syntax

```
:inch_To_PosX( <nInches> ) --> nPosX
```

Arguments

<nInch>

This is a numeric value indicating inches.

Description

The method calculates a horizontal X position in pixels from inches and returns the result as a numeric value. This is required when the print position must be changed with method :setPos(), which accepts only pixel coordinates.

:inch_To_PosY()

Calculates a vertical Y position from inches.

Syntax

```
:inch_To_PosY( <nInches> ) --> nPosY
```

Arguments

<nInch>

This is a numeric value indicating inches.

Description

The method calculates a vertical Y position in pixels from inches and returns the result as a numeric value. This is required when the print position must be changed with method :setPos(), which accepts only pixel coordinates.

:mm_To_PosX()

Calculates a horizontal X position from millimeters.

Syntax

```
:mm_To_PosX( <nMilliMeters> ) --> nPosX
```

Arguments

<nMilliMeters>

This is a numeric value indicating millimeters.

Description

The method calculates a horizontal X position in pixels from millimeters and returns the result as a numeric value. This is required when the print position must be changed with method :setPos(), which accepts only pixel coordinates.

:mm_To_PosY()

Calculates a vertical Y position from millimeters.

Syntax

```
:mm_To_PosY( <nMilliMeters> ) --> nPosY
```

Arguments

`<nMilliMeters>`

This is a numeric value indicating millimeters.

Description

The method calculates a vertical Y position in pixels from millimeters and returns the result as a numeric value. This is required when the print position must be changed with method :setPos(), which accepts only pixel coordinates.

:maxCol()

Number of fixed font characters that fit into one line

Syntax

```
:maxCol() --> nMaxCol
```

Description

Method :maxCol() determines the maximum number of characters that can be printed in one line and returns the result as a numeric value. This value is only exact for fixed fonts. When a proportional font is selected, the result is calculated from an average character width, and can only be seen as an estimate.

:maxRow()

Number of text lines fit on one page.

Syntax

```
:maxRow() --> nMaxRow
```

Description

Method :maxRow() determines the maximum number of lines that can that can be printed on one sheet of paper and returns the result as a numeric value.

:pCol()

Emulates the PCol() function.

Syntax

```
:pCol() --> nPrinterColumn
```

Arguments

Description

The method emulates the PCol() function which returns the current column position of the print head. This is only relevant for fixed fonts.

:pRow()

Emulates the PRow() function.

Syntax

```
:pRow() --> nPrinterRow
```

Arguments

Description

The method emulates the PRow() function which returns the current row position of the print head.

:setPrc()

Changes the current print position using row/column coordinates.

Syntax

```
:setPrc( <nRow>, <nCol> ) --> NIL
```

Arguments

`<nRow>`

This is a numeric value specifying the new row position of the printer (Y direction).

`<nCol>`

A numeric value specifying the new column position of the printer (X direction).

Description

The method emulates the SetPrc() function which changes the current print position based on row/column coordinates. The method converts row/column coordinates to graphical X/Y coordinates (pixels) and calls :setPos(). The return value is always NIL.

Document methods

:endDoc()

Terminates the current print job.

Syntax

```
:endDoc( [<lAbortDoc>] ) --> lSuccess
```

Arguments

`<lAbortDoc>`

This logical parameter defaults to .F. (false) which informs the spooler that print output can be sent to the printer. When .T. (true) is passed, all spooled print output is discarded.

Description

Method :endDoc() must be called as the last method of a print job. It signals the spooler that the xHarbour application produces no more print output. The spooler then sends print output to the printer, unless *<lAbortDoc>* is set to .T. (true).

The method returns a logical value indicating success of the operation.

:endPage()

Marks the end of a page.

Syntax

```
:endPage( [<lNewPage>] ) --> lSuccess
```

Arguments

`<lNewPage>`

This logical parameter defaults to .T. (true) which informs the spooler that print output will continue on the next page. Set *<lNewPage>* to .F. (false) when this is the last page of the print job.

Description

Method :endPage() informs the spooler that no more print output will occur on the current page. The method calls internally :startPage() to begin printing on a new page, unless *<lNewPage>* is set to .F. (false).

:newLine()

Moves the current print position to the beginning of the next line.

Syntax

```
:newLine() --> nPosY
```

Description

Method :newLine() changes the current print position by setting the X coordinate (horizontal) to :leftMargin and adding :lineHeight to the current Y position (vertical). The return value is the new Y position in pixel.

:newPage()

Starts a new page.

Syntax

```
:newPage() --> lSuccess
```

Description

Method :newPage() instructs the spooler that print output will occur on the next page. The method returns a logical value indicating success of the operation.

:startDoc()

Instructs the spooler to begin with a new document.

Syntax

```
:startDoc( [<cDocumentName>] ) --> lSuccess
```

Arguments

<cDocumentName>

This is an optional character string holding the name of the document to begin with. The character string is displayed in the printer spooler window. If *<cDocumentName>* is omitted, it is created from the xHarbour application name and a time stamp.

Description

Method :startDoc() must be called after :create(). It istructs the spooler to begin with a new document and capture subsequent print output. The print output is sent to the printer after the document is closed with method :endDoc().

:startPage()

Starts a new page and reconfigures page settings.

Syntax

```
:startPage() --> lSuccess
```

Description

Method :startPage() begins a new page and reconfigures the page settings when they are changed during a print job. These settings are :binNumber, :formType, :landscape, :setDuplexType() and :setPrintQuality().

Font and text metrics

:getCharHeight()

Returns the average character height of the current font in pixels.

Syntax

```
:getCharHeight() --> nPixels
```

Description

The method returns the average character height of the current font as a numeric value. The unit is pixels.

:getCharWidth()

Returns the average character width of the current font in pixels.

Syntax

```
:getCharWidth() --> nPixels
```

Description

The method returns the average character width of the current font as a numeric value. The unit is pixels.

:getTextHeight()

Returns the height of a character string in pixels.

Syntax

```
:getTextHeight( <cString> ) --> nPixelsHeight
```

Arguments

```
<cString>
```

This is a character string to determine the height for.

Description

The method :getTextHeight() determines the maximum height of a character string in pixels and returns the result as a numeric value. For example, the string "Troja" has a bigger height than the string "io".

:getTextWidth()

Returns the width of a character string in pixels.

Syntax

```
:getTextWidth( <cString> ) --> nPixelsWidth
```

Arguments

```
<cString>
```

This is a character string to determine the width for.

Description

The method :getTextHeight() determines the maximum width of a character string in pixels and returns the result as a numeric value. This is especially useful when a proportional font is selected where different characters require a different amount of space. The letter "i" needs less space than "W", for example.

Font methods

:bold()

Queries or changes the font weight for bold printing.

Syntax

```
:bold( [<nNewFontWeight>] ) --> nOldFontWeight
```

Arguments

`<nNewFontWeight>`

This is a numeric value specifying the new font weight for printing. The following #define constants listed in WinGdi.ch can be used:

Constants for font weights

Constant	Value
FW_DONTCARE	0
FW_THIN	100
FW_EXTRALIGHT	200
FW_LIGHT	300
FW_NORMAL	400
FW_MEDIUM	500
FW_SEMIBOLD	600
FW_BOLD	700
FW_EXTRABOLD	800
FW_HEAVY	900

Description

Method :bold() returns the currently defined font weight used for bold printing as a numeric value. When *<nNewFontWeight>* is specified, the new font weight is set during printing.

:charSet()

Queries or changes the character set for text printing.

Syntax

```
:charSet( [<nNewCharSet>] ) --> nOldCharSet
```

Arguments

`<nNewCharSet>`

This is a numeric value specifying the character set for printing. The following #define constants listed in WinGdi.ch can be used:

Constants for character sets

Constant	Value
ANSI_CHARSET	0
DEFAULT_CHARSET *)	1
SYMBOL_CHARSET	2
SHIFTJIS_CHARSET	128
HANGEUL_CHARSET	129
HANGUL_CHARSET	129
GB2312_CHARSET	134
CHINESEBIG5_CHARSET	136
OEM_CHARSET	255
JOHAB_CHARSET	130
HEBREW_CHARSET	177
ARABIC_CHARSET	178
GREEK_CHARSET	161
TURKISH_CHARSET	162
VIETNAMESE_CHARSET	163
THAI_CHARSET	222

EASTEUROPE_CHARSET	238
RUSSIAN_CHARSET	204

**) default*

Description

Method :chrSet() returns the currently defined character set for text as a numeric value. When *<nCharSet>* is specified, the new character set is set during printing.

:getFonts()

Queries information about available fonts.

Syntax

```
:getFonts() --> aFontInfo
```

Description

Method :getfonts() collects information about all available fonts and returns it in form of a two dimensional array. The columns of the array contain the following data:

Font information array

Column No	Data type	Description
1	Character	Name of the font
2	Logical	.T. == Fixed font, .F. == Proportional font
3	Logical	.T. == True Type font, .F. == Bitmap font
4	Numeric	Character set required

Fonts listed in the font information array can be used for printing. A font is selected with method :setFont().

:italic()

Queries or sets italic printing.

Syntax

```
:italic( [<lNewSetting>] ) --> lOldSetting
```

Arguments

<lNewSetting>

If a logical value is passed, it defines the new *Italic* setting. .T. (true) switches *Italic* on, and .F. (false) switches it off.

Description

Method :italic() returns the currently defined *Italic* setting used for printing text as a logical value. When *<lNewSetting>* is specified, the new *Italic* mode is set during printing.

:underLine()

Queries or sets underlined printing.

Syntax

```
:underLine( [<lNewSetting>] ) --> lOldSetting
```

Arguments

<lNewSetting>

If a logical value is passed, it defines the new Underline setting. .T. (true) switches Underline on, and .F. (false) switches it off.

Description

Method :underLine() returns the currently defined Underline setting used for printing text as a logical value. When *<lNewSetting>* is specified, the new Underline mode is set during printing.

:setDefaultFont()

Sets the default font.

Syntax

```
:setDefaultFont() --> lSuccess
```

Description

Method :setDefaultFont() selects the "Courier New" font as current font.

:setFont()

Selects the current font.

Syntax

```
:setFont( <cFontName>    , ;
         [<nPointSize>] , ;
         [<nFontWidth>] , ;
         [<nFontWeight>], ;
         [<lUnderline>] , ;
         [<lItalic>]    , ;
         [<nCharSet>]     ) --> lSuccess
```

Arguments

<cFontName>

This is a character string holding the name of the font to select as current. It is case sensitive.

<nPointSize>

This is a numeric value indicating the font size in points. If omitted, a default font size is used.

<nFontWidth>

This can be a numeric value indicating the font width in pixels. If omitted, a default font width is used.

Alternatively, a two element array can be passed. It is used to calculate the font width in the unit "Characters Per Inch" (CPI) by dividing the second element by the first element. For example:

```
aFontWidth := { 3, 50 }

nCPI := aFontWidth[2]/aFontWidth[1]

? nCPI  // 16.67 characters per inch
```

When the array {0,0} is passed, the default font width is selected.

Note: when the second element contains a negative value, fixed character spacing is enforced, even if the font is a proportional font.

`<nFontWeight>`

This is a numeric value indicating the font weight. If omitted, a default font weight is used. #define constants are available in the file WinGdi.ch that can be used for *<nFontWeight>*:

Constants for font weights

Constant	Value
FW_DONTCARE	0
FW_THIN	100
FW_EXTRALIGHT	200
FW_LIGHT	300
FW_NORMAL	400
FW_MEDIUM	500
FW_SEMIBOLD	600
FW_BOLD	700
FW_EXTRABOLD	800
FW_HEAVY	900

`<lUnderline>`

This is a logical value specifying the underlined mode of the font. The value .T. (true) switches underlined on, and .F. (false) switches it off. When omitted, the current underlined mode of the Win32Prn object is used.

`<lItalic>`

This is a logical value specifying the *italic* mode of the font. The value .T. (true) switches *italic* on, and .F. (false) switches it off. When omitted, the current *italic* mode of the Win32Prn object is used.

`<nCharSet>`

This is a numeric value indicating the character set. If omitted, the current character set of the Win32Prn object is used. #define constants are available in the file WinGdi.ch that can be used for *<nCharSet>*:

Constants for character sets

Constant	Value
ANSI_CHARSET	0
DEFAULT_CHARSET *)	1
SYMBOL_CHARSET	2
SHIFTJIS_CHARSET	128
HANGEUL_CHARSET	129
HANGUL_CHARSET	129
GB2312_CHARSET	134
CHINESEBIG5_CHARSET	136
OEM_CHARSET	255
JOHAB_CHARSET	130
HEBREW_CHARSET	177
ARABIC_CHARSET	178
GREEK_CHARSET	161
TURKISH_CHARSET	162
VIETNAMESE_CHARSET	163
THAI_CHARSET	222
EASTEUROPE_CHARSET	238

RUSSIAN_CHARSET	204

**) default*

Description

Method :setFont() selects the current font. This font is used by methods :textOut() or :textOutAt() for printing text. When the font with the desired properties exists and can be selected, the method returns .T. (true). Note that the operating system may select a font that does not match the passed parameters exactly, but chooses a "best fit" font instead.

Use method :getFonts() to obtain a font information array holding data of all fonts available.

Printer methods

:getDeviceCaps()

Queries device capabilities.

Syntax

```
:getDeviceCaps( <nCaps> ) --> nDeviceCaps
```

Arguments

`<nCaps>`

This is a numeric value indicating the device capability to query.

Description

There are numerous device capabilities that can be queried. The file WinGdi.ch contains #define constants for the parameter *<nCaps>* and explanations for device capabilities. Note that :getDeviceCaps() requires method :create() be called for querying device capabilities.

:setDuplexType()

Queries or changes the duplex printing mode.

Syntax

```
:setDuplexType( [<nNewDuplex>] ) --> nOldDuplex
```

Arguments

`<nNewDuplex>`

This is a numeric value specifying the new duplex mode for printing. The following #define constants listed in WinGdi.ch can be used:

Constants for duplex printing

Constant	Value	Description
DMDUP_SIMPLEX *)	1	Normal (nonduplex) printing.
DMDUP_VERTICAL	2	Short-edge binding, that is, the long edge of the page is horizontal.
DMDUP_HORIZONTAL	3	Long-edge binding, that is, the long edge of the page is vertical.

**) default*

Description

Method :setDuplexType() returns the currently defined mode for duplex printing as a numeric value. When *<nNewDuplex>* is specified, :setDuplexType() must either be called before method :create(), or method :startPage() must be called afterwards to activate the changed duplex printing mode on the next page.

:setPrintQuality()

Queries or changes the printing quality.

Syntax

```
:setPrintQuality( [<nNewPrintQuality>] ) --> nOldPrintQuality
```

Arguments

`<nNewPrintQuality>`

This is a numeric value specifying the new printing quality. The following #define constants listed in WinGdi.ch can be used:

Constants for the printing quality

Constant	Value
DMRES_DRAFT	-1
DMRES_LOW	-2
DMRES_MEDIUM	-3
DMRES_HIGH	-4

Description

Method :setPrintQuality() returns the currently defined setting for the printing quality as a numeric value. When *<nNewPrintQuality>* is specified, :setPrintQuality() must either be called before method :create(), or method :startPage() must be called afterwards to activate the changed printing quality on the next page.

Print output methods

:arc()

Prints an arc.

Syntax

```
:arc( <nX1>, <nY1>, <nX2>, <nY2> ) --> lSuccess
```

Arguments

`<nX1> and <nY1>`

These two numeric values are the X and Y coordinates of the upper-left corner of a rectangle surrounding the arc.

`<nX2> and <nY2>`

These two numeric values are the X and Y coordinates of the lower-right corner of a rectangle surrounding the arc.

Description

Method :arc() accepts the coordinates of a bounding rectangle and draws an elliptical arc inside the rectangle using the current pen. The pen must be selected previously with method :setPen().

The method returns a logical value indicating success of the operation.

:box()

Prints a box.

Syntax

```
:box( <nX1>    , ;
      <nY1>    , ;
      <nX2>    , ;
      <nY2>    , ;
     [<nWidth>], ;
     [<nHeight>] ) --> lSuccess
```

Arguments

`<nX1> and <nY1>`

These two numeric values are the X and Y coordinates of the upper-left corner of the box to print.

`<nX2> and <nY2>`

These two numeric values are the X and Y coordinates of the lower-right corner of the box to print.

`<nWidth> and <nHeight>`

If both numeric parameters are passed, the box is drawn with rounded edges. *<nWidth>* defines the radius in pixels for a rounded edge in horizontal, and *<nHeight>* in vertical direction.

Description

Method :box() accepts the coordinates of a rectangle and draws the rectangle using the current pen. The pen must be selected previously with method :setPen(). If both parameters *<nWidth>* and *<nHeight>* are specified, the box is drawn with rounded edges.

The method returns a logical value indicating success of the operation.

:drawBitmap()

Prints a bitmap image.

Syntax

```
:drawBitmap( <oWin32Bmp> ) --> lSuccess
```

Arguments

`<oWin32Bmp>`

This is a Win32Bmp() object maintaining a bitmap image.

Description

Method :drawBitmap() prints a bitmap image file loaded by the passed Win32Bmp object. The coordinates for the bitmap image on paper are specified with the Win32Bmp object.

:ellipse()

Prints an ellipse.

Syntax

```
:ellipse( <nX1>, <nY1>, <nX2>, <nY2> ) --> lSuccess
```

Arguments

`<nX1> and <nY1>`

These two numeric values are the X and Y coordinates of the upper-left corner of a rectangle surrounding the ellipse.

`<nX2> and <nY2>`

These two numeric values are the X and Y coordinates of the lower-right corner of a rectangle surrounding the ellipse.

Description

Method :ellipse() accepts the coordinates of a bounding rectangle and draws an ellipse inside the rectangle using the current pen. The pen must be selected previously with method :setPen().

The method returns a logical value indicating success of the operation.

:fillRect()

Prints a filled rectangle.

Syntax

```
:fillRect( <nX1>    , ;
           <nY1>    , ;
           <nX2>    , ;
           <nY2>    , ;
           <nRGBColor>) --> NIL
```

Arguments

`<nX1> and <nY1>`

These two numeric values are the X and Y coordinates of the upper-left corner of a rectangle to fill with a color.

`<nX2> and <nY2>`

These two numeric values are the X and Y coordinates of the lower-right corner of a rectangle to fill with a color.

`<nRGBColor>`

This is a numeric RGB color value defining the fill color.

Description

Method :fillRect() accepts the coordinates of a rectangle and fills it with the color *<nRGBColor>*. The return value is NIL.

:line()

Prints a line

Syntax

```
:line( <nX1>, <nY1>, <nX2>, <nY2> ) --> lSuccess
```

Arguments

`<nX1> and <nY1>`

These two numeric values are the X and Y coordinates of the start point of a line.

`<nX2> and <nY2>`

These two numeric values are the X and Y coordinates of the end point of a line.

Description

Method :line() accepts the coordinates of the start and en point for a line and draws the line using the current pen. The pen must be selected previously with method :setPen().

The method returns a logical value indicating success of the operation.

:setBkMode()

Selects the background color mix mode.

Syntax

```
:setBkMode( <nMode> ) --> lSuccess
```

Arguments

`<nMode>`

This is a numeric value specifying the background color mix mode for printing. The following #define constants listed in WinGdi.ch can be used:

Constants for the background color mix mode

Constant	Value	Description
TRANSPARENT	1	Background is filled with the current background color before printing.
OPAQUE	2	Background remains untouched.

Description

Method :setBkMode() defines the background color mix mode to use when a print output method overwrites previous print output.

The method returns a logical value indicating success of the operation.

:setColor()

Defines text color and alignment.

Syntax

```
:setColor( <nClrText>, [<nClrPane>], [<nAlign>} ) --> NIL
```

Arguments

`<nClrText>`

This is a numeric RGB color value defining the text color (foreground).

`<nClrPane>`

This is an optional numeric RGB color value defining the pane color (background).

`<nAlign>`

This is an optional numeric value specifying the text alignment at the current print position. The following values can be used:

Values for text alignment

Value	Description
0 *)	Text is printed left aligned to the current print position
1	Text is printed right aligned to the current print position
2	Text is printed centered on the current print position

**) default*

Two constants for horizontal and vertical alignment can be added to define horizontal and vertical text alignment with one parameter.

Description

Method :setColor() defines foreground and background color plus alignment for printing text strings. The color values must be defined as numeric RGB colors. The defined colors and text alignment are used for text printing methods like :textOut() or :textOutAt().

The return value of :setColor() is always NIL.

:setPen()

Defines the pen for drawing lines.

Syntax

```
:setPen( <nStyle>, <nWidth>, <nColor> ) --> nPenHandle
```

Arguments

`<nStyle>`

This numeric parameter defines the pen style used for drawing lines. #define constants are available in the file WinGdi.ch for setting the pen style:

Constants for pen styles

Constant	Value	Description
PS_SOLID	0	Pen draws solid line
PS_DASH	1	Pen draws dashed line (-------)
PS_DOT	2	Pen draws dotted line (.......)
PS_DASHDOT	3	Pen draws dashes and dots (_._._._)
PS_DASHDOTDOT	4	Pen draws dashes and two dots (_.._.._)
PS_NULL	5	Pen is invisible

`<nWidth>`

This numeric parameter defines the line width in pixels. When set to zero, the pen draws a 1 pixel wide line, regardless of any transformation.

<nColor>

This is a numeric RGB color value defining the line color.

Description

Method :setPen() defines the current pen used for drawing lines. Lines are drawn by methods producing basic graphical elements such as :arc(), :box() or :line(). The method returns a numeric handle to the current pen. This can be used with other Windows GDI functions.

:setPos()

Queries or changes the current print position.

Syntax

```
:setPos( [<nXPos>], [<nYPos>] ) --> aOldPos
```

Arguments

<nXPos>

If specified, this numeric parameter defines the new X coordinate for the print position.

<nYPos>

If specified, this numeric parameter defines the new Y coordinate for the print position.

Description

Method :setPos() returns a two element array {nPosX,nPosY} holding the X and Y coordinates of the print position before the method is called. When parameters are passed, the method sets the new X and/or Y print position accordingly.

:textAtFont()

Prints text at the specified X/Y position and restores previous font settings.

Syntax

```
:textAtFont([<nPosX>]      , ;
            [<nPosY>]      , ;
             <cString>     , ;
            [<cFontName>]  , ;
            [<nPointSize>] , ;
            [<nFontWidth>] , ;
            [<nFontWeight>], ;
            [<lUnderLine>] , ;
            [<lItalic>]    , ;
            [<lNewLine>]   , ;
            [<lUpdatePosX>], ;
            [<nTextColor>] , ;
            [<nTextAlign>]   ) --> lSuccess
```

Arguments

<nPosX> and <nPosY>

These two numeric values are the X and Y coordinates where *<cString>* is output. They default to the current X and Y coordinate of the print position.

<cString>

This is a character string holding the text to print.

<cFontName>

This is a character string holding the name of the font to select as current. It is case sensitive.

<nPointSize>

This is a numeric value indicating the font size in points. If omitted, a default font size is used.

<nFontWidth>

This is a numeric value indicating the font width in pixels. If omitted, a default font width is used.

<nFontWeight>

This is a numeric value indicating the font weight. If omitted, a default font weight is used. #define constants are available in the file WinGdi.ch that can be used for *<nFontWeight>*:

Constants for font weights

Constant	Value
FW_DONTCARE	0
FW_THIN	100
FW_EXTRALIGHT	200
FW_LIGHT	300
FW_NORMAL	400
FW_MEDIUM	500
FW_SEMIBOLD	600
FW_BOLD	700
FW_EXTRABOLD	800
FW_HEAVY	900

<lUnderline>

This is a logical value specifying the underlined mode of the font. The value .T. (true) switches underlined on, and .F. (false) switches it off. When omitted, the current underlined mode of the Win32Prn object is used.

<lItalic>

This is a logical value specifying the *italic* mode of the font. The value .T. (true) switches *italic* on, and .F. (false) switches it off. When omitted, the current *italic* mode of the Win32Prn object is used.

<lNewLine>

This parameter defaults to .F. (false). When set to .T. (true), the print position is moved to the beginning of the next line after *<cString>* is output.

<lUpdatePosX>

This parameter defaults to .T. (true) which changes the X coordinate of the print position to the end of *<cString>* after printing. Subsequent print output appears to the right of *<cString>*. Passing .F. (false), leaves the X coordinate unchanged.

<nTextColor>

This is a numeric RGB color value defining the text color (foreground).

<nTextAlign>

This is an optional numeric value specifying the text alignment at the current print position. The following values can be used:

Values for text alignment

Value	Description
0 *)	Text is printed left aligned to the current print position
1	Text is printed right aligned to the current print position
2	Text is printed centered on the current print position

**) default*

Description

Method :textAtFont() is a combination of methods :texOutAt(), :setFont() and :setColor(). It prints the text string at the given location using the specified font and color. The previous font and color settings are restored after printing. This allows for changing font and color temporarily without having to define new settings for the current font.

The method returns a logical value indicating success of the operation.

:textOut()

Prints text at the current print position.

Syntax

```
:textOut( <cString>]    , ;
         [<lNewLine>]    , ;
         [<lUpdatePosX>], ;
         [<nTexAlign>]     ) --> lSuccess
```

Arguments

<cString>

This is a character string holding the text to print.

<lNewLine>

This parameter defaults to .F. (false). When set to .T. (true), the print position is moved to the beginning of the next line after *<cString>* is output.

<lUpdatePosX>

This parameter defaults to .T. (true) which changes the X coordinate of the print position to the end of *<cString>* after printing. Subsequent print output appears to the right of *<cString>*. Passing .F. (false), leaves the X coordinate unchanged.

<nTextAlign>

This is an optional numeric value specifying the text alignment at the current print position. The following values can be used:

Values for text alignment

Value	Description
0 *)	Text is printed left aligned to the current print position
1	Text is printed right aligned to the current print position
2	Text is printed centered on the current print position

**) default*

Description

Method :textOut() is the most commonly used one for printing text strings. It uses the current :setFont() and :setColor() settings, and updates the X coordinate of the print position by default. The method does

not wrap text to a new line when the output exceeds :rightMargin. Use method :getTextWidth() to determine if *<cString>* still fits into the current line. For example:

```
IF ::posX + ::getTextWidth( <cString> ) > ::rightMargin
   ::newLine()
ENDIF
```

The method returns a logical value indicating success of the operation.

:textOutAt()

Prints text at the specified X/Y position.

Syntax

```
:textOutAt([<nPosX>]      , ;
           [<nPosY>]      , ;
            <cString>     , ;
           [<lNewLine>]   , ;
           [<lUpdatePosX>], ;
           [<nTextAlign>]   ) --> lSuccess
```

Arguments

`<nPosX> and <nPosY>`

These two numeric values are the X and Y coordinates where *<cString>* is output. They default to the current X and Y coordinate of the print position.

`<cString>`

This is a character string holding the text to print.

`<lNewLine>`

This parameter defaults to .F. (false). When set to .T. (true), the print position is moved to the beginning of the next line after *<cString>* is output.

`<lUpdatePosX>`

This parameter defaults to .T. (true) which changes the X coordinate of the print position to the end of *<cString>* after printing. Subsequent print output appears to the right of *<cString>*. Passing .F. (false), leaves the X coordinate unchanged.

`<nTextAlign>`

This is an optional numeric value specifying the text alignment at the current print position. The following values can be used:

Values for text alignment

Value	Description
0 *)	Text is printed left aligned to the current print position
1	Text is printed right aligned to the current print position
2	Text is printed centered on the current print position

**) default*

Description

Method :textOutAt() works exactly like method :textOut(). The only difference are the first two parameters allowing to change the current print position before text is output.

The method returns a logical value indicating success of the operation.

System resources

:create()

Requests system resources for a GDI printer.

Syntax

```
:create() --> lSuccess
```

Description

Method :create() creates the device context for the selected printer and connects the Win32Prn object to the corresponding printer driver. Without this method, print output is not possible.

The method returns a logical value indicating success of the operation.

:destroy()

Releases system resources of a GDI printer.

Syntax

```
:destroy() --> lSuccess
```

Description

Method :destroy() releases the device context for the selected printer and disconnects from the printer driver. The method must be called when print output is complete. A Win32Prn object can call the :create() method again after :destroy().

The method returns a logical value indicating success of the operation.

Info

See also:	SET PRINTER, Win32Bmp()
Category:	Object functions, Printer functions, xHarbour extensions
Header:	wingdi.ch
Source:	rtl\win32prn.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates basic print output using different
// fonts, colors and graphical elements.
// Note: create the EXE using the -gui switch

   #include "WinGdi.ch"

   #define  RGB_BLACK       RGB(   0,  0,  0 )
   #define  RGB_RED         RGB( 255,  0,  0 )
   #define  RGB_GREEN       RGB(   0,255,  0 )
   #define  RGB_BLUE        RGB(   0,  0,255 )
   #define  RGB_CYAN        RGB(   0,255,255 )
   #define  RGB_YELLOW      RGB( 255,255,  0 )
   #define  RGB_MAGENTA     RGB( 255,  0,255 )
   #define  RGB_WHITE       RGB( 255,255,255 )
```

```
PROCEDURE Main
   LOCAL cPrinter := "EPSON Stylus DX5000 Series"
   LOCAL oPrinter
   LOCAL aFonts, cFont, nFont

   // Create printer object and configure print job
   oPrinter           := Win32Prn():new( cPrinter )
   oPrinter:landscape := .F.
   oPrinter:formType  := DMPAPER_A4
   oPrinter:copies    := 1

   // Create device context
   IF .NOT. oPrinter:create()
      Alert( "Cannot create device context" )
      QUIT
   ENDIF

   // Create print job
   IF .NOT. oPrinter:startDoc( "xHarbour test page" )
      Alert( "Cannot create document" )
      QUIT
   ENDIF

   // Text in fixed font
   oPrinter:textOut( "Text in default font" )
   oPrinter:bold( FW_EXTRABOLD )
   oPrinter:textOut( oPrinter:fontName )
   oPrinter:bold( FW_NORMAL )
   oPrinter:newLine()

   aFonts := oPrinter:getFonts()
   nFont  := AScan( aFonts, ;
                    {|a| "ARIAL" $ Upper(a[1]) } )

   cFont  := aFonts[nFont,1]

   // Text in proportional font
   oPrinter:setFont( cFont )
   oPrinter:textOut( "Text in Arial font" )
   oPrinter:bold( FW_EXTRABOLD )
   oPrinter:textOut( oPrinter:fontName )
   oPrinter:bold( FW_NORMAL )
   oPrinter:newLine()

   // Colored text
   oPrinter:setColor( RGB_YELLOW, RGB_BLUE )
   oPrinter:textOut( "Yellow on Blue" )
   oPrinter:newLine()

   // Draw colored line across page
   oPrinter:setPen( PS_DASH, 5, RGB_GREEN )
   oPrinter:line( oPrinter:posX, ;
                  oPrinter:posY, ;
                  oPrinter:rightMargin, ;
                  oPrinter:posY  )

   // Send output to printer
   oPrinter:endDoc()

   // Release GDI device context
   oPrinter:destroy()
RETURN
```

```
FUNCTION RGB( nRed, nGreen, nBlue )
RETURN ( nRed + ( nGreen * 256 ) + ( nBlue * 65536 ) )
```

Command Reference

? | ??

Displays values of expressions to the console window.

Syntax

```
? [<expression,...>]
?? [<expression,...>]
```

Arguments

`<expression,...>`

<expression> is an optional, comma separated list of expressions whose values are output. When no expression is specified, the ? command outputs a new line while ?? outputs nothing.

Description

The ? and ?? commands are text mode console commands that display the result of a comma separated list of expressions to the currently selected output device. This can be the screen, or console window or the printer. The functional equivalent of both commands are the QOut() and QQOut() functions.

The difference between ? and ?? is that the ? command first outputs a carriage-return/line-feed pair so that the output of *<expression,...>* always begins at a new line, while ?? outputs the values of *<expression,...>* at the current cursor or printhead position.

The ? or ?? command first locates the current cursor or printhead position and shifts it one to the right. Row() and Col() are updated with the new cursor position if SET PRINTER is OFF, otherwise they are updated with the new printhead position.

Should the output of ? or ?? reach the right border of the screen (defined by MaxCol()), it will wrap to the next line. Should the output of ? or ?? reach the bottom border of the screen (defined by MaxRow()), the screen will scroll up one line.

It is possible to output the expression(s) to the printer by using the SET PRINTER ON before using the ? or ?? command. It is also possible to output to a text file using the SET ALTERNATE TO command, followed by the SET ALTERNATE ON command. The SET CONSOLE OFF command will prevent display on the screen without interrupting the output to the printer or text file.

When the expression(s) need formatting, use the Transform() function or any user-defined function. For padding, use any of the Pad() functions to center, left align or right align the expression(s).

Info

See also:	@...SAY, PadC() | PadL() | PadR(), QOut() | QQOut(), SET ALTERNATE, SET CONSOLE, SET DEVICE, SET PRINTER, Transform()
Category:	Console commands
Source:	rtl\console.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// This example displays a list of expressions:

   PROCEDURE Main
      LOCAL var1 := 1, var2 := Date(), var3 := .T.

      ? "This line will be displayed on the console."
      ? "This line will be displayed beneath the previous line."
      ?? "No carriage return / linefeed for this expression!"
```

```
   ? var1, var2, var3, "These were variables"
RETURN
```

@...BOX

Displays a box on the screen.

Syntax

```
@ <nTop>, <nLeft>, <nBottom>, <nRight> BOX <cBoxString> ;
                                 [COLOR <cColor>]
```

Arguments

`<nTop> and <nLeft>`

Numeric values indicating the screen coordinates for the upper left corner of the @...BOX output.

`<nBottom> and <nRight>`

Numeric values indicating the screen coordinates for the lower right corner of the @...BOX output.

`<cBoxString>`

The appearance of the box to display is as a character string holding up to nine characters. The first eight characters define the border of the box while the ninth character is used to fill the box. #define constants to be used for *<cBoxString>* are available in the BOX.CH #include file.

Pre-defined box strings for @...BOX

Constant	Description
B_SINGLE	Single-line box
B_DOUBLE	Double-line box
B_SINGLE_DOUBLE	Single-line top, double-line sides
B_DOUBLE_SINGLE	Double-line top, single-line sides

`<cColor>`

An optional SetColor() compliant color string can be specified to draw the box. It defaults to the standard color of SetColor().

Description

The @...BOX command displays a box on the screen as specified with *<cBoxString>*, using the standard color of SetColor() or *<cColor>*, if specified.

The first eight characters of the string *<cBoxString>* define the border of the box in clockwise direction, beginning with the upper left corner. An optional nineth character fills the area inside the box. Alternatively, a single character can be passed which is used to draw the entire box border.

When the box is completely drawn, the cursor is positioned at the coordinates *<nTop>*+1 and *<nLeft>*+1, so that Row() and Col() can be used to start displaying text in the upper left corner of the box area.

Info

See also: @...CLEAR, @...SAY, @...TO, DispBox(), Scroll()
Category: Output commands
Header: box.ch
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates how characters are used to draw a box.
// Alphabetic characters define <cBoxString> instead of characters
// holding graphic signs.

   #include "Box.ch"

   PROCEDURE Main
      CLS
      @ 10, 10, 20, 50 BOX "AbCdEfGhi" COLOR "W+/R"

      Inkey(0)

      @ 10, 10, 20, 50 BOX B_DOUBLE + Space(1)
      @ Row(), Col() SAY "Using #define constant"

      @ MaxRow()-1, 0
   RETURN
```

@...CLEAR

Clears the contents of the screen.

Syntax

```
@ <nTop>, <nLeft> [ CLEAR [TO <nBottom>, <nRight>] ]
```

Arguments

`@ <nTop>, <nLeft>`

Numeric values indicating the screen coordinates for the upper left corner of the @...CLEAR command. If the CLEAR option is not used, only one row of the screen is cleared from the position *<nRow>*, *<nCol>* to the rightmost column.

`CLEAR`

When this option is used, a rectangular area on the screen is cleared, beginning at *<nTop>* and *<nLeft>* down to *<nBottom>* and *<nRight>*.

`TO <nBottom>, <nRight>`

Numeric values indicating the screen coordinates for the lower right corner of the @...CLEAR command. *<nBottom>* defaults to MaxRow() and *<nRight>* defults to MaxCol(). Both parameters require the CLEAR option be used.

Description

The @...CLEAR command is used to clear a rectangular area on the screen in text mode applications. This is accomplished by displaying white space characters in the specified region using the standard color value of SetColor().

After the screen is cleared, the cursor is positioned at the top, left corner of the cleared rectangle. The new cursor position is reflected by the Row() and Col() functions.

Info

See also: @...BOX, CLEAR SCREEN, DispBox(), Scroll(), SetColor(), SetPos()
Category: Output commands
Source: rtl\scroll.c, rtl\setpos.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates various possibilities of
// erasing the screen and parts thereof.

   PROCEDURE Main

      SetColor( "W+/B" )
      CLEAR SCREEN

      SetColor( "W+/R" )

      @ 2, 0                    // deletes the second row

      @ 20, 40 CLEAR            // erase area to bottom/right

      @  5, 30 CLEAR TO 15, 55  // use all four coordinates

   RETURN
```

@...GET

Creates a Get object (entry field) and displays it to the screen

Syntax

```
@ <nRow>, <nCol> ;
     [SAY <xSay> [PICTURE <cSayPict>]  COLOR <cSayColor>] ;
      GET <xVar> [PICTURE <cGetPict>] [COLOR <cGetColor>] ;
    [WHEN <lWhen>] ;
   [VALID <lValid> | RANGE <xMinVal>, <xMaxVal> ] ;
 [MESSAGE <cMessage> ] ;
    [SEND <msg>] ;
 [GUISEND <msg>]
```

Arguments

`@ <nRow>, <nCol>`

The parameters are numeric values specifying the row and column coordinates for the screen output. The range for rows on the screen is 0 to MaxRow(), and for columns it is 0 to MaxCol(). The coordinate 0,0 is the upper left corner of the screen.

`SAY <xSay>`

This is a value of data type C, D, L or N which is displayed in front of the Get object. It is typically a character string indicating the name of the entry field.

`PICTURE <cSayPict>`

If specified, *<cSayPict>* is a character string holding the PICTURE format to be used for displaying *<xSay>*.

`COLOR <cSayColor>`

The parameter *<cSayColor>* is an optional character string defining the color for the output of *<xSay>*. It defaults to the current SetColor() setting. When *<cSayColor>* is specified as a literal color string, it must be enclosed in quotes.

`GET <xVar>`

This is a memory or field variable holding the value to be edited by the user.

`PICTURE <cGetPict>`

If specified, *<cSayPict>* is a character string holding the PICTURE format to be used for display and editing of *<xVar>*.

`COLOR <cGetColor>`

The parameter *<cGetColor>* is an optional character string defining the colors for the display and editing of *<xVar>*. It defaults to the current SetColor() setting. The first color value is used for the display of *<xVar>*, while the second defines the editing color. When *<cGetColor>* is specified as a literal color string, it must be enclosed in quotes.

`WHEN <lWhen>`

This is a logical expression or a code block returning a logical value. If specified, *<lWhen>* must result in .T. (true) to allow editing. If the expression yields .F. (false), the Get object is not editable but skipped.

`VALID <lValid>`

This is a logical expression or a code block returning a logical value. If specified, *<lValid>* must result in .T. (true) to end editing. When the result is .F. (false), the currently edited value of *<xVar>* is invalid and the user cannot leave the Get object unless editing is cancelled.

`RANGE <xMinVal>, <xMaxVal>`

When the value of <*xVar*> is numeric or a Date, the minimum and maximum values allowed for <*xVar*> can be specified with the RANGE clause. Note that RANGE and VALID are mutually exclusive.

`MESSAGE <cMessage>`

This is an optional character string displayed in the message row when a Get object is active. The message row is defined with the MSG AT option of the READ command.

`SEND <msg>`

This is an optional message to be sent to the Get object before it is displayed. Refer to the Get class for possible messages that can be sent to the Get object.

`GUISEND <guimsg>`

This is an optional message to be sent to a control object associated with a Get() object before both are displayed. Such control objects are created with extended @...GET commands:

Extended GET commands and control objects

Command	Control class
@...GET CHECKBOX	HbCheckBox()
@...GET LISTBOX	HbListBox()
@...GET PUSHBUTTON	HbPushButton()
@...GET RADIOGROUP	HbRadioGroup()
@...GET TBROWSE	TBrowse()

Description

The @...GET command creates a new object of the Get class, adds it to the *GetList* array and displays it to the screen. The *GetList* array is a default PUBLIC varible that is reserved for being used by the Get system. It is recommended, however, to declare a new variable named GetList and initialize it with an empty array before the first @...GET command is issued. The value to be edited is held in the variable <*xVar*>.

If the SAY clause is used, <*xSay*> is displayed at the screen position <*nRow*>, <*nCol*>, followed by the value of <*xVar*>. The PICTURE <*cSayPict*> clause optionally specifies a picture format string for the display of <*xSay*>, while the COLOR <*cSayColor*> defines the color. Refer to SetColor() and Transform() for color values and picture formatting of the SAY clause.

If PICTURE <*cGetPict*> is specified, it defines the picture formatting of <*xVar*> for display and editing. See the table below for these picture formatting rules.

If the WHEN clause is specified, it determines wether or not a Get object can receive input focus during the READ command. The <*lWhen*> expression must be a logical expression or a code block returning a logical value. If <*lWhen*> yields .T. (true) the corresponding Get object receives input during READ. When it is .F. (false) this Get object is skipped during READ.

The VALID clause is similar to the WHEN clause, but it is evaluated during the READ command before the user advances to the next get object, i.e. before the current Get object looses input focus. If <*lValid*> yields .T. (true), the next Get object receives input focus. Otherwise, the input focus remains with the current Get object and the user must change the edited value until <*lValid*> results in .T. (true), or editing is cancelled.

Instead of the VALID clause, the RANGE option can be used for numeric values and Dates. The minimum and maximum value valid for <*xVar*> must then be specified with <*xMinVal*> and <*xMaxVal*>.

The PICTURE clause <*cGetPict*> defines the formatting of <*xVar*> for display and editing. It is a character string defining a picture function and/or template. A picture function begins with the @

character. If the picture string contains both, the picture function must appear first, followed by a blank space and the template characters.

PICTURE functions for editing

@A	Only alphabetic characters are allowed.
@B	Numbers are left justified.
@C	CR is displayed after positive numbers.
@D	Date values are displayed in the SET DATE format.
@E	Dates are displayed in British format, numbers in European format.
@K	Clears the Get when the first key pressed is alphanumeric.
@R	Inserts non-template characters for display.
@S<*nWidth*>	Limits the display to <*nWidth*> characters and allows horizontal scrolling when the edited value exceeds <*nWidth*>.
@X	DB is displayed after negative numbers.
@Z	Displays zero values as blanks.
@!	Forces all letters to uppercase.
@(	Displays negative numbers in parentheses with leading spaces.
@)	Displays negative numbers in parentheses without leading spaces.

PICTURE templates for editing

A	Only alphabetic characters are allowed.
N	Only alphanumeric characters are allowed
X	Any character is allowed.
L	Only T or F is allowed For logical values.
Y	Only Y or N is allowed for logical values.
9	Only digits, including signs, are allowed.
#	Only digits, signs and blank spaces are allowed.
!	Alphabetic characters are converted to uppercase.
$	The dollar sign is displayed in place of leading spaces for numbers.
*	Numbers are padded with the asterisk.
.	Position of the decimal point.
,	Position of the comma.

Info

See also: @...SAY, @...CLEAR, Col(), Get(), GetNew(), GetReader(), READ, ReadModal(), Row()
Category: Get system, Input commands
Source: rtl\getsys.prg, rtl\tgetlist.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays two Get entry fields using
// different picture formatting rules

    PROCEDURE Main()
       LOCAL GetList := {}
       LOCAL cVar := Space(50)
       LOCAL nId  := 0

       CLS
       @ 3,1 SAY "Name" GET cVar PICTURE "@!S 30"
       @ 4,1 SAY "Id  " GET nId  PICTURE "999.999"
```

```
   READ

   ? "The name you entered is", cVar
   ? "The id you entered is"  , nId
RETURN
```

@...GET CHECKBOX

Creates a Get object as check box and displays it to the screen.

Syntax

```
@ <nRow>, <nCol> ;
      GET <lLogicVar> ;
 CHECKBOX ;
 [CAPTION <cCaption>] ;
 [MESSAGE <cMessage>] ;
    [WHEN <lWhen>] ;
   [VALID <lValid>] ;
   [COLOR <cColor>] ;
   [FOCUS <bFocus>] ;
   [STATE <bState>] ;
   [STYLE <cStyle>] ;
    [SEND <msg>] ;
 [GUISEND <guimsg>]
```

Arguments

`@ <nRow>, <nCol>`

The parameters are numeric values specifying the row and column coordinates for the screen output. The range for rows on the screen is 0 to MaxRow(), and for columns it is 0 to MaxCol(). The coordinate 0,0 is the upper left corner of the screen.

`GET <lLogicVar> CHECKBOX`

This is a memory or field variable holding a logical value to be edited by the user.

`CAPTION <cCaption>`

This is a character string displayed as caption of the check box. The caption string may include an ampersand character (&) which marks the next character as accelerator key. Pressing the Alt key together with the accelerator key allows the user to jump directly to the check box.

`MESSAGE <cMessage>`

This is an optional character string displayed in the message row when the check box has input focus. The message row is defined with the MSG AT option of the READ command.

`WHEN <lWhen>`

This is a logical expression or a code block returning a logical value. If specified, *<lWhen>* must result in .T. (true) to allow editing. If the expression yields .F. (false), the check box cannot receive input focus.

`VALID <lValid>`

This is a logical expression or a code block returning a logical value. If specified, *<lValid>* must result in .T. (true) to end editing. When the result is .F. (false), the currently edited value of *<lLogicVar>* is invalid and the user cannot leave the check box unless editing is cancelled.

`COLOR <cColor>`

The parameter *<cColor>* is an optional character string defining the colors for the display of the check box and its caption. It defaults to the current SetColor() setting. *<cColor>* must have four color values (see ColorSelect()). They are used as follows:

Color values for check boxes

Color value	Description
1	Color when the check box does not have input focus
2	Color when the check box has input focus
3	Color for the check box's caption
4	Color for the check box's accelerator key

`FOCUS <bFocus>`

This is an optional code block which is evaluated when the check box receives input focus. The code block is evaluated without parameters.

`STATE <bState>`

This is an optional code block which is evaluated when the check box's state is changed by the user. The code block is evaluated without parameters.

`STYLE <cStyle>`

The display style of a check box can be specified with an optional character string of four characters. They are used as follows:

Characters for the display style

Position	Description
1	Left delimiter of check box
2	Character for the Checked state
3	Character for the Unchecked state
4	Right delimiter of check box

`SEND <msg>`

This is an optional message to be sent to the Get object before it is displayed. Refer to the Get class for possible messages that can be sent to the Get object.

`GUISEND <guimsg>`

This is an optional message to be sent to the HbCheckbox() object associated with the Get() object before both are displayed. Refer to the HbCheckBox class for possible messages that can be sent to the check box object.

Description

The @...GET CHECKBOX command creates a Get() object and assigns a HbCheckBox() object to the oGet:control instance variable. The HbCheckBox() object displays a check box in text mode and allows a user to edit a logical state (On or Off) within the READ command. The variable <*lLogicVar*> must be initialized with a logical value before READ is called.

The Get object created with @...GET CHECKBOX is stored in the current Getlist array. The Get object communicates with the associated HbCheckBox object while the READ command is active. This communication ensures that focus and state changes are properly displayed on the screen.

Info

See also: @...GET, Get(), HbCheckBox()
Category: Get system, Input commands
Source: rtl\checkbox.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays two Gets as check boxes using
// different styles.

   PROCEDURE Main
      LOCAL lVar1 := .F.
      LOCAL lVar2 := .T.

      CLS

      @ 3, 3 GET lVar1 CHECKBOX ;
         CAPTION "&Married      " ;
           STYLE "[x ]"   ;
           COLOR "N/BG,W+/BG,N/BG,GR+/BG"

      @ 5, 3 GET lVar2 CHECKBOX ;
         CAPTION "&Self employed" ;
           STYLE "[o ]"   ;
           COLOR "N/BG,W+/BG,N/BG,GR+/BG"

      READ

   RETURN
```

@...GET LISTBOX

Creates a Get object as list box and displays it to the screen.

Syntax

```
@ <nTop>, <nLeft>, <nBottom>, <nRight> ;
      GET <nVar>|<cVar> ;
  LISTBOX <aListItems> ;
 [CAPTION <cCaption>] ;
 [MESSAGE <cMessage>] ;
    [WHEN <lWhen>] ;
   [VALID <lValid>] ;
   [COLOR <cColor>] ;
   [FOCUS <bFocus>] ;
   [STATE <bState>] ;
    [SEND <msg>] ;
 [GUISEND <guimsg>] ;
[DROPDOWN] ;
[SCROLLBAR]
```

Arguments

`@ <nTop>, <nLeft>, <nBottom>, <nRight>`

Numeric values indicating the screen coordinates for the upper left and the lower right corner of the list box output. The range for rows on the screen is 0 to MaxRow(), and for columns it is 0 to MaxCol(). The coordinate 0,0 is the upper left corner of the screen.

`GET <nVar>|<cVar>`

This is the edited variable. It must be initialized either with a numeric value or as a character string. If it is numeric, the variable receives the ordinal position of the selected item held in *<aListItems>* when editing ends. When it is a character string, the selected item of *<aListItems>* is assigned.

`LISTBOX <aListItems>`

A one dimensional array holding character strings defines the list box items a user can select from.

`CAPTION <cCaption>`

This is a character string displayed as caption of the list box. The caption string may include an ampersand character (&) which marks the next character as accelerator key. Pressing the accelerator key allows the user to jump directly to the list box.

`MESSAGE <cMessage>`

This is an optional character string displayed in the message row when the list box has input focus. The message row is defined with the MSG AT option of the READ command.

`WHEN <lWhen>`

This is a logical expression or a code block returning a logical value. If specified, *<lWhen>* must result in .T. (true) to allow editing. If the expression yields .F. (false), the list box cannot receive input focus.

`VALID <lValid>`

This is a logical expression or a code block returning a logical value. If specified, *<lValid>* must result in .T. (true) to end list box selection. When the result is .F. (false), the currently selected item is invalid and the user cannot leave the list box unless the selection is cancelled.

COLOR <cColor>

The parameter <*cColor*> is an optional character string defining the colors for the display of the list box and its caption. It defaults to the current SetColor() setting. <*cColor*> must have up to eight color values (see ColorSelect()). They are used as follows:

Color values for list boxes

Color value	Description
1	Color for unselected list box items when the list box does not have input focus
2	Color for the selected list box item when the list box does not have input focus
3	Color for unselected list box items when the list box has input focus
4	Color for the selected list box item when the list box has input focus
5	Color for the list box's border
6	Color for the list box's caption
7	Color for the list box's accelerator key
8	Color for the list box's drop down button

FOCUS <bFocus>

This is an optional code block which is evaluated when the list box receives input focus. The code block is evaluated without parameters.

STATE <bState>

This is an optional code block which is evaluated when the list box's selection is changed by the user. The code block is evaluated without parameters.

SEND <msg>

This is an optional message to be sent to the Get object before it is displayed. Refer to the Get class for possible messages that can be sent to the Get object.

GUISEND <guimsg>

This is an optional message to be sent to the HbListbox() object associated with the Get() object before both are displayed. Refer to the HbListbox class for possible messages that can be sent to the list box object.

DROPDOWN

If this option is specified, the list box is initially displayed in one row and can be dropped down by pressing the Space key when the list box has input focus, or clicking the dowp down button.

SCROLLBAR

This option displays a scroll bar in the right border of the list box.

Description

The @...GET LISTBOX command creates a Get() object and assigns a HbListbox() object to the oGet:control instance variable. The HbListbox() object displays a list box in text mode and allows a user to select an item from a list of items within the READ command. The variable <*aListItems*> must be initialized with an array before READ is called.

The Get object created with @...GET LISTBOX is stored in the current Getlist array. The Get object communicates with the associated HbListBox object while the READ command is active. This communication ensures that focus and selection changes are properly displayed on the screen.

Info

See also: @...GET, Get(), HbListbox(), HbScrollbar()
Category: Get system, Input commands
Source: rtl\listbox.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example creates a normal and a drop down list box.

   #include "HbLang.ch"
   #include "Inkey.ch"

   PROCEDURE Main
      LOCAL aDays[7]
      LOCAL aHours[24]
      LOCAL cDay := "", nHour := 1
      LOCAL cColor := "N/BG,W+/BG,W+/BG,W+/R,GR+/BG,N/BG,GR+/BG,N/R"

      FOR i:=1 TO 7
         aDays[i] := HB_LangMessage( HB_LANG_ITEM_BASE_DAY + i-1 )
      NEXT

      FOR i:=1 TO 24
         aHours[i] := Padl( i-1, 2, "0" ) + ":00"
      NEXT

      SET EVENTMASK TO INKEY_ALL
      SET COLOR TO ( cColor )
      CLS
      @  3, 15,  7, 25 GET cDay ;
                   LISTBOX aDays ;
                   CAPTION "Select &day " ;
                     COLOR cColor

      @ 10, 15, 20, 25 GET nHour ;
                   LISTBOX aHours ;
                   CAPTION "Select &hour" ;
                     COLOR cColor ;
                  DROPDOWN SCROLLBAR
      READ

      ? "Day :", cDay
      ? "Hour:", aHours[nHour], nHour-1
   RETURN
```

@...GET PUSHBUTTON

Creates a Get object as push button and displays it to the screen.

Syntax

```
 @ <nRow>, <nCol> ;
       GET <lLogicVar> ;
PUSHBUTTON ;
  [CAPTION <cCaption>] ;
  [MESSAGE <cMessage>] ;
     [WHEN <lWhen>] ;
    [VALID <lValid>] ;
    [COLOR <cColor>] ;
    [FOCUS <bFocus>] ;
    [STATE <bState>] ;
    [STYLE <cStyle>] ;
     [SEND <msg>] ;
  [GUISEND <guimsg>]
```

Arguments

`@ <nRow>, <nCol>`

The parameters are numeric values specifying the row and column coordinates for the screen output. The range for rows on the screen is 0 to MaxRow(), and for columns it is 0 to MaxCol(). The coordinate 0,0 is the upper left corner of the screen.

`GET <lLogicVar> PUSHBUTTON`

This is a memory or field variable holding a logical value. It is required as a place holder for the Get, and is set to .F. (false) when the READ command is terminated.

`CAPTION <cCaption>`

This is a character string displayed as caption of the push button. The caption string may include an ampersand character (&) which marks the next character as accelerator key. Pressing the Alt key together with the accelerator key allows the user to jump directly to the push button.

`MESSAGE <cMessage>`

This is an optional character string displayed in the message row when the push button has input focus. The message row is defined with the MSG AT option of the READ command.

`WHEN <lWhen>`

This is a logical expression or a code block returning a logical value. If specified, *<lWhen>* must result in .T. (true) to allow editing. If the expression yields .F. (false), the push button cannot receive input focus.

`VALID <lValid>`

This is a logical expression or a code block returning a logical value. If specified, *<lValid>* must result in .T. (true) to end editing. When the result is .F. (false), the push button retains input focus and READ cannot be ended unless editing is cancelled.

`COLOR <cColor>`

The parameter *<cColor>* is an optional character string defining the colors for the display of the push button. It defaults to the current SetColor() setting. *<cColor>* must have four color values (see ColorSelect()). They are used as follows:

Color values for push buttons

Color value	Description
1	Color when the push button does not have input focus
2	Color when the push button has input focus
3	Color for the push button is pressed
4	Color for the push button's accelerator key

`FOCUS <bFocus>`

This is an optional code block which is evaluated when the push button receives input focus. The code block is evaluated without parameters.

`STATE <bState>`

This is an optional code block which is evaluated when the push button's state is changed by the user. The code block is evaluated without parameters.

`STYLE <cStyle>`

An optional character string of zero, two or eight characters defines the display style. It defaults to "<>" which are the left and right delimiting characters of the push button. When a null string is used for *<cStyle>*, no delimiting characters are used.

A character string of eight characters defines the border drawn around the pushbutton. They are used in the same way as with the DispBox() function. The #define constants available in Box.ch can be used:

Constants for borders

Constant	Description
B_SINGLE	Single-line border
B_DOUBLE	Double-line border
B_SINGLE_DOUBLE	Single-line top, double-line sides
B_DOUBLE_SINGLE	Double-line top, single-line sides

If *<cStyle>* has eight characters, the push button has a height of three screen rows, otherwise it is displayed in one screen row.

`SEND <msg>`

This is an optional message to be sent to the Get object before it is displayed. Refer to the Get class for possible messages that can be sent to the Get object.

`GUISEND <guimsg>`

This is an optional message to be sent to the HbPushButton() object associated with the Get() object before both are displayed. Refer to the HbPushButton class for possible messages that can be sent to the push button object.

Description

The @...GET PUSHBUTTON command creates a Get() object and assigns a HbPushButton() object to the oGet:control instance variable. The HbPushButton() object displays a push button in text mode and allows a user to press it while the READ command is active. The variable *<lLogicVar>* contains always .F. (false) when READ is terminated.

The Get object created with @...GET PUSHBUTTON is stored in the current Getlist array. The Get object communicates with the associated HbPushButton object while the READ command is active. This communication ensures that focus and state changes are properly displayed on the screen.

Info

See also: @...GET, Get(), HbPushButton()
Category: Get system, Input commands
Source: rtl\pushbtn.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates how push buttons can be integrated
// in the Get list and are activated during READ

   PROCEDURE Main
      LOCAL cLName, cFName
      LOCAL lBtn1    := .F.
      LOCAL lBtn2    := .F.
      LOCAL lBtn3    := .F.
      LOCAL nBtn     := 0
      LOCAL cGetClr  := "W+/B,W+/R,N/BG,GR+/BG"
      LOCAL cBtnClr  := "N/G,W+/G,GR+/N,GR+/G"

      SET EVENTMASK TO INKEY_ALL
      SET COLOR TO N/BG
      CLS

      USE Customer ALIAS Cust
      cFName := Cust->FirstName
      cLName := Cust->LastName

      @  8, 20 GET cFName ;
           CAPTION "&First name:" ;
             COLOR cGetClr

      @ 10, 20 GET cLName ;
           CAPTION " &Last name:" ;
             COLOR cGetClr

      @ 12, 20 GET lBtn1 PUSHBUTTON ;
           CAPTION " &Save " ;
             COLOR cBtnClr ;
             STATE {|| nBtn := IsPressed( 1 ) }

      @ 12, 30 GET lBtn2 PUSHBUTTON ;
           CAPTION " &Undo " ;
             COLOR cBtnClr ;
             STATE {|| nBtn := IsPressed( 2 ) }

      @ 12, 40 GET lBtn3 PUSHBUTTON ;
           CAPTION " E&xit " ;
             COLOR cBtnClr ;
             STATE {|| nBtn := IsPressed( 3 ) }

      DO WHILE .T.
         READ SAVE

         DO CASE
         CASE nBtn == 1
            IF RLock()
               REPLACE Cust->FirstName WITH cFName
               REPLACE Cust->LastName  WITH cLName
```

```
            DbUnlock()
            EXIT
         ELSE
            Alert( "Unable to save;Retry again later" )
         ENDIF

      CASE nBtn == 2
         cFName := Cust->FirstName
         cLName := Cust->LastName
         nBtn   := 0
         AEval( GetList, {|oGet| oGet:display() } )

      CASE nBtn == 3
         EXIT

      ENDCASE
   ENDDO

   ASize( GetList, 0 )
   CLOSE ALL
RETURN
```

@...GET RADIOGROUP

Creates a Get object as radio button group and displays it to the screen.

Syntax

```
 @ <nTop>, <nLeft>, <nBottom>, <nRight> ;
       GET <nVar>|<cVar> ;
RADIOGROUP <aRadioButtons> ;
  [CAPTION <cCaption>] ;
  [MESSAGE <cMessage>] ;
     [WHEN <lWhen>] ;
    [VALID <lValid>] ;
    [COLOR <cColor>] ;
    [FOCUS <bFocus>] ;
     [SEND <msg>] ;
  [GUISEND <guimsg>]
```

Arguments

`@ <nTop>, <nLeft>, <nBottom>, <nRight>`

Numeric values indicating the screen coordinates for the upper left and the lower right corner of the radio button group. The range for rows on the screen is 0 to MaxRow(), and for columns it is 0 to MaxCol(). The coordinate 0,0 is the upper left corner of the screen.

`GET <nVar>|<cVar>`

This is the edited variable. It must be initialized either with a numeric value or as a character string. If it is numeric, the variable receives the ordinal position of the selected radio button held in *<aRadioButtons>* when the selection ends. When it is a character string, the string data of the selected radio button of *<aRadioButtons>* is assigned.

`RADIOGROUP <aRadioButtons>`

This is a one dimensional array holding HbRadioButton() objects. They display individual that are displayed within the radio group.

`CAPTION <cCaption>`

This is a character string displayed as caption of the radio button group. The caption string may include an ampersand character (&) which marks the next character as accelerator key. Pressing the accelerator key allows the user to jump directly to the radio button group.

`MESSAGE <cMessage>`

This is an optional character string displayed in the message row when the radio button group has input focus. The message row is defined with the MSG AT option of the READ command.

`WHEN <lWhen>`

This is a logical expression or a code block returning a logical value. If specified, *<lWhen>* must result in .T. (true) to allow selection. If the expression yields .F. (false), the radio button group cannot receive input focus.

`VALID <lValid>`

This is a logical expression or a code block returning a logical value. If specified, *<lValid>* must result in .T. (true) to end selection in the radio button group. When the result is .F. (false), the currently selected option is invalid and the user cannot leave the radio button group unless the selection is cancelled.

COLOR <cColor>

The parameter <*cColor*> is an optional character string defining the colors for the display of the list box and its caption. It defaults to the current SetColor() setting. <*cColor*> must have up to eight color values (see ColorSelect()). They are used as follows:

Color values for radio button groups

Color value	Description
1	Color for the border around the radio button group
2	Color for the caption
3	Color for the accelerator key

FOCUS <bFocus>

This is an optional code block which is evaluated when the radio button group receives input focus. The code block is evaluated without parameters.

SEND <msg>

This is an optional message to be sent to the Get object before it is displayed. Refer to the Get class for possible messages that can be sent to the Get object.

GUISEND <guimsg>

This is an optional message to be sent to the HbRadioGroup() object associated with the Get() object before both are displayed. Refer to the HbRadioGroup class for possible messages that can be sent to the radio button group object.

Description

The @...GET RADIOGROUP command creates a Get() object and assigns a HbRadioGroup() object to the oGet:control instance variable. The HbRadioGroup() object displays all HbRadioButton() objects stored in the <*aRadioButtons*> array in text mode and allows a user to select one of the redio buttons while the READ command is active. The position or the data of the selected radio button is assigned to the edited variable when READ is terminated.

The Get object created with @...GET RADIOGROUP is stored in the current Getlist array. The Get object communicates with the associated HbRadioGroup object while the READ command is active. This communication ensures that focus and state changes are properly displayed on the screen.

Info

See also:	@...GET, Get(), HbRadioButton(), HbRadioGroup()
Category:	Get system, Input commands
Source:	rtl\radiogrp.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates how radio button groups can be integrated
// in the Get list and are filled with radio buttons.

   #include "Inkey.ch"

   PROCEDURE Main
      LOCAL aRadio1[3], aRadio2[4]
      LOCAL nSpeed, cPages
      LOCAL cColor1 := "W/B,W+/B,R/W+"
      LOCAL cColor2 := "N/BG,N/BG,GR+/BG,GR+/BG,N/BG,W+/BG,GR+/BG"

      SET EVENTMASK TO INKEY_ALL
```

```
   SET COLOR TO N/BG
   CLS
   nSpeed := 1            //default to the first item.

   aRadio1[ 1 ] := HbRadioButton():new( 4, 22, "&slow" )
   aRadio1[ 2 ] := HbRadioButton():new( 5, 22, "&medium" )
   aRadio1[ 3 ] := HbRadioButton():new( 6, 22, "&fast" )
   AEval( aRadio1, {|o| o:colorSpec := cColor2 } )

   cPages := "all"
   aRadio2[ 1 ] := HbRadioButton():new( 10, 22, "&first page", "this" )
   aRadio2[ 2 ] := HbRadioButton():new( 11, 22, "&this page" , "first")
   aRadio2[ 3 ] := HbRadioButton():new( 12, 22, "&last page" , "last" )
   aRadio2[ 4 ] := HbRadioButton():new( 13, 22, "&all pages" , "all"  )
   AEval( aRadio2, {|o| o:colorSpec := cColor2 } )

   @  3, 20,  7, 40 GET nSpeed ;
          RADIOGROUP aRadio1 ;
             CAPTION "&Speed" ;
               COLOR cColor1

   @  9, 20, 14, 40 GET cPages ;
          RADIOGROUP aRadio2 ;
             CAPTION "&Pages" ;
               COLOR cColor1
   READ

   ? nSpeed, cPages
RETURN
```

@...GET TBROWSE

Creates a Get object as browser and displays it to the screen

Syntax

```
@ <nTop>, <nLeft>, <nBottom>, <nRight> ;
      GET <xVar> ;
  TBROWSE <oTBrowse> ;
 [MESSAGE <cMessage>] ;
    [WHEN <lWhen>] ;
   [VALID <lValid>] ;
    [SEND <msg>] ;
 [GUISEND <guimsg>] ;
```

Arguments

`@ <nTop>, <nLeft>, <nBottom>, <nRight>`

Numeric values indicating the screen coordinates for the upper left and the lower right corner of the browse output. The range for rows on the screen is 0 to MaxRow(), and for columns it is 0 to MaxCol(). The coordinate 0,0 is the upper left corner of the screen.

`GET <xVar>`

This is a memory or field variable holding a logical value. It is required as a place holder for the Get, and is not changed when the READ command is terminated.

`TBROWSE <oTBrowse>`

This must be a completely configured TBrowse() object to be associated with the new Get.

`MESSAGE <cMessage>`

This is an optional character string displayed in the message row when the TBrowse object has input focus. The message row is defined with the MSG AT option of the READ command.

`WHEN <lWhen>`

This is a logical expression or a code block returning a logical value. If specified, *<lWhen>* must result in .T. (true) to allow editing. If the expression yields .F. (false), the TBrowse object cannot receive input focus.

`VALID <lValid>`

This is a logical expression or a code block returning a logical value. If specified, *<lValid>* must result in .T. (true) to end browsing. When the result is .F. (false), the currently selected item is invalid and the user cannot leave the TBrowse unless browsing is cancelled.

`SEND <msg>`

This is an optional message to be sent to the Get object before it is displayed. Refer to the Get class for possible messages that can be sent to the Get object.

`GUISEND <guimsg>`

This is an optional message to be sent to the HbListbox() object associated with the Get() object before both are displayed. Refer to the TBrowse class for possible messages that can be sent to the TBrowse object.

Description

The @...GET TBROWSE command creates a Get() object and assigns a TBrowse() object to the oGet:control instance variable. The TBrowse object must be configured and displays data of the underlying data source (database or array) in a text mode browse view which and allows a user navigate the data source within the READ command.

The Get object created with @...GET TBROWSE is stored in the current Getlist array. The Get object communicates with the associated TBrowse object while the READ command is active. This communication ensures that focus and selection changes are properly displayed on the screen.

Info

See also: @...GET, Get(), TBrowse()
Category: Get system, Input commands
Source: rtl\listbox.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example combines three Get entry fields with a browse
// and three push buttons. When the database is navigated
// and the user selects a record with the Enter key, READ
// is terminated and restarted with new data read from the
// selected record.

   #include "GetExit.ch"
   #include "Inkey.ch"

   PROCEDURE Main
      LOCAL cLName, cFName, cPhone
      LOCAL lBtn1   := .F.
      LOCAL lBtn2   := .F.
      LOCAL lBtn3   := .F.
      LOCAL nBtn    := 0
      LOCAL dummy   := NIL
      LOCAL cGetClr := "W+/B,W+/R,N/BG,GR+/BG"
      LOCAL cBtnClr := "N/G,W+/G,GR+/N,GR+/G"
      LOCAL oTBrowse, aColumns[3], bSkip, bSaved

      SET EVENTMASK TO INKEY_ALL
      SET COLOR TO N/BG
      CLS

      USE Customer ALIAS Cust
      oTBrowse           := TBrowseDb( 8, 50, 14, 72 )
      oTBrowse:colorSPec := cGetClr
      oTBrowse:headSep   := "-"
      oTBrowse:cargo     := Recno()

      aColumns[1] := TBColumnNew( "Last" , {|| CUst->LastName  } )
      aColumns[2] := TBColumnNew( "First", {|| CUst->FirstName } )
      aColumns[3] := TBColumnNew( "Phone", {|| CUst->Phone     } )

      AEval( aColumns, {|o| oTBrowse:addColumn(o) } )

      cFName := Cust->FirstName
      cLName := Cust->LastName
      cPhone := Cust->Phone

      @  8, 20 GET cFName ;
           CAPTION "&First name:" ;
             COLOR cGetClr

      @ 10, 20 GET cLName ;
           CAPTION " &Last name:" ;
             COLOR cGetClr
```

```
   @ 12, 20 GET cPhone ;
        CAPTION "     &Phone:" ;
          COLOR cGetClr

   @  8, 50, 14, 72 GET dummy ;
                TBROWSE oTBrowse ;
                GUISEND forceStable() ;
                  VALID {|| nBtn := RecChanged( oTBrowse ), .T. }

   @ 16, 20 GET lBtn1 PUSHBUTTON ;
        CAPTION " &Save " ;
          COLOR cBtnClr ;
          STATE {|| nBtn := IsPressed( 1 ) }

   @ 16, 30 GET lBtn2 PUSHBUTTON ;
        CAPTION " &Undo " ;
          COLOR cBtnClr ;
          STATE {|| nBtn := IsPressed( 2 ) }

   @ 16, 40 GET lBtn3 PUSHBUTTON ;
        CAPTION " E&xit " ;
          COLOR cBtnClr ;
          STATE {|| nBtn := IsPressed( 3 ) }

   DO WHILE nBtn == 0
      READ SAVE

      DO CASE
      CASE nBtn == 1
         IF RLock()
            REPLACE Cust->FirstName WITH cFName
            REPLACE Cust->LastName  WITH cLName
            REPLACE Cust->Phone     WITH cPhone
            DbCommit()
            DbUnlock()
         ENDIF

      CASE nBtn == 2
         cFName := Cust->FirstName
         cLName := Cust->LastName
         cPhone := Cust->Phone
         nBtn   := 0
         AEval( GetList, {|oGet| oGet:display() } )
      CASE nBtn == 3
        nBtn := 0
      ENDCASE
   ENDDO

   ASize( GetList, 0 )
   CLOSE ALL
RETURN


FUNCTION RecChanged()
   LOCAL oGet     := GetActive()
   LOCAL oTBrowse := oGet:control
   LOCAL nButton  := 0

   IF oGet:exitState == GE_ENTER
      IF oTBrowse:cargo <> Recno()
         ReadKill( .T. )
```

```
         nButton := 2
      ENDIF
   ELSE
      DbGoto( oTBrowse:cargo )
      oTBrowse:refreshAll()
      oTBrowse:forceStable()
   ENDIF

RETURN nButton


FUNCTION IsPressed( nButton )
   IF GetActive():control:buffer
      ReadKill( .T. )
   ELSE
      nButton := 0
   ENDIF
RETURN nButton
```

@...PROMPT

Displays a menu item for a text mode menu.

Syntax

```
@ <nRow>, <nCol> PROMPT <cMenuItem> ;
               [MESSAGE <bcMessage>]
```

Arguments

`@ <nRow>, <nCol>`

Numeric values indicating the screen coordinates for the display of <*cMenuItem*>.

`PROMPT <cMenuItem>`

A character string holding the menu item of the text mode menu.

`MESSAGE <bcMessage>`

Optionally, a character string can be associated with <*cPrompt*> which is displayed as a message when SET MESSAGE is set to a screen row larger than zero. Instead of a character string, a code block can be specified. In must return a character string.

Description

The @...PROMPT command is used to define a single menu item of a simple text mode menu. A text mode menu is usually built by specifying the @...PROMPT command multiple times before the MENU TO command is issued to activate the menu.

The screen cursor is located one column to the right of <*cMenuItem*> after @...PROMPT has displayed the menu item.

Note: For an example of @...PROPMT usage see the MENU TO command.

Info

See also:	AChoice(), MENU TO, SET MESSAGE, SET WRAP, SetColor()
Category:	Input commands
Source:	rtl\menuto.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

@...SAY

Displays data at a particular row and column position.

Syntax

```
@ <nRow>, <nCol> ;
      SAY <expression> ;
 [PICTURE <cPicture>] ;
   [COLOR <cColor>]
```

Arguments

`@ <nRow>, <nCol>`

The parameters are numeric values specifying the row and column coordinates for the output. The range for rows on the screen is 0 to MaxRow(), and for columns it is 0 to MaxCol(). The coordinate 0,0 is the upper left corner of the screen.

When SET DEVICE TO PRINTER is active, the largest coordinate for both, row and column, is 32766.

`SAY <expression>`

This is an expression whose value is output to the current device.

`PICTURE <cPicture>`

The PICTURE clause defines a character string which specifies how to format the value of *<expression>* for output. See the explanation of formatting rules below.

`COLOR <cColor>`

The parameter *<cColor>* is an optional character string defining the color for output. It defaults to the current SetColor() setting. When *<cColor>* is specified as a literal color string, it must be enclosed in quotes.

Description

@...SAY is a text-mode output command that outputs the value of an expression to the current device, which is either the screen or a printer. Only values of simple data types like Character, Date, Numeric and Logic are output. Complex values, like Array or Object cannot be used with @...SAY.

The output device for @...SAY is selected with the SET DEVICE command. The SET CONSOLE setting has no effect. When the SCREEN is the output device, @...SAY outputs the value of *<expression>* at the specified row and column position and moves the screen cursor to the end of the displayed value. Row() and Col() reflect the new position of the cursor.

With printed output, the printhead position is changed and PRow() and PCol() are updated accordingly. When the row coordinate for printed output is smaller than the current value of PRow(), an EJECT command is automatically sent to the printer. If the column coordinate including SET MARGIN is smaller than the current value of PCol(), a "new line" command is sent to the printer along with the number of spaces to position the printhead to the new column. To redirect the @...SAY output to a file, SET DEVICE TO PRINTER and SET PRINTER TO *<fileName>*.

The output color can be specified with a color string compliant with the SetColor() function. The default color used is the standard color, which is the first color value of a color string.

Output can be formatted using the PICTURE clause. It specifies a character string that may consist of two parts: a picture function and a picture mask. A picture function begins with the @ sign, followed by one or more letters indicating a picture function to apply. A picture mask is a series of characters. When both, picture function and mask, are used in the picture string, they must be separated with a single space character and the picture function must be placed first in the picture string.

Picture function

A picture function specifies formatting rules for the entire output string. It must begin with the @ sign followed by one or more letters listed in the table below:

Picture function characters

Function	Formatting rule
B	Displays numbers left-justified
C	Displays CR after positive numbers
D	Displays dates in SET DATE format
E	Displays dates and numbers in British format
L	Displays numbers with leading zeros instead of blank spaces
R	Nontemplate characters are inserted
X	Displays DB after negative numbers
Z	Displays zeros as blanks
(	Encloses negative numbers in parentheses
!	Converts alphabetic characters to uppercase

Picture mask

The picture mask must be separated by a single space from the picture function. When no picture function is used, the picture string is identical with the picture mask. The mask defines formatting rules for individual characters in the output string. Characters from the following table can be used. The position of a character of a picture mask specifies formatting for the character of the output string at the same position. An exception is the @R function which causes non-mask characters being inserted into the output string.

Picture mask characters

Character	Formatting rule
A,N,X,9,#	Displays digits for any data type
L	Displays logicals as "T" or "F"
Y	Displays logicals as "Y" or "N"
!	Converts alphabetic characters to uppercase
$	Displays a dollar sign in place of a leading space in a number
*	Displays an asterisk in place of a leading space in a number
.	Specifies a decimal point position
,	Specifies a comma position

Info

See also: ? \| ??, @...CLEAR, @...GET, Col(), GetNew(), PCol(), PRow(), QOut() \| QQOut(), Row(), SetColor(), Transform()
Category: Output commands
Source: rtl\console.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates different formatting with @...SAY and
// the PICTURE clause

   PROCEDURE Main
      LOCAL nGain  := 8596.58
      LOCAL nLoss  := -256.50
      LOCAL cPhone := "5558978532"
      LOCAL cName  := "Jon Doe"
```

```
    @ 10, 1 SAY nGain PICTURE "@E 9,999.99"
    // result: 8.596,58

    @ 11, 1 SAY nLoss PICTURE "@)"
    // Result: (256.50)

    @ 12, 1 SAY cPhone PICTURE "@R (999)999-9999"
    // Result: (555)897-8532

    @ 13, 1 SAY cName PICTURE "@!"
    // Result: JOHN DOE
RETURN
```

@...TO

Displays a single or double lined frame on the screen.

Syntax

```
@ <nTop>, <nLeft> TO <nBottom>, <nRight> ;
              [COLOR <cColor>] [DOUBLE]
```

Arguments

`@ <nTop>, <nLeft>`

Numeric values indicating the screen coordinates for the upper left corner of the frame to display.

`TO <nBottom>, <nRight>`

Numeric values indicating the screen coordinates for the lower right corner of the frame to display.

`<cColor>`

An optional SetColor() compliant color string can be specified to draw the frame. It defaults to the standard color of SetColor().

`DOUBLE`

This option displays the frame with a double line. If omitted, a single line is used.

Description

The @...TO command displays a frame on the screen using the standard color of SetColor() or *<cColor>*, if specified.

When the frame is completely drawn, the cursor is positioned at the coordinates *<nTop>*+1 and *<nLeft>*+1, so that Row() and Col() can be used to start displaying text in the upper left corner of the frame area.

Info

See also:	@...BOX, @...CLEAR, DispBox()
Category:	Output commands
Source:	rtl\box.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays two frames using different colors and
// borders.

   PROCEDURE Main
      CLS
      @ 10, 5 TO 20, 35 COLOR "W+/B"
      @ Row(), Col() SAY "Single"

      @ 10, 45 TO 20, 75 COLOR "N/BG" DOUBLE
      @ Row(), Col() SAY "Double"
   RETURN
```

ACCEPT

Basic user input routine.

Syntax

```
ACCEPT [<xMessage>] TO <cVarName>
```

Arguments

`<xMessage>`

This is an optional expression of data type C, D, L or N. The value of *<xMessage>* is displayed before the user can input data.

`<cVarName>`

This is a character string holding the name of the variable to assign the user input to. It must be of PRIVATE or PUBLIC scope. If *<cVarName>* does not exist, a PRIVATE variable is created.

Description

The ACCEPT command exists for compatibility reasons only. It is a very basic text-mode command that accepts user input to be assigned as a character string to a single memory variable.

During user input only the Return key is recognized as termination key. The Escape key does not terminate the ACCEPT command.

Info

See also:	@...GET, @...SAY, Inkey(), INPUT, KEYBOARD
Category:	Input commands
Source:	rtl\accept.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows a simple command line utility used to
// count the lines in an ASCII file. Note the PARAMETERS
// statement: it declares a parameter of PRIVATE scope for Main().

   PROCEDURE Main
      PARAMETERS cFileName

      IF Empty( cFileName )
         ACCEPT "Enter a file name: " TO cFileName
      ENDIF

      IF Empty( cFileName )
         ? "User aborted"
      ELSEIF .NOT. File( cFileName )
         ? "File not found:", cFileName
      ELSE
         ? "Line count:", LTrim( Str( FLineCount( cFileName ) ) )
      ENDIF
   RETURN
```

APPEND BLANK

Creates a new record in the current work area.

Syntax

```
APPEND BLANK
```

Description

The APPEND BLANK command creates a new record for the database file open in the current work area and moves the record pointer to the new record. All fields of the record are filled with empty data of the corresponding field data types. That is: Character fields are filled with blank spaces, Date fields contain CtoD(""), logical fields contain .F. (false), Memo fields contain nothing and Numeric fields contain 0.

Shared file access: When the database is opened in SHARED mode, the APPEND BLANK command attempts to lock the ghost record before appending a new record to the file (the ghost record is the one located at LastRec()+1). If this record lock fails, function NetErr() is set to .T. (true) and no record is appended. When the record lock succeeds, it remains active until a new APPEND BLANK command is issued or the lock is explicitely released with the UNLOCK command. A file lock obtained with the FLock() function is not released by the APPEND BLANK command.

Note: APPEND BLANK works only in the current work area. Its functional equivalent is the DbAppend() function which can create a new record in other work areas when used in an aliased expression.

Info

See also:	APPEND FROM, DbAppend(), FLock(), Neterr(), RLock(), USE
Category:	Database commands
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The examples demonstrates a basic technique for creating a new record
// in a shared database:

   PROCEDURE Main
      LOCAL cFirstName := Space(40)
      LOCAL cLastName  := Space(50)

      USE Address ALIAS Addr SHARED NEW
      IF .NOT. NetErr()
         SET INDEX TO AddressA, AddressB
      ENDIF

      CLS
      @ 10,5 SAY "Enter first name:" GET cFirstName
      @ 11,5 SAY "Enter last name :" GET cLastName
      READ

      IF .NOT. Empty( cFirstName + cLastName )
         APPEND BLANK
         IF NetErr()
            ? "Unable to create new record"
         ELSE
            REPLACE addr->FirstName WITH cFirstName  , ;
                    addr->LastName  WITH cLastName
```

```
      ENDIF
    ENDIF

    USE
  RETURN
```

APPEND FROM

Imports records from a database file or an ASCII text file.

Syntax

```
APPEND FROM <sourceFile> ;
    [FIELDS <fieldNames,...> ;]
    [<Scope> ; ]
     [WHILE <lWhileCondition> ;]
       [FOR <lForCondition> ;]
       [VIA <rddName> ;]
       [SDF | DELIMITED [WITH BLANK | TAB | PIPE | <xDelimiter> ;]
  [CODEPAGE <cCodePage> ;]
[CONNECTION <nConnection>] ]
```

Arguments

`FROM <sourceFile>`

This is the name of the source file as a literal character string or a character expression enclosed in parentheses. When the file name is specified without a file extension, .dbf is assumed as default extension, unless the SDF or DELIMITED option is specified. In this case, the default file extension is .txt.

`FIELDS <fieldNames,...>`

The names of the fields to import from the external file can be specified as a comma separated list of literal field names or character expressions enclosed in parentheses. When this option is omitted, all fields of the source file are imported.

`<Scope>`

This option defines the number of records to import. It defaults to ALL. The NEXT *<nCount>* scope imports the first *<nCount>* records, while the RECORD *<nRecno>* scope imports only one record having the record number *<nRecno>*.

`WHILE <lWhileCondition>`

This is a logical expression indicating to continue importing records while the condition is true. The APPEND FROM operation stops as soon as *<lWhileCondition>* yields .F. (false).

`FOR <lForCondition>`

This is a logical expression which is evaluated for all records in the source file. Those records where *<lForCondition>* yields .T. (true) are imported.

`VIA <rddName>`

The VIA option specifies the replaceable database driver (RDD) to use for opening the source file. *<rddName>* is a character expression. If it is written as a literal name, it must be enclosed in quotes.

When no RDD is specified, the RDD active in the current work area is used to open the import file. If the VIA clause is used, the specified RDD must be linked to the application. Use the REQUEST command to force an RRD be linked.

`SDF`

The SDF option specifies the source file as an ASCII file in System Data Format. See file format description below.

`DELIMITED`

The DELIMETED option specifies the source file as a delimited ASCII file where field values are separated with a comma and Character values are enclosed with a delimiting character. The default delimiter for Character values is a double quotation mark.

`DELIMITED WITH BLANK | TAB | PIPE`

When the delimiter is specifiwed as BLANK, field values in the ASCII text file are separated by one space and character fields are not enclosed in delimiters. Alternatively, the delimiting character between field values can be specified as TAB (Chr(9)) or PIPE (Chr(124)).

`DELIMITED WITH <xDelimiter>`

The WITH option specifies the delimiting character to enclose values of Character fields in. *<xDelimiter>* can be specified as a literal character or a character expression enclosed in parentheses.

<xDelimiter> can also be specified as an array with two elements: { *<cCharacterDelimiter>*, *<cFieldDelimiter>* }. If this option is used, the array must be enclosed in parentheses. It defines the delimiting characters for field values of type "C" and the delimiters between field values.

Important: If the DELIMITED WITH option is used in the APPEND FROM command, it must be placed as the last option in the command.

`CODEPAGE <cCodePage>`

This is a character string specifying the code page to use for character strings stored in the new database. It defaults to the return value of HB_SetCodePage().

`CONNECTION <nConnection>`

This option specifies a numeric server connection handle. It is returned by a server connection function which establishes a connection to a database server, such as SR_AddConnection() of the xHarbour Builder SQLRDD. When CONNECTION is used, the APPEND FROM command appends data from a database on the server.

Description

APPEND FROM is a database command used to import data from a second file into the current work area. Imported records are added to the end of the table open in the current work area. When the external file is a database, only fields with matching names are imported. Fields with matching names must have the same data type, otherwise a runtime error occurs.

The file in the current work area can be opened in SHARED mode. APPEND FROM automatically maintains required record locks while data is being imported.

The import file is attempted to be opened in SHARED and READONLY mode. If opening the import file fails, APPEND FROM raises a runtime error.

When the external file is an ASCII text file, all date values must be stored in the StoD() compliant format yyyymmdd. Other specifications for supported ASCII formats ar listed in the folowing tables:

SDF Text File

File Element	Format
Character fields	Padded with trailing blanks
Date fields	yyyymmdd
Logical fields	T or F
Memo fields	Ignored
Numeric fields	Padded with leading blanks or zeros
Field separator	None
Record separator	Carriage return/linefeed
End of file marker	0x1A or CHR(26)

DELIMITED Text File

File Element	Format
Character fields	May be delimited, with trailing blanks truncated
Date fields	yyyymmdd
Logical fields	T or F
Memo fields	Ignored
Numeric fields	Leading zeros may be truncated
Field separator	Comma
Record separator	Carriage return/linefeed
End of file marker	0x1A or CHR(26)

DELIMITED WITH BLANK Text File

File Element	Format
Character fields	Not delimited, trailing blanks may be truncated
Date fields	yyyymmdd
Logical fields	T or F
Memo fields	Ignored
Numeric fields	Leading zeros may be truncated
Field separator	Single blank space
Record separator	Carriage return/linefeed
End of file marker	0x1A or CHR(26)

Field length: When the length of Character fields in the current work area is shorter than the length of a corresponding field in the import file, character data is truncated up to the length of the field in the current work area. Truncated data is lost. If the field is longer, remaining characters are filled with blank spaces.

When the length of a numeric field in the current work area does not suffice to hold all digits of an imported numeric field, a runtime error is created.

Deleted records: Records in an import database file that are marked for deletion are ignored when SET DELETED is set to ON. These records are not imported. With SET DELETED OFF, all records matching the import scope are imported and the deleted flag is set accordingly.

Info

See also: COPY TO, DbAppend(), UPDATE
Category: Database commands
Source: rdd\dbcmd.c, rtl\dbdelim.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example creates a subset of records from a database
// within an intermediate file.

   PROCEDURE Main

      USE Address NEW
      COPY STRUCTURE TO Tmp          // create an intermediate file
      USE Tmp

      APPEND FROM Address FOR "LONDON" $ Upper(CITY)

      Browse()
      USE
```

```
   ERASE Tmp.dbf                  // delete intermediate file
RETURN
```

AVERAGE

Calculates the average of numeric expressions in the current work area.

Syntax

```
AVERAGE <expressions,...> ;
     TO <resultVarNames,...> ;
       [<Scope>] ;
 [WHILE <lWhileCondition>] ;
   [FOR <lForCondition>]
```

Arguments

`AVERAGE <expressions,...>`

This is a comma separated list of numeric expressions that are evaluated for the records of the current work area.

`TO <resultVarNames,...>`

A comma separated list of variable names that are assigned the results of the calculation. The number of *<resultVarNames,...>* must match exactly the number of *<expressions,...>*.

`<Scope>`

This option defines the number of records to process. It defaults to ALL. The NEXT *<nCount>* scope processes the next *<nCount>* records.

`WHILE <lWhileCondition>`

This is a logical expression indicating to continue calculation while the condition is true. The AVERAGE operation stops as soon as *<lWhileCondition>* yields .F. (false).

`FOR <lForCondition>`

This is a logical expression which is evaluated for all records in the current work area. Those records where *<lForCondition>* yields .T. (true) are included in the calculation.

Description

The AVERAGE command is used to calculate average values for one or more numeric expressions. The expressions are evaluated in the current work area. The number of records to include in the calculation can be restricted using a FOR and/or WHILE condition or by explicitely defining a scope.

Info

See also:	COUNT, DbEval(), SUM, TOTAL
Category:	Database commands
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// This example calculates the average payments for the first half
// of this year:

   PROCEDURE Main
      LOCAL nAvg

      USE Sales NEW

      AVERAGE Payment TO nAvg FOR Month(SALEDATE) < 7 .AND. ;
```

```
                                        Year(SALEDATE) == Year(Date())
      USE

      ? nAvg
   RETURN
```

CANCEL

Terminates program execution unconditionally.

Syntax

```
CANCEL
```

Description

CANCEL is a compatibility command that is replaced by the QUIT command. Both perform the same action of terminating an xHarbour application.

Info

See also: QUIT
Category: Environment commands
Source: Library is rtl.lib

Example

```
// Refer to the QUIT command
```

CLEAR ALL

Closes files in all work areas and releases all dynamic memory variables.

Syntax

```
CLEAR ALL
```

Description

The CLEAR ALL command closes all databases and associated files, like index and memo files, releases all dynamic memory variables (PRIVATE amd PUBLIC variables), closes alternate files, and finally selects work area 1 as current. As a result, all resources related to work areas and dynamic memory variables are released.

Note: GLOBAL, LOCAL and STATIC memory variables cannot be released with CLEAR ALL. Also, files opened with FOpen() remain open.

Info

See also:	CLEAR MEMORY, CLOSE, RELEASE, RESTORE, SAVE, USE, SET ALTERNATE, SET PRINTER
Category:	Memory commands
LIB:	xhb.lib
DLL:	xhbdll.dll

CLEAR GETS

Releases all Get objects in the current GetList array.

Syntax

```
CLEAR GETS
```

Description

CLEAR GETS terminates the READ command and releases all Get objects in the currently visible *GetList* array. An empty array is assigned to the PUBLIC variable *GetList*. This variable references an array containing all Get objects created with the @...GET command.

Info

See also: @...CLEAR, @...GET, CLOSE, READ, RELEASE, SET TYPEAHEAD
Category: Output commands, Get system
LIB: xhb.lib
DLL: xhbdll.dll

CLEAR MEMORY

Releases all dynamic memory variables.

Syntax

```
CLEAR MEMORY
```

Description

The CLEAR MEMORY command releases all dynamic memory variables (PRIVATE and PUBLIC variables). When the command is complete, neither symbolic names of PRIVATE and PUBLIC variables exist nor their values. This is different from the RELEASE command, which assigns the value NIL to dynamic variables currently visible.

Note: GLOBAL, LOCAL and STATIC memory variables are not affected by CLEAR MEMORY.

Info

See also: CLEAR ALL, CLEAR GETS, RELEASE, RESTORE, SAVE
Category: Memory commands
Source: rtl\memvars.c
LIB: xhb.lib
DLL: xhbdll.dll

CLEAR SCREEN

Clears the screen in text mode.

Syntax

```
CLEAR SCREEN

or

CLS
```

Description

CLEAR SCREEN is used in text-mode applications to erase the entire screen display and move the cursor to position 0,0.

Info

See also:	@...CLEAR, Scroll(), SetPos()
Category:	Output commands
Source:	rtl\scroll.c, rtl\setpos.c
LIB:	xhb.lib
DLL:	xhbdll.dll

CLEAR TYPEAHEAD

Empties the keyboard buffer.

Syntax

```
CLEAR TYPEAHEAD
```

Description

The CLEAR TYPEAHEAD command clears the keyboard buffer by removing all pending key strokes from the internal buffer. Key strokes are collected in this buffer, for example, when the user holds a key pressed and the application needs more time for processing key strokes than recording them.

CLEAR TYPEAHEAD is particularly used before calling a function that supports a default key handling, such as Achoice() or DbEdit() or MemoEdit(). When CLEAR TYPEAHEAD is called before such functions, they begin in a defined state.

Info

See also: HB_KeyPut(), Inkey(), KEYBOARD, SET TYPEAHEAD
Category: Input commands
Source: rtl\inkey.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates the effect of clearing the keyboard
// buffer

   PROCEDURE MAIN

      KEYBOARD "xHarbour"

      DO WHILE NextKey() <> 0          // produce screen output
         ? Chr( Inkey() )              // by polling the keyboard
      ENDDO                            // buffer

      KEYBOARD "xHarbour"
      CLEAR TYPEAHEAD

      DO WHILE NextKey() <> 0          // No screen output
         ? Chr( Inkey() )
      ENDDO

   RETURN
```

CLOSE

Closes one or more specified files.

Syntax

```
CLOSE [<cAlias> | ALL | ALTERNATE | DATABASES | FORMAT | INDEXES ]
```

Arguments

<cAlias>

This is the alias name of the work area whose files should be closed. It can be specified as a literal alias name or a character expression enclosed in parentheses.

ALL

This option closes all files in all work areas. Work areas become unused with all filters, indexes, relations and format definitions released. In addition, alternate files are closed.

ALTERNATE

Only the currently open alternate file is closed. The option has the same effect as issuing the SET ALTERNATE TO command without a file name.

DATABASES

All files associated with work areas are closed. This affects database, index and memo files. All work areas become unused with all filters, indexes and relations released. The current format definition stays intact.

FORMAT

Only the currently defined format definition is released. The option has the same effect as issuing the SET FORMAT TO command without a format name.

INDEXES

All index files open in the current work area are closed.

Description

The CLOSE command accepts various options to specify one of more files to close. When it is invoked without an option, the files in the current work area are closed. This has the same effect as the USE command called with no file name.

Info

See also:	DbCloseAll(), DbCloseArea(), QUIT, RETURN, SET ALTERNATE, SET INDEX, Set(), USE
Category:	Database commands, Environment commands
Source:	rdd\dbcmd.c, rtl\set.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates the effect of the CLOSE command
// together with different options

   PROCEDURE Main

      USE Address ALIAS Addr INDEX Addr1, Addr2

      ? IndexOrd()            // result: 1
```

```
    CLOSE INDEXES

    ? IndexOrd()              // result: 0
    ? Alias()                 // result: ADDR

    CLOSE Addr
    ? Alias()                 // result: ""

RETURN
```

CLOSE LOG

Closes all open log channels.

Syntax

```
CLOSE LOG
```

Description

The command closes all open log channels previously opened with INIT LOG.

Info

See also:	INIT LOG, SET TRACE
Category:	Log commands, xHarbour extensions
Header:	hblog.ch
Source:	rtl\hblog.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

COMMIT

Writes memory buffers of all used work areas to disk.

Syntax

```
COMMIT
```

Description

The COMMIT command flushes the memory buffers of all used work areas and writes them to disk. Data held in memory for database and index files are permanently written to the respective files. Changes to the data become visible to other processes in a network environment.

It is recommended to call COMMIT before UNLOCK. To flush the memory buffers of a single work area, call function DbCommit().

Info

See also: DbCommit(), DbCommitAll(), GO, SKIP
Category: Database commands
Source: rdd\dbcmd.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The examples demonstrates a basic technique for changing a record
// in a shared database:

   PROCEDURE Main
      LOCAL cFirstName := Space(40)
      LOCAL cLastName  := Space(50)

      USE Address ALIAS Addr SHARED NEW
      IF ! NetErr()
         SET INDEX TO AddressA, AddressB
      ENDIF

      CLS
      @ 10,5 SAY "Enter first name:" GET cFirstName
      @ 11,5 SAY "Enter last name :" GET cLastName
      READ

      IF .NOT. RLock()
         ? "Unable to lock record"
      ELSE
         REPLACE addr->FirstName WITH cFirstName  , ;
                 addr->LastName  WITH cLastName
         COMMIT
         UNLOCK
      ENDIF

      USE
   RETURN
```

CONTINUE

Resumes a pending LOCATE command.

Syntax

```
CONTINUE
```

Description

The CONTINUE command resumes a database search initiated with the LOCATE command in the current work area. The search is continued with the FOR condition specified with the last LOCATE command. The scope restriction and WHILE condition of LOCATE is ignored.

CONTINUE searches for the next record in the current work area that matches the FOR condition of the last LOCATE command issued for the current work area. If a match is found, the record pointer is positioned on this record and the Found() function returns .T. (true). When there is no match, the record pointer is positioned on the ghost record (Lastrec()+1), function Found() returns .F. (false) and function Eof() returns .T. (true).

Each work area can have its own LOCATE condition. It remains active until a new LOCATE command is executed or the work area is closed.

Info

See also:	DbSeek(), Eof(), Found(), LOCATE, SEEK
Category:	Database commands
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example uses LOCATE and CONTINUE to list customers
// living in the city of New York.

   PROCEDURE MAIN
      LOCAL cCity := "New York"
      FIELD FirstName, LastName

      USE Customer NEW

      LOCATE FOR Upper(FIELD->City) = Upper(cCity)

      DO WHILE Found()
         ? LastName, FirstName
         CONTINUE
      ENDDO

      CLOSE Customer
   RETURN
```

COPY FILE

Copies a file to a new file or to an output device.

Syntax

```
COPY FILE <cSourceFile> TO <cTargetFile>|<cDevice>
```

Arguments

`<cSourceFile>`

This is the name of the source file to copy. It must include path and file extension and can be specified as a literal file name or as a character expression enclosed in parentheses.

`TO <cTargetFile>`

This is the name of the target file to create. It must include path and file extension and can be specified as a literal file name or as a character expression enclosed in parentheses.

`TO <cDevice>`

This is the name of an output device, such as LPT1, for example. It can be specified as a literal device name or as a character expression enclosed in parentheses. When the device does not exist, a file with the name *<cDevice>* is created.

Description

COPY FILE is a file command that copies a source file to a new file or output device. When *<cSourceFile>* does not include drive and directory information, the file is searched first in the current directory and then in the directory specified with SET DEFAULT. If *<cSourceFile>* is not found, a runtime error is generated.

When *<cTargetFile>* exists, it is overwritten without warning.

Info

See also:	COPY TO, ERASE, FErase(), File(), FRename(), RENAME, SET DEFAULT
Category:	File commands
Source:	rtl\copyfile.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// This example demonstrates various possibilities of the COPY FILE
// command

   PROCEDURE Main
      LOCAL cFile := "TEST.TXT"

      COPY FILE (cFile) TO Tmp.txt

      COPY FILE Tmp.txt TO LPT1

   RETURN
```

COPY STRUCTURE

Copies the current database structure to a new database file.

Syntax

```
COPY STRUCTURE [FIELDS <fieldNames,...>] ;
            TO <cDatabase>
```

Arguments

`FIELDS <fieldNames>`

A comma separated list of literal field names, or character expressions enclosed in parentheses, specifies the database fields of the current work area to be included in the new database file. When no fields are specified, all fields are included.

`TO <cDatabase>`

This is the name of the database file to create. It can include path and file extension. When no path is given, the file is created in the current directory. The default file extension is DBF. *<cDatabase>* can be specified as a literal file name or as a character expression enclosed in parentheses.

Description

COPY STRUCTURE is a database command used for creating an empty database file based on the field definitions of the current work area. All fields of the current work area are used for the new database unless a list of field names is provided.

Should *<cDatabase>* already exist, it will be overwritten without warning.

Info

See also:	APPEND FROM, COPY STRUCTURE EXTENDED, CREATE, DbAppend(), DbCopyStruct(), DbCopyExtStruct(), DbCreate()
Category:	Database commands
Source:	rtl\dbstrux.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates a technique for creating a subset of records
// in a temporary database file.

   PROCEDURE Main
      USE Address NEW
      COPY STRUCTURE TO Temp

      USE Temp
      APPEND FROM Address FOR Trim(Upper(Address->City)) = "LONDON"

      < database operations with temporary file >

      CLOSE Temp
      ERASE Temp.dbf
   RETURN
```

COPY STRUCTURE EXTENDED

Copies field information to a new database file.

Syntax

```
COPY STRUCTURE EXTENDED TO <cDatabaseExt>
```

Arguments

TO <cDatabaseExt>

This is name of the database file to create. It can include path and file extension. When no path is given, the file is created in the current directory. The default file extension is DBF. *<cDatabaseExt>* can be specified as a literal file name or as a character expression enclosed in parentheses.

Description

The COPY STRUCTURE EXTENDED command copies field information of the current work area into a new database file. Field information is stored in the records of the new file and can be retrieved using the following pre-defined field names and database structure:

Fields in a structure extended file

Position	Field name	Type	Length	Decimals
1	FIELD_NAME	Character	10	0
2	FIELD_TYPE	Character	1	0
3	FIELD_LEN	Numeric	3	0
4	FIELD_DEC	Numeric	4	0

A database of this structure is called "structure extended" since it stores structural information of a database in its records. The CREATE FROM database command can then be used to create a database file of this structure programmatically.

Note: The length of a field of type Character is calculated as the sum of FIELD_LEN plus 256 * FIELD_DEC. The maximum length, however, is limited to 64k. Use a MEMO field to store longer Character strings.

Info

See also: COPY STRUCTURE, CREATE, CREATE FROM, DbCopyExtStruct(), DbCreate(), DbStruct(), FieldName(), Type()
Category: Database commands
Source: rtl\dbstrux.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The examples demonstrates a technique for changing the structure of
// an existing database file at runtime.

   PROCEDURE Main
      LOCAL cOldFile := "Address.dbf"
      LOCAL cBackup  := StrTran( cOldFile, ".dbf", ".bak" )

      USE (cOldFile)
      COPY STRUCTURE EXTENDED TO Tmp

      USE Tmp EXCLUSIVE
```

```
      APPEND BLANK
      REPLACE FIELD->FIELD_NAME WITH "Newfield"
      REPLACE FIELD->FIELD_TYPE WITH "C"
      REPLACE FIELD->FIELD_LEN  WITH 20
      REPLACE FIELD->FIELD_DEC  WITH 0
      CLOSE Tmp

      RENAME (cOldFile) TO (cBackup)
      CREATE (cOldFile) FROM Tmp

      APPEND FROM (cBackup)
      CLOSE (cOldFile)
      ERASE Tmp.dbf
   RETURN
```

COPY TO

Exports records from the current work area to a database or an ASCII text file.

Syntax

```
     COPY TO <targetFile> ;
     [FIELDS <fieldNames,...> ;]
    [<Scope> ; ]
      [WHILE <lWhileCondition> ;]
        [FOR <lForCondition> ;]
        [VIA <rddName>]
        [SDF | DELIMITED [WITH BLANK | TAB | PIPE | <xDelimiter> ] ]
   [CODEPAGE <cCodePage>] ;
 [CONNECTION <nConnection>]]
```

Arguments

`TO <targetFile>`

This is the name of the target file as a literal character string or a character expression enclosed in parentheses. When the file name is specified without a file extension, .dbf is assumed as default extension, unless the SDF or DELIMITED option is specified. In this case, the default file extension is .txt.

`FIELDS <fieldNames,...>`

The names of the fields to export to the external file can be specified as a comma separated list of literal field names or character expressions enclosed in parentheses. When this option is omitted, all fields of the source file are exported.

`<Scope>`

This option defines the number of records to export. It defaults to ALL. The NEXT <*nCount*> scope exports the next <*nCount*> records, while the RECORD <*nRecno*> scope exports only one record having the record number <*nRecno*>.

`WHILE <lWhileCondition>`

This is a logical expression indicating to continue exporting records while the condition is true. The COPY TO operation stops as soon as <*lWhileCondition*> yields .F. (false).

`FOR <lForCondition>`

This is a logical expression which is evaluated for all records in the current work area. Those records where <*lForCondition*> yields .T. (true) are exported.

`VIA <rddName>`

The VIA option specifies the replaceable database driver (RDD) to use for opening the target file. <*rddName*> is a character expression. If it is written as a literal name, it must be enclosed in quotes.

When no RDD is specified, the RDD active in the current work area is used to open the external file. If the VIA clause is used, the specified RDD must be linked to the application. Use the REQUEST command to force an RRD be linked.

`SDF`

The SDF option specifies the target file as an ASCII file in System Data Format. See file format description below.

`DELIMITED`

The DELIMETED option specifies the target file as a delimited ASCII file where field values are separated with a comma and Character values are enclosed with a delimiting character. The default delimiter for Character values is a double quotation mark.

`DELIMITED WITH BLANK | TAB | PIPE`

When the delimiter is specified as BLANK, field values in the new created ASCII text file are separated by one space and character fields are not enclosed in delimiters. Alternatively, the delimiting character between field values can be specified as TAB (Chr(9)) or PIPE (Chr(124)).

`DELIMITED WITH <xDelimiter>`

The WITH option specifies the delimiting character to enclose values of Character fields in. *<xDelimiter>* can be specified as a literal character or a character expression enclosed in parentheses.

<xDelimiter> can also be specified as an array with two elements: { *<cCharacterDelimiter>*, *<cFieldDelimiter>* }. If this option is used, the array must be enclosed in parentheses. It defines the delimiting characters for field values of type "C" and the delimiters between field values.

Important: If the DELIMITED WITH option is used in the COPY TO command, it must be placed as the last option in the command.

`CODEPAGE <cCodePage>`

This is a character string specifying the code page to use for character strings stored in the new database. It defaults to the return value of HB_SetCodePage().

`CONNECTION <nConnection>`

This option specifies a numeric server connection handle. It is returned by a server connection function which establishes a connection to a database server, such as SR_AddConnection() of the xHarbour Builder SQLRDD. When CONNECTION is used, the COPY TO command copies data to a database on the server.

Description

COPY TO is a database command used to export data of the current work are into a second file. Exported records are added to the end of the target file. The target file receives data from all fields of the current work area, unless a list of fields is explicitely specified.

The file in the current work area can be opened in SHARED mode. COPY TO creates the target file and opens it in EXCLUSIVE mode while data is being exported.

When the target file is an ASCII text file, all date values are written as a string in the format yyyymmdd. Other specifications for supported ASCII formats ar listed in the folowing tables:

SDF Text File

File Element	Format
Character fields	Padded with trailing blanks
Date fields	yyyymmdd
Logical fields	T or F
Memo fields	Ignored
Numeric fields	Padded with leading blanks or zeros
Field separator	None
Record separator	Carriage return/linefeed
End of file marker	0x1A or CHR(26)

DELIMITED Text File

File Element	Format
Character fields	May be delimited, with trailing blanks truncated
Date fields	yyyymmdd
Logical fields	T or F
Memo fields	Ignored
Numeric fields	Leading zeros may be truncated
Field separator	Comma
Record separator	Carriage return/linefeed
End of file marker	0x1A or CHR(26)

DELIMITED WITH BLANK Text File

File Element	Format
Character fields	Not delimited, trailing blanks may be truncated
Date fields	yyyymmdd
Logical fields	T or F
Memo fields	Ignored
Numeric fields	Leading zeros may be truncated
Field separator	Single blank space
Record separator	Carriage return/linefeed
End of file marker	0x1A or CHR(26)

Deleted records: Records that are marked for deletion in the current work area are ignored when SET DELETED is set to ON. These records are not exported. With SET DELETED OFF, all records matching the import scope are exported and the deleted flag is set accordingly in the target file.

Info

See also: APPEND FROM, COPY FILE, COPY STRUCTURE
Category: Database commands
Source: rdd\dbcmd.c, rtl\dbdelim.prg
LIB: xhb.lib
DLL: xhbdll.dll

Examples

```
// The example creates a subset of records from a database
// within an intermediate file.

   PROCEDURE Main

      USE Address NEW

      COPY TO Tmp FOR "LONDON" $ Upper(CITY)
      USE Tmp

      Browse()
      USE

      ERASE Tmp.dbf                  // delete intermediate file
   RETURN

// The example creates different delimited ASCII files.

   PROCEDURE Main

      USE Address NEW
```

```
    // Creates a regular DELIMITED ASCII file
    COPY TO Test.txt DELIMITED

    // Uses Chr(9) as field delimiter
    COPY TO Test1.txt DELIMITED WITH TAB

    // Uses "|" as field delimiter
    COPY TO Test2.txt DELIMITED WITH PIPE

    // Encloses character values in single quotes and separates
    // fields with Chr(255)
    COPY TO Test3.txt DELIMITED WITH ( {"'", Chr(255) } )

    USE
 RETURN
```

COUNT

Counts records in the current work area.

Syntax

```
COUNT TO <varName> ;
        [<Scope>] ;
  [WHILE <lWhileCondition>] ;
    [FOR <lForCondition>]
```

Arguments

`TO <varName>`

The name of the variable that is assigned the result of the COUNT operation. When *<varName>* does not exist, it is created as a PRIVATE variable.

`<Scope>`

This option defines the number of records to process. It defaults to ALL. The NEXT *<nCount>* scope processes the next *<nCount>* records.

`WHILE <lWhileCondition>`

This is a logical expression indicating to continue counting while the condition is true. The COUNT operation stops as soon as *<lWhileCondition>* yields .F. (false).

`FOR <lForCondition>`

This is a logical expression which is evaluated for all records in the current work area. Those records where *<lForCondition>* yields .T. (true) are included in the count.

Description

The COUNT command is used to count the number of records in the current work area based on the result of one or more logical expressions. The expressions are evaluated in the current work area. The number of records to include in the count can be restricted using a FOR and/or WHILE condition or by explicitely defining a scope.

Info

See also:	AVERAGE, DbEval(), SUM, TOTAL, UPDATE
Category:	Database commands
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example counts the number of sales that exceed a limit

   PROCEDURE Main
      LOCAL nSales := 0

      USE Sales NEW
      COUNT TO nSales FOR FIELD->Amount * FIELD->Price > 500
      USE

      ? "Sales of more than 500:", nSales
   RETURN
```

CREATE

Creates and opens an empty structure extended database file.

Syntax

```
CREATE <cDatabaseExt>
```

Arguments

<cDatabaseExt>

This is the name of the database file to create. It can include path and file extension. When no path is given, the file is created in the current directory. The default file extension is DBF. *<cDatabaseExt>* can be specified as a literal file name or as a character expression enclosed in parentheses.

Description

The CREATE command creates a new database file and opens it exclusively in the current work area. The database has the following pre-defined field names and database structure:

Fields in a structure extended file

Position	Field name	Type	Length	Decimals
1	FIELD_NAME	Character	10	0
2	FIELD_TYPE	Character	1	0
3	FIELD_LEN	Numeric	3	0
4	FIELD_DEC	Numeric	4	0

A database of this structure is called "structure extended" since it stores structural information of a database in its records. The CREATE FROM database command can then be used to create a database file of this structure programmatically.

Note: The length of a field of type Character is calculated as the sum of FIELD_LEN plus 256 * FIELD_DEC. The maximum length, however, is limited to 64k. Use a MEMO field to store longer Character strings.

Info

See also: COPY STRUCTURE EXTENDED, CREATE FROM, DbCopyStruct(), DbCopyExtStruct(), DbCreate(), DbStruct()
Category: Database commands
Source: rtl\dbstrux.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates how to create a database file programmatically
// using a structure extended file.

   PROCEDURE Main
      CREATE Temp
      APPEND BLANK

      REPLACE FIELD_NAME WITH "Lastname" , ;
              FIELD_TYPE WITH "C"  , ;
              FIELD_LEN  WITH 25   , ;
              FIELD_DEC  WITH 0
      CLOSE Temp
```

```
      CREATE Address FROM Temp
   RETURN
```

CREATE FROM

Creates a new database file from a structure extended file.

Syntax

```
      CREATE <cDatabase> FROM <cDatabaseExt> [NEW] ;
      [ALIAS <cAlias>] ;
        [VIA <rddName>]
   [CODEPAGE <cCodePage>] ;
[CONNECTION <nConnection>]]
```

Arguments

CREATE <cDatabase>

This is name of the database file to create. It can include path and file extension. When no path is given, the file is created in the current directory. The default file extension is DBF. *<cDatabase>* can be specified as a literal file name or as a character expression enclosed in parentheses.

FROM <cDatabaseExt>

This is name of the structure extended database file that holds field information for the database to create. *<cDatabaseExt>* can be specified as a literal file name or as a character expression enclosed in parentheses.

NEW

The newly created file is automatically opened in a new work area when the NEW option is specified. Omitting this option causes the file be opened in the current work area.

ALIAS <cAlias>

Optionally, the alias name for the work area of *<cDatabase>* can be specified. It defaults to the file name of *<cDatabase>*. If specified, the alias name must begin with an underscore or an alphabetic character, followed by a series of underscores, digits or alphabetic characters,

VIA <rddName>

The VIA option specifies the replaceable database driver (RDD) to use for opening the new database file. *<rddName>* is a character expression. If it is written as a literal name, it must be enclosed in quotes. If omitted, *<rddName>* defaults to the return value of RddSetDefault().

CODEPAGE <cCodePage>

This is a character string specifying the code page to use for character strings stored in the new database. It defaults to the return value of HB_SetCodePage().

CONNECTION <nConnection>

This option specifies a numeric server connection handle. It is returned by a server connection function which establishes a connection to a database server, such as SR_AddConnection() of the xHarbour Builder SQLRDD. When CONNECTION is used, the CREATE FROM command creates a database on the server.

Description

The CREATE FROM command uses field information stored in the records of a structure extended file to create a new database file. A structure extended datbase file has the following pre-defined field names and structure:

Fields in a structure extended file

Position	Field name	Type	Length	Decimals
1	FIELD_NAME	Character	10	0
2	FIELD_TYPE	Character	1	0
3	FIELD_LEN	Numeric	3	0
4	FIELD_DEC	Numeric	4	0

CREATE FROM reads field information from the structure extended file and creates from it a new database file. A structure extended file may have more fields than listed in the table, but only the listed ones are required.

The new database file is automatically opened.

Note: The length of a field of type Character is calculated as the sum of FIELD_LEN plus 256 * FIELD_DEC. The maximum length, however, is limited to 64k. Use a MEMO field to store longer Character strings.

Info

See also: COPY STRUCTURE, COPY STRUCTURE EXTENDED, CREATE, DbCopyStruct(), DbCreate(), DbStruct()
Category: Database commands
Source: rtl\dbstrux.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// please refer to the COPY STRUCTURE EXTENDED example for a usage
// scenario of CREATE FROM.
```

DEFAULT TO

Assigns a default value to a NIL argument.

Syntax

```
DEFAULT <varName1> TO <xValue1> ;
     [, <varNameN> TO <xValueN> ]
```

Arguments

`<xValue>`

This is an expression whose value is assigned to <*varName*> when the variable contains NIL.

`<varName>`

The symbolic names of the variable to assign a default value to.

Description

The DEFAULT TO command assigns the value of <*xValue*> to the variable <*varName*> when <*varName*> contains NIL. Multiple variables can be assigned default values by separating a list of TO options with commas.

The command is used in functions and procedures that accept optional arguments. Optional arguments receive NIL as value and are assigned default values using DEFAULT TO.

Info

See also:	:=, Default(), PValue(), IF, STORE, Valtype()
Category:	Memory commands
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows a function that normalizes strings in
// length and capitilization.

   FUNCTION Normalize( cStr, nPad )

       DEFAULT cString TO "" , ;
               nPad    TO Len(cStr)

   RETURN PadR( Upper( cStr[1] ) + Lower( SubStr( cStr, 2 ), nPad )
```

DELETE

Marks one or more records for deletion.

Syntax

```
DELETE [<Scope>]
[WHILE <lWhileCondition>]
  [FOR <lForCondition>]
```

Arguments

<Scope>

This option defines the number of records to mark for deletion. It defaults to the current record. The NEXT <*nCount*> scope deletes the next <*nCount*> records, while the RECORD <*nRecno*> scope deletes only one record having the record number <*nRecno*>. The option ALL marks all records for deletion.

WHILE <lWhileCondition>

This is a logical expression indicating to continue marking records for deletion while the condition is true. The DELETE command stops as soon as <*lWhileCondition*> yields .F. (false).

FOR <lForCondition>

This is a logical expression which is evaluated for all records in the current work area. Those records where <*lForCondition*> yields .T. (true) are marked for deletion.

Description

The DELETE command marks one or more records in the current work area for deletion. Records with the deletion flag set become unvisible when SET DELTED is set to ON. Whether or not the deleted flag is set for the current record can be tested with the Deleted() function. The flag marks a record only logically for deletion, it does not remove s record physically from a database file. The logical deletion flag can be removed with the RECALL command.

When a database is open in SHARED mode, the current record must be locked with the RLock() function before DELETE may be called. Failure to do so generates a runtime error. If multiple records should be marked for deletion, the entire file must be locked using FLock(), or the file must be open in EXCLUSIVE mode.

The PACK command removes all records with the deletion flag set physically from a database. The ZAP command removes all records at once, leaving an empty database file.

Note: When the current record is marked for deletion and SET DELETED is ON, the record remains visible until the record pointer is moved.

Info

See also:	DbDelete(), DbEval(), DbRecall(), Deleted(), FLock(), PACK, RECALL, RLock(), SET DELETED, ZAP
Category:	Database commands
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates the effect of SET DELETED with a record
// marked for deletion

   PROCEDURE Main
```

```
   USE Customer EXCLUSIVE

   GO TOP
   ? Deleted()                    // result; .F.
   ? Recno()                      // result: 1
   DELETE
   SET DELETED ON

   GO TOP
   ? Deleted()                    // result; .F.
   ? Recno()                      // result: 2

   SET DELETED OFF
   GO TOP
   ? Deleted()                    // result; .T.
   ? Recno()                      // result: 1

   RECALL
   ? Deleted()                    // result; .F.

   CLOSE Customer
RETURN
```

DELETE FILE

Deletes a file from disk.

Syntax

```
DELETE FILE <cFilename>
ERASE <cFilename>
```

Arguments

`<cFilename>`

This is name of the file to delete. It must include path and file extension and can be specified as a literal file name or as a character expression enclosed in parentheses. The path can be omitted from *<cFilename>* when the file resides in the current directory.

Description

DELETE FILE, or its synonym ERASE, is a file command that deletes a file from disk. When *<cFilename>* does not include drive and directory information, the file is searched only in the current directory. Directories specified with SET DEFAULT or SET PATH are ignored by this command.

The file must be closed before it can be deleted.

When the file *<cFilename>* is not found or when it is in use, either a runtime error is generated, or a file error occurs that can be queried with function FError().

Info

See also:	CLOSE, CurDir(), FErase(), FError(), File(), FRename(), RENAME, USE
Category:	File commands
Source:	rtl\philes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example deletes a group of index files in the current directory.

   #include "Directry.ch"

   PROCEDURE Main
      LOCAL aDir := Directory( "*.ntx" )
      LOCAL i, imax := Len( aDir )

      FOR i:=1 TO imax
         DELETE FILE ( aDir[i,F_NAME] )
      NEXT
   RETURN
```

DELETE TAG

Deletes a tag from an index.

Syntax

```
DELETE TAG <cIndexTag1> [IN <cIndexFile1>]
        [, <cIndexTagN> [IN <cIndexFileN>] ]
```

Arguments

<cIndexTag>

This is the symbolic name of the index to remove from an index file. It can be specified as a literal name or a character expression enclosed in parentheses.

<cIndexFile>

<cIndexFile> is the name of the file containing the index to remove. If not specified, all index files open in the current work area are searched for the index with the name *<cIndexTag>*. The file name can be specified as a literal name or a character expression enclosed in parentheses. When the file extension is omitted, it is determined by the database driver that opened the file.

Description

The DELETE TAG command deletes an index from an index file in the current work area. The index is identified by its tag name, which is equivalent to an alias name of a work area. If no index file is specified, all indexes in the current work area are searched for a matching tag name. When the tag name does not exist or the index file is not open, a runtime error is generated.

If the index to delete is the controlling index, the records in the current work area are listed in physical order after index deletion.

Note: The number of indexes that can be stored in an index file is determined by the replaceable database driver maintaining a work area. While DBFNTX and DBFNDX support only one index per file, the DBFCDX RDD supports many indexes in one index file.

Info

See also:	INDEX, OrdDestroy(), SET ORDER, SET INDEX
Category:	Database commands, Index commands
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example creates three indexes in one file using the DBFCDX driver.
// The effect of deleting the controlling index is demonstrated.

   REQUEST DBFCDX

   PROCEDURE Main
      RddSetDefault( "DBFCDX" )

      USE Customer
      INDEX ON Upper(FirstName) TAG FName TO Cust01
      INDEX ON Upper(LastName)  TAG LName TO Cust01
      INDEX ON Upper(City)      TAG City  TO Cust01

      FOR n:=1 TO 3
         ? OrdName(n), OrdKey(n)
```

```
      NEXT
      WAIT

      SET ORDER TO TAG City
      Browse()

      DELETE TAG City
      Browse()

      USE
   RETURN
```

DO

Executes a function or procedure.

Syntax

```
DO <ProcedureName> [WITH <params,...>]
```

Arguments

`<ProcedureName>`

This is the symbolic name of the function or procedure to execute.

`WITH <params,...>`

Optionally, a comma separated list of parameters to pass to the function or procedure can be specified following the WITH option.

Description

The DO command exists for compatibility reasons. It executes the function or procedure having the symbolic name *<ProcedureName>* and passes the parameters specified as *<params,...>* on to it.

Note: it is recommended to use the execution operator instead of the DO command.

Info

See also: ( ), FUNCTION, PROCEDURE
Category: Statements
Source: vm\do.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates two possibilities of calling
// a subroutine. The first uses the DO command while the
// second uses the () execution operator.

   PROCEDURE Main

      // Procedural syntax
      DO Test WITH "Hello", "World"

      // Functional syntax
      Test( "Hi", "There" )

   RETURN


   PROCEDURE Test( c1, c2 )
      ? c1, c2
   RETURN
```

EJECT

Ejects the current page from the printer.

Syntax

```
EJECT
```

Description

The EJECT command ejects the current page from the printer by sending a form feed control character (Chr(12)) to the printer. In addition, the functions for maintaining the current postion of the printhead are reset to zero (see PCol() and PRow()).

If a printer row is addressed with @...SAY that is smaller than the current printhead row, EJECT is automatically executed. To suppress this automatic formfeed, use SetPrc() to reset the internal printer row and column counters.

Info

See also:	DevPos(), IsPrinter(), PCol(), PRow(), SET CONSOLE, SET DEVICE, SET PRINTER, SetPrc()
Category:	Console commands, Printer commands
Source:	rtl\console.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The exampledemonstrates a simple printing routine that lists
// data from a customer database

   PROCEDURE Main
      LOCAL nLines := 50
      LOCAL nPages := 1

      USE Customer NEW
      SET PRINTER ON
      SET PRINTER TO LPT1

      DO WHILE ! Eof()
         ? "Page: "+Str(nPages,8)
         ? "Date: "+DtoC(Date())

         DO WHILE .NOT. Eof() .AND. PRow() <= nLines
            ? LastName, FirstName, Phone
            SKIP
         ENDDO

         nPages ++
         EJECT            // this resets PRow()
      ENDDO

      SET PRINTER OFF

   RETURN
```

ERASE

Deletes a file from disk.

Syntax

```
ERASE <cFilename>
DELETE FILE <cFilename>
```

Arguments

`<cFilename>`

This is name of the file to delete. It must include path and file extension and can be specified as a literal file name or as a character expression enclosed in parentheses. The path can be omitted from *<cFilename>* when the file resides in the current directory.

Description

ERASE, or its synonym DELETE FILE, is a file command that deletes a file from disk. When *<cFilename>* does not include drive and directory information, the file is searched only in the current directory. Directories specified with SET DEFAULT or SET PATH are ignored by this command.

The file must be closed before it can be deleted.

When the file *<cFilename>* is not found or when it is in use, either a runtime error is generated, or a file error occurs that can be queried with function FError().

Info

See also:	CLOSE, CurDir(), FErase(), FError(), File(), FRename(), RENAME, USE
Category:	File commands
Source:	rtl\philes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example deletes a group of index files in the current directory.

   #include "Directry.ch"

   PROCEDURE Main
      LOCAL aDir := Directory( "*.ntx" )
      LOCAL i, imax := Len( aDir )

      FOR i:=1 TO imax
         ERASE ( aDir[i,F_NAME] )
      NEXT
   RETURN
```

FIND

Searches an index for a specified key value.

Syntax

```
FIND <cSearchString>
```

Arguments

```
<cSearchString>
```

<cSearchString> is a search condition to match agains the index keys. The search string can be specified by using a literal string or a character expression enclosed in parentheses.

Description

The FIND command exists for compatibility reasons and is not recommended. It is superseded by the SEEK command

Info

See also:	DbSeek(), SEEK, SET INDEX, SET ORDER
Category:	Database commands
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// refer to the example with SEEK
```

GO

Moves the record pointer to a specified position.

Syntax

```
GO[TO] <nRecno> | BOTTOM | TOP
```

Arguments

<nRecno>

This is a numeric expression indentifying the record number to move the record pointer to.

BOTTOM

This option positions the record pointer on the last logical record.

TOP

This option positions the record pointer on the first logical record.

Description

The GO[TO] command positions the record pointer in the current work area to the specified position. The TOP and BOTTOM positions refer to the first and last logical records, while a numeric record number identifies a physical record.

If <*nRecno*> is smaller than 1 or larger than Lastrec(), the record pointer is positioned on the ghost record located at position Lastrec()+1.

Note: Logical conditions restricting visibility of records, like SET DELETED, SET FILTER or SET SCOPE, are only taken into account with the TOP and BOTTOM options of the GO command. Positioning the record pointer to a physical record number ignores such conditions.

Info

See also:	Bof(), DbGoto(), Eof(), LastRec(), RecNo(), SET DELETED, SET FILTER, SET RELATION, SET SCOPE, SKIP
Category:	Database commands
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates logical and physical positioning of the
// record pointer.

   PROCEDURE Main
      USE Customer
      DELETE
      ? Recno(), Deleted()             // result: 1  .T.

      SET DELETED ON
      GO TOP                           // Logical positioning
      ? Recno(), Deleted()             // result: 2  .F.

      GOTO 1                           // Physical positioning
      ? Recno(), Deleted()             // result: 1  .T.

      USE
   RETURN
```

INDEX

Creates an index and/or index file.

Syntax

```
    INDEX ON <indexExpr> ;
        [TAG <cIndexName>] ;
         [TO <cIndexFile>] ;
        [FOR <lForCondition>] ;
      [WHILE <lWhileCondition>] ;
        [ALL] ;
       [NEXT <nNumber>] ;
     [RECORD <nRecNo>] ;
       [REST] ;
       [EVAL <bBlock>] ;
      [EVERY <nInterval>] ;
     [UNIQUE] ;
 [ASCENDING|DESCENDING] ;
[USECURRENT] ;
  [ADDITIVE] ;
    [CUSTOM] ;
[NOOPTIMIZE] ;
 [TEMPORARY] ;
 [USEFILTER] ;
 [EXCLUSIVE]

Note: the TAG and TO clauses are optional, but
it is necessary to specify at least one of them.
```

Arguments

ON <indexExpr>

This is an expression which is evaluated for the records in the current work area. The value of *<indexExpr>* determines the logical order of records when the index is the controlling index. The data type of the index may be Character, Date, Numeric or Logical. The maximum length of an index expression and its value is determined by the replaceable database driver used to create the index.

Important: When the index expression yields values of data type Character, the index value must be of constant length. The functions Trim() or Ltrim() cannot be used in an index expression since they result in a corrupt index.

TAG <cIndexName>

This is the symbolic name of the index to create in an index file. It can be specified as a literal name or a character expression enclosed in parentheses.

TO <cIndexFile>

<cIndexFile> is the name of the file that stores the new index. The file name can be specified as a literal name or a character expression enclosed in parentheses. When the file extension is omitted, it is determined by the database driver that creates the file.

Although both TAG and TO clauses are optional, at least one of them must be used.

FOR <lForCondition>

This is an optional logical expression which is evaluated for all records in the current work area. Those records where *<lForCondition>* yields .T. (true) are included in the index. The FOR condition is stored in the index file and is maintained by the database driver when records are

updated. That is, if a record is changed so that it does not match the FOR condition, it is removed from the index.

The FOR expression cannot exceed 250 characters in length. RDDs that do not support a FOR condition when creating indexes generate a runtime error when this option is used.

`WHILE <lWhileCondition>`

This is a logical expression indicating to continue index creation while the condition is true. The INDEX command stops evaluating records as soon as *<lWhileCondition>* yields .F. (false).

Unlike the FOR expression, which is stored in the index file and exists throughout the lifetime of an index, the WHILE condition is only evaluated during index creation. It is discarded when the index is complete.

RDDs that do not support a FOR condition when creating indexes generate a runtime error when this option is used.

`ALL`

The option specifies that *<indexExpr>* should be evaluated with all records in the current work area. It is the default scope option if the INDEX command.

`NEXT <nNumber>`

This option restricts the number of records to evaluate during index creation to *<nNumber>*.

`RECORD <nRecNo>`

This option adds only the record with the record number *<nRecNo>* to the index.

`REST`

The REST scope instructs the database driver to evaluate *<indexExpr>* for the records beginning with the current record down to the end of file.

`EVAL <bBlock>`

This parameter is a code block that must return a logical value. The indexing operation continues as long as Eval(*<bBlock>*) returns .T. (true). The operation is stopped when Eval(*<bBlock>*) returns .F. (false).

The code block is evaluated for each record unless the EVERY option is specified. It is recommened to use EVERY in conjunction with EVAL.

`EVERY <nInterval>`

This option is a numeric value specifying the number of records after which the EVAL block is evaluated. This can improve the indexing operation especially for large databases. EVERY is ignored when no EVAL block is supplied.

`UNIQUE`

The option suppresses inclusion of records that yield duplicate index values. When an index value exists already in an index, a second record resulting in the same index value is not added to the index.

`ASCENDING|DESCENDING`

These options are mutually exclusive. They specify if the index is created in ascending or descending order. If not specified, ASCENDING is the default.

When the ASCENDING or DESCENDING option is not supported by the database driver, a runtime error is generated.

USECURRENT

The option instructs the database driver to use the current logical order of records for navigating the database during index creation. The logical order is determined by the controlling index and the SET SCOPE restriction.

When the USECURRENT clause is omitted, the records in the current work area are evaluated in physical order.

ADDITIVE

The option makes sure that index files remain open during index creation. If not specified, all files but <*cIndexFile*> are closed prior to indexing.

CUSTOM

This option creates an empty index that is custom built. Index entries must be added or deleted explicitely using functions OrdKeyAdd() and OrdKeyDel(). RDDs that support custom indexes do not add or delete keys automatically.

NOOPTIMIZE

This specifies a non-optimized FOR condition. If not specified, it will be optimized if the RDD supports it.

TEMPORARY

If this option is specified, a temporary index is created which is automatically destroyed when the index is closed. The temporary index may be created in memory only or in a temporary file. This lies in the responsibility of the RDD used for index creation.

USEFILTER

This option instructs the RDD to recognize a filter condition active in the current work area that is set with SET FILTER or SET DELETED. In this case, filtered records are not included in the index.

EXCLUSIVE

With this option, the index file is created in EXCLUSIVE file access mode. By default, the index file is created in SHARED mode.

Description

The INDEX command is used to created indexes and/or index files. An index is stored in an index file which is maintained separately from a database. Indexes provide for logical ordering of records in a work area by specifying the expression <*indexExpr*>. An index file is created by the INDEX command and the index expression is evaluated with the records in the current work area. The resulting index value is stored in sorted order in an index file.

With RDDs that support multiple indexes per index file, such as DBFCDX, indexes are added to <*cIndexFile*> when this file is open in the current work area.

The commands SET INDEX and SET ORDER are used to open an existing index file and select an index as the controlling one. When an index is selected as the controlling index, database navigation with SKIP occurs in logical index order and not in the physical order of records as they are stored in the database.

Open index files can be re-organized using the REINDEX command. Note, however, that only the FOR condition and the ASCENDING/DESCENDING option are maintained with REINDEX. All other options available with the INDEX command are not used with REINDEX.

Info

See also: CLOSE, DbOrderInfo(), Descend(), DtoS(), IndexKey(), IndexOrd(), OrdCreate(), OrdKeyNo(), REINDEX, SEEK, SET INDEX, SET ORDER, SORT, USE
Category: Database commands, Index commands
Source: rdd\dbcmd.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example creates three indexes in one index file and
// browses the customer database in the index order LName

   REQUEST DBFCDX

   PROCEDURE Main
       RddSetDefault( "DBFCDX" )
       USE Customer

       INDEX ON Upper(FirstName) TAG FName TO Cust01
       INDEX ON Upper(LastName)  TAG LName TO Cust01
       INDEX ON Upper(City)      TAG City  TO Cust01

       SET ORDER TO TAG LName
       Browse()

       USE
   RETURN
```

INIT LOG

Initializes the Log system and opens requested log channels.

Syntax

```
INIT LOG ;
[CONSOLE(...)] ;
  [DEBUG(...)] ;
  [EMAIL(...)] ;
   [FILE(...)] ;
[MONITOR(...)] ;
 [SYSLOG(...)] ;
   [NAME <cExeName> ] ;
```

Arguments

`(...)`

The INIT LOG command opens one or more log channels to receive log messages via the LOG command. Each log channel can be configured using a list of additional parameters enclosed in parentheses. The first parameter for all log channel options defines the priority a log message must have at minimum for a log channel to receive the message. #define constants are available in the file HbLogDef.ch that can be used for the priority:

Log channel priorities

Constant	Value	Description
HB_LOG_DEFAULT	-1	All messages are logged
HB_LOG_CRITICAL	1	Critical log messages
HB_LOG_ERROR	2	Error log messages
HB_LOG_WARNING	3	Warning messages
HB_LOG_INFO *)	4	Informational messages
HB_LOG_DEBUG	5	Debug messages

**) default*

A log channel created with the priority HB_LOG_WARNING receives log messages of priority HB_LOG_DEFAULT, HB_LOG_CRITICAL, HB_LOG_ERROR and HB_LOG_WARNING, but it does not receive log messages of priority HB_LOG_INFO or HB_LOG_DEBUG. Thus, different log channels can receive log messages of different priority.

`CONSOLE( [<nPrioCon>] )`

Log messages are displayed on the console. This option accepts only the priority level as additional parameter.

`DEBUG( [<nPrioDbg>], [<nMaxPrio>] )`

This option opens a log channel which sends received log messages to a system debugger. If there is no system debugger, this log channel does nothing. *<nPrioDbg>* is the priority of the log channel, while *<nMaxPrio>* is an optional upper limit for the log message priority to be sent to the debugger.

`EMAIL( [<nPrioMail>],[<cHelo>],<cServer>,<cMailTo>,[<cSubj>],[<cFrom>] )`

This log channel sends log messages per eMail and accepts the most additional parameters. The configuration parameters for the eMail log channel are as follows.

Configuration of the eMail log channel

Parameter		Description
<nPrioMail>	optional	Numeric priority level of log channel
<cHelo>	optional	Character string sent with the HELO command of the SMTP protocol.
<cServer>	required	Character string specifying the mail server.
<cMailTo>	required	Character string specifying the mail address of the receipient.
<cSubject>	optional	Character string specifying the subject line of the eMail.
<cFrom>	optional	Character string specifying the mail address of the sender.

```
FILE( [<nPrioFile>],<cFileName>,[<nFileSize>],[<nBackups>] )
```

This opion defines *<cFileName>* as the name of a log file where log messages are written into. Optionally, the size of the log file can be restricted to a maximum file size (numeric *<nFileSize>*). If a log file grows beyond *<nFileSize>*, it is renamed, and a new file *<cFileName>* is created. The renamed file receives an extension made up of digits from .001 to .999, so that existing log files are not overwritten. In addition, the number of automatically created backup files is limited to *<nBackups>* files, if this numeric parameter is given.

```
MONITOR( [<nPrioMon>],[<nPort>] )
```

This option creates a log channel that sends log messages to an IP port, so that a remote monitoring process can receive log messages. By default, xHarbour applications using this option listen on *<nPort>* for incoming connections and send a log message via this port to a remote process. The default port *<nPort>* used by this log channel is 7761. That is, a remote process must connect to the message sending process using the same port number. Refer to INetRecv() for more information on how to obtain the messages sent via the MONITOR log channel in a remote monitoring process.

```
SYSLOG( [<nPrioSys>], <nSysID> )
```

With this option, a log channel is created that sends log messages to the operating system's log file. The log channel must be created with a unique numeric *<nSysID>*.

```
NAME <cExeName>
```

This is an optional character string holding the name of the EXE file creating log messages.

Description

INIT LOG is a powerful command initializing xHarbour's Log system and creating up to six different log channels. After initilization, the LOG command is used to send log messages to all open log channels, each of which serves its own output device or communication channel. This allows for displaying log messages on the screen (CONSOLE), collecting them in a local file (FILE) or the system log file (SYSLOG), posting them to a system debugger (DEBUG), sending them per eMail (EMAIL), or monitoring them from a remote process via TCP/IP (MONITOR).

The first log message is created by INIT LOG. It simply indicates that logging has started so that the application NAME appears as first entry in the log. Subsequent log messages must be created with the LOG command. The last log entry is created by the CLOSE LOG command, after which the Log system becomes unavailable. After CLOSE LOG is executed, INIT LOG must be called again to reinitialize the Log system.

Log messages can be formatted to include default information, such as date and time stamps of the log entry. Several output formats for a log entry are predefined and can be selected using the SET LOG STYLE command.

Info

See also: CLOSE LOG, LOG, SET LOG STYLE, SET TRACE, TraceLog()
Category: Log commands, xHarbour extensions
Header: hblog.ch, hblogdef.ch
Source: rtl\hblog.prg, rtl\hblognet.prg
LIB: xhb.lib
DLL: xhbdll.dll

Examples

```
// The example opens three log channels and logs entered
// data to the console, a local file and a remote monitoring
// process.

   #include "Inkey.ch"
   #include "Hblog.ch"

   PROCEDURE Main
      LOCAL cData

      INIT LOG Console() ;
               Monitor() ;
                  File( NIL, "Testlog.log" )

      DO WHILE .T.
         CLS

         // let the user enter some data
         cData := Space(80)
         @ 2,2 SAY "Enter a string:" GET cData PICTURE "@S30"
         READ

         IF Lastkey() == K_ESC
            EXIT
         ENDIF

         CLS
         LOG Trim(cData)
         WAIT "Press a key..."

         IF Lastkey() == K_ESC
            EXIT
         ENDIF
      ENDDO

      CLOSE LOG
   RETURN
```

Monitoring log messages

```
// This example implements a simple monitoring process for
// log messages created in the first example.

   #include "Inkey.ch"

   PROCEDURE Main( cServerIPAddress )
      LOCAL pSocket, cData, nBytes

      IF Empty( cServerIPAddress )
         // IP address of the local station
```

```
      cServerIPAddress := "127.0.0.1"
   ENDIF

   INetInit()
   CLS

   ? "Connecting..."

   DO WHILE Inkey(1) <> K_ESC
      // connect to log message server on the default port 7761
      pSocket := INetConnect( cServerIPAddress, 7761 )

      IF INetErrorCode( pSocket ) <> 0
         ? "Socket error:", INetErrorDesc( pSocket )
         ?? " press ESC to quit"
         LOOP
      ENDIF

      EXIT
   ENDDO

   IF Lastkey() == K_ESC
      QUIT
   ENDIF

   ? "Log monitor up and running"

   DO WHILE Inkey(.1) <> K_ESC
      // wait 2 seconds for incoming data
      IF InetDataReady( pSocket, 2000 ) > 0
         cData := Space(4096)
         nBytes:= INetRecv( pSocket, @cData, Len( cData ) )
      ELSE
         nBytes := 0
      ENDIF

      IF nBytes > 0
         // a log message is received
         ? Left( cData, nBytes )

      ELSEIF INetErrorCode( pSocket ) <> 0
         ? "Socket error:", INetErrorDesc( pSocket )

         IF INetErrorCode( pSocket ) == -2
            // Connection closed on remote site
            EXIT
         ELSE
            ?? " press ESC to quit"
         ENDIF

      ELSE
         // Indicate that I'm still alive
         ?? "."
      ENDIF
   ENDDO

   // disconnect from server
   INetClose( pSocket )

   INetCleanUp()
RETURN
```

INPUT

Assigns the result of an input expression to a variable.

Syntax

```
INPUT [<xMessage>] TO <cVarName>
```

Arguments

`<xMessage>`

This is an optional expression of data type C, D, L or N. The value of *<xMessage>* is displayed before the user can input an expression.

`<cVarName>`

This is a character string holding the name of the variable to assign the result of user input to. It must be of PRIVATE or PUBLIC scope. If *<cVarName>* does not exist, a PRIVATE variable is created.

Description

The INPUT command exists for compatibility reasons only. It is a very basic text-mode command that allows a user for entering an expression. The input character string is compiled with the macro operator (&) and the result of the compiled expression is assigned to a single memory variable.

During user input only the Return key is recognized as termination key. The Escape key does not terminate the INPUT command.

Note: if an invalid expression is entered, a runtime error is generated.

Info

See also: ACCEPT
Category: Input commands
Source: rtl\input.prg
LIB: xhb.lib
DLL: xhbdll.dll

JOIN

Merges records of two work areas into a new database.

Syntax

```
JOIN WITH <cAlias> ;
       TO <cDatabase> ;
      FOR <lForCondition> ;
  [FIELDS <fieldNames,...>]
```

Arguments

`WITH <cAlias>`

This is the alias name of the second work area to merge records of the current work area with. It can be specified as a literal name or as a character expression enclosed in parentheses.

`TO <cDatabase>`

This is name of the database file to create. It can include path and file extension. When no path is given, the file is created in the current directory. The default file extension is DBF. *<cDatabase>* can be specified as a literal file name or as a character expression enclosed in parentheses.

`FOR <lForCondition>`

This is a logical expression which is evaluated for all records during the JOIN operation. Those records where *<lForCondition>* yields .T. (true) are included in the resulting database.

`FIELDS <fieldNames>`

The names of the fields to include in the resulting database can be specified as a comma separated list of literal field names or character expressions enclosed in parentheses. When this option is omitted, all fields of the primary and secondary work area are included. Use aliased field names to specify fields from the secondary work area.

Description

The JOIN command merges records from the current, or primary, work area with records from a secondary work area into a new database. For each record in the primary work area, all records of the secondary work area are scanned and the FOR condition is evaluated. When this evaluation is .T. (true), a new record is added to the target database.

Records marked for deletion in either work area are ignored when SET DELETED is ON. If this setting is OFF, all records are processed in both work areas. A deletion flag, however, is not transferred to the target database. That is, no record in the target database is marked for deletion.

Caution: The JOIN command can turn out to be extremely time consuming since the FOR condition is evaluated Lastrec() in primary work area times Lastrec() in secondary work area. Use this command with care, especially when merging large databases.

Info

See also: DbJoin(), DbSetRelation(), INDEX, SET RELATION
Category: Database commands

Example

```
// This example joins Customer.dbf with Invoices.dbf to produce Sales.dbf

   PROCEDURE Main
```

```
   USE Invoices  ALIAS Inv NEW
   USE Customer  ALIAS Cust NEW

   JOIN WITH Inv TO Sales ;
         FOR CustNo = Inv->CustNo ;
      FIELDS CustNo, FirstName, LastName, Inv->Total

   CLOSE DATABASE
RETURN
```

KEYBOARD

Writes a string or numeric key codes into the keyboard buffer.

Syntax

```
KEYBOARD <cString>
KEYBOARD <nInkeyCode>
KEYBOARD <aKeyCodes>
```

Arguments

<cString>

This is the character string that is written into the keyboard buffer.

<nInkeyCode>

Alternatively, a numeric key code can be specified. Normally, the #define constants listed in the Inkey.ch files are used for *<nInkeyCode>*.

<aKeyCodes>

A mixture of character strings or numeric key codes can be specified as a one dimensional array.

Description

The KEYBOARD command first clears the keyboard buffer and then fills it with the key codes specified as character strin, numeric values or within an array. Thus, all pending key strokes are discarded before new characters are written into the keyboard buffer. They remain in the buffer until being fetched from it during a wait state in which the keyboard buffer is polled for the next key stroke.

Wait states are employed by functions and commands that wait for user input, such as Achoice(), READ or MemoEdit().

Info

See also:	Chr(), CLEAR TYPEAHEAD, HB_KeyPut(), Inkey(), LastKey(), NextKey(), SET KEY, SET TYPEAHEAD
Category:	Input commands
Header:	Inkey.ch
Source:	rtl\inkey.c
LIB:	xhbdll.lib
DLL:	xhbdll.dll

Examples

```
// The example writes a string into the keyboard buffer so that
// Memoedit() begins editing this text in a new line.

   PROCEDURE MAIN
      LOCAL cString

      KEYBOARD "xHarbour:" + Chr(13) + Chr(10)

      cString := MemoEdit()

      CLS
      ? cString
   RETURN

// The example does the same as the previous one but
// passes an array to the KEYBOARD command.
```

```
PROCEDURE MAIN
   LOCAL cString

   KEYBOARD { "xHarbour:", 13, 10 }

   cString := MemoEdit()

   CLS
   ? cString
RETURN
```

LIST

Lists records of a work area to the console, file or printer.

Syntax

```
       LIST [<Expression,...>] ;
        [FOR  <lForCondition>] ;
      [WHILE  <lWhileCondition>] ;
       [NEXT  <nCount>] ;
     [RECORD  <nRecno>] ;
[TO PRINTER | TO FILE <cFilename>] ;
       [REST] ;
        [ALL] ;
        [OFF]
```

Arguments

`<Expression>`

An optional, comma separated list of expressions to display can be specified. If omitted, all fields of a record are output.

`FOR <lForCondition>`

This is a logical expression which is evaluated for all records in the current work area. Those records where *<lForCondition>* yields .T. (true) are displayed.

`WHILE <lWhileCondition>`

This is a logical expression indicating to continue processing records while the condition is true. The LIST command stops as soon as *<lWhileCondition>* yields .F. (false).

`NEXT <nCount>`

This option defines the number of records to display. It defaults to the current record. The NEXT *<nCount>* scope processes the next *<nCount>* records.

`RECORD <nRecno>`

If specified, only the record with the ID *<nRecno>* is displayed.

`TO PRINTER`

This option directs output to the printer. TO PRINTER and TO FILE are mutually exclusive options.

`TO FILE <cFilename>`

The output can be directed to a file with the name *<cFileName>*.

`REST`

This option lists the records beginning with the current record to the last record. If not specified, output begins with the first record.

`ALL`

The option ALL processes all records. It becomes the default scope when a FOR condition is specified.

`OFF`

If specified, this option suppresses the display of record numbers. By default, record numbers appear in the output.

Description

LIST is a database command displaying the results of a list of expressions to the console, a file or a printer. The command iterates the records in the current work area according to optional FOR and/or WHILE conditions, and the defined scope as specified with the ALL, NEXT, REST or RECORD options.

When records are marked for deletion, an asterisk (*) is output for deleted records when SET DELETED is set to OFF.

Info

See also: ? | ??, DbEval(), DbList()
Category: Database commands, Output commands
Source: rdd\dblist.prg
LIB: xhb.lib
DLL: xhbdll.dll

LOCATE

Scans the current work area for a record matching a condition.

Syntax

```
LOCATE [<Scope>] FOR <lForCondition> ;
    [WHILE <lWhileCondition>] ;
```

Arguments

`<Scope>`

This option defines the number of records to scan. It defaults to ALL. The NEXT <*nCount*> scope scans the next <*nCount*> records, while REST scans records beginning from the current record to the end of file. The search begins with the first record unless a scope is defined.

`FOR <lForCondition>`

This is a logical expression which is evaluated in the current work area while records are scanned. The record scan stops when <*lForCondition*> yields .T. (true).

`WHILE <lWhileCondition>`

This is a logical expression indicating to continue scanning records while the condition is true. The LOCATE operation stops as soon as <*lWhileCondition*> yields .F. (false).

Description

The LOCATE command sequentially scans the records in current work area and stops if a record matching the FOR condition is found. In this case, function Found() returns .T. (true). When no match is found, or when the WHILE condition yields .F. (false), or when the record pointer gets out of scope, Found() return .F. (false), and the scan stops.

If a record matching the FOR condition is found, it becomes the current one. The search can then be continued using the CONTINUE command to find the next matching record. Hence, the FOR condition of the LOCATE command remains active for subsequent CONTINUE commands. Each work area may have its own LOCATE condition, which is persistent until a new LOCATE command is issued or the work area is closed. Note, however, that scope and WHILE condition are not available for a subsequent CONTINUE command.

Both commands, LOCATE and CONTINUE, can be used very flexible for performing complex searches that do not rely on an indexed database. Since the records in the current database are scanned sequentially, a search with LOCATE/CONTINUE can be time consuming. A search on an indexed database with SEEK is much faster but has the drawback of a less flexible search condition.

Info

See also:	CONTINUE, DbSeek(), Eof(), Found(), SEEK, SET FILTER, SET SCOPE
Category:	Database commands
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example uses LOCATE and CONTINUE to list customers
// living in the city of New York.

   PROCEDURE MAIN
      LOCAL cCity := "New York"
      FIELD FirstName, LastName
```

```
   USE Customer NEW

   LOCATE FOR Upper(FIELD->City) = Upper(cCity)

   DO WHILE Found()
      ? LastName, FirstName
      CONTINUE
   ENDDO

   CLOSE Customer
RETURN
```

LOG

Sends a message to open log channels.

Syntax

```
LOG <data,...> ;
[PRIORITY CRITICAL |
             DEBUG |
           DEFAULT |
             ERROR |
              INFO |
           WARNING ]
```

Arguments

`<data,...>`

A comma separated list of arbitrary data to send as message to the open log channels.

`PRIORITY`

Optionally, the priority of this log message can be specified using one of the keywords CRITICAL, DEBUG, DEFAULT, ERROR, INFO or WARNING.

Description

The LOG command accepts a comma separated list of values and creates from it a log message. This log message is sent to all log channels previously opened with the INIT LOG command. If the PRIORITY option is omitted, the priority assigned to each log channel is used. If a PRIORITY keyword is specified, the log message is only received by a log channel when the PRIORITY of this LOG message is equal or higher than the priority of the log channel.

Info

See also: CLOSE LOG, INIT LOG, SET LOG STYLE, TraceLog()
Category: Log commands, xHarbour extensions
Header: hblog.ch
Source: rtl\hblog.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays log messages of all priorities to the console,
// but logs only CRITICAL and ERROR messages to the Testlog.log file.

   #include "Hblog.ch"

   PROCEDURE Main
      INIT LOG Console() ;
               File( HB_LOG_ERROR , "Testlog.log" )

      LOG "CRITICAL" PRIORITY CRITICAL
      LOG "ERROR"    PRIORITY ERROR
      LOG "WARNING"  PRIORITY WARNING
      LOG "INFO"     PRIORITY INFO
      LOG "DEBUG"    PRIORITY DEBUG

      CLOSE LOG
   RETURN
```

MENU TO

Activates a text-mode menu defined with @...PROMPT commands.

Syntax

```
MENU TO <xMenuVar>
```

Arguments

<xMenuVar>

This is the name of a memory variable to receive the result of menu selection. If the variable does not exist, a PRIVATE variable of this name is created.

Description

MENU TO is a compatibility command activating a text-mode menu whose menu items are defined with @...PROMPT commands. When the menu is activated, the user can navigate a light bar with the cursor keys, indicating the currently selected menu item. The light bar is displayed in the enhanced color value of SetColor() while other menu items are displayed in the standard color.

A user selects a menu item by pressing the Return, PgUp or PgDn key. As a result, the MENU TO command terminates user input and assigns the ordinal position of the selected menu item as a numeric value to *<xMenuVar>*.

Menu selection is also terminated when the Escape key is pressed. In this case, no menu item is selected and the value zero is assigned to *<xMenuVar>*.

Info

See also:	@...PROMPT, AChoice(), SET MESSAGE, SET WRAP
Category:	Input commands
Source:	rtl\menuto.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example builds a simple text-mode menu for editing
// a text file or browsing a database.

   PROCEDURE Main
      SET MESSAGE TO MaxRow()
      SET WRAP ON

      DO WHILE .T.
         CLS
         @ 3, 20 SAY    "Select an option        " COLOR "W+/R"

         @ 5, 20 PROMPT "Database - Browse()    " MESSAGE {|| MyMsg(1) }
         @ 7, 20 PROMPT "Text file - MemoEdit()" MESSAGE {|| MyMsg(2) }
         @ 9, 20 PROMPT "QUIT                   " MESSAGE {|| MyMsg(3) }

         MENU TO nPrompt

         SWITCH nPrompt
         CASE 1
           USE Customer
           Browse()
           USE
           EXIT
```

```
      CASE 2
        MemoEdit( MemoRead( "TestMenu.prg" ) )
        EXIT

      DEFAULT
        QUIT
      END
   ENDDO

RETURN


FUNCTION MyMsg( n )
   SWITCH n
   CASE 1 ; RETURN "Browses a database"   ; EXIT
   CASE 2 ; RETURN "Displays a text file" ; EXIT
   CASE 3 ; RETURN "Ends program"         ; EXIT
   END
RETURN ""
```

PACK

Removes records marked for deletion physically from a database file.

Syntax

```
PACK
```

Description

The PACK command removes all records marked for deletion physically from the database file open in the current work area. Disk space occupied by deleted records is recovered and all open indexes are REINDEXed. When the PACK command is complete, the record pointer is positioned on the first logical record.

The database nust be open in EXCLUSIVE mode to run the PACK command. Otherwise, a runtime error is generated.

Info

See also:	DELETE, Deleted(), RECALL, REINDEX, SET DELETED, USE, ZAP
Category:	Database commands
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example removes inactive customers from a database:

   PROCEDURE Main
      USE Customer INDEX Cust01, Cust02 EXCLUSIVE
      ? LastRec()                       // result: 200

      DELETE FOR Year(CONTCTDATE)-Year(Date()) > 5
      PACK

      ? LastRec()                       // result: 194
      CLOSE Customer
   RETURN
```

QUIT

Terminates program execution unconditionally.

Syntax

```
QUIT
```

Description

The QUIT command terminates program execution unconditionally. Before control goes back to the operating system, all files are closed and all EXIT PROCEDURES are executed, unless a runtime error occurs during program shutdown. In this case, the ErrorLevel() function is set to 1 which is the applications's return code to the operating system.

Info

See also: BEGIN SEQUENCE, Break(), ErrorLevel(), EXIT PROCEDURE, RETURN, TRY...CATCH
Category: Memory commands
Source: vm\initexit.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example shows a typical usage of the QUIT command for a
// command line utility that relies on a command line parameter
// to work properly

   PROCEDURE Main( cFileName )

     CLS
     IF Empty( cFileName ) .OR. ! File( cFileName )
        ? "Usage myApp.exe <fileName>"
        QUIT
     ENDIF

     <file processing>

  RETURN
```

READ

Activates editing of @...GET entry fields in text mode.

Syntax

```
READ [SAVE] ;
     [MENU <oTopBarMenu>] ;
   [MSG AT <nRow>, <nLeft>, <nRight>] ;
      [MSG COLOR <cColor>]
```

Arguments

`SAVE`

If SAVE is specified, the *GetList* array holding the Get objects declared with @...GET remains intact when READ is terminated. By default, the *GetList* array is emptied when READ is complete.

`MENU <oTopBarMenu>`

Optionally, a TopBarMenu() object can be specified. In this case, the menu can be activated during READ.

`MSG AT <nRow>, <nLeft>, <nRight>`

This option specifies three numeric values as screen coordinates for the region where the message of the active Get object is displayed. *<nRow>* is the screen row, while *<nLeft>* and *<nRight>* are the column boundaries for message display. A message is defined with the MESSAGE option of the @...GET command.

`MSG COLOR <cColor>`

The parameter *<cColor>* is an optional character string defining the color for the message to display.

Description

The READ command activates full screen editing of all Get objects previously declared with @...GET commands. A user can enter data in the displayed entry fields and navigate between them. Editing starts with the first entry field, unless the WHEN clause of @...GET indicates that editing is not allowed. In this case, the next entry field that is allowed to be edited is activated and receives input focus.

When a user changes to the next entry field, a VALID condition, if present, is evaluated with the current value of the entry field. If this data validation fails, input focus remains with the current entry field, otherwise the next Get object, or entry field, receives input focus.

The READ command recognizes different types of keyboard input for editing entry fields, navigating between them, and terminating data entry:

Keys for navigation

Key	Description
Left	Moves cursor one character to the left
Ctrl+Left	Moves cursor one word to the left
Right	Moves cursor one character to the right
Ctrl+Right	Moves cursor one word to the right
Home	Moves cursor to the first character
End	Moves cursor to the last character
Comma | Period	Moves cursor to behind decimal point (only when numeric values are edited)
Up	Activates previous GET

Down	Activates next GET
Ctrl+Home	Activates the first GET
Return	Activates next GET

Keys for editing

Key	Description
ASCII character	Changes the edit buffer (if PICTURE format permits)
Backspace	Deletes one character left of the cursor
Ctrl+Backspace	Deletes one word left of the cursor
Ctrl+T	Deletes one word right of the cursor
Ctrl+U	Undo changes
Ctrl+Y	Deletes all characters from cursor to the end
Delete	Deletes one character at the cursor position
Insert	Toggles insert mode

Keys for terminating READ

Key	Description
Esc	Terminates READ and discards the currently edited value
Down	Terminates READ when the last GET has input focus and when ReadExit() is .T. (true)
Up	Terminates READ when the first GET has inout focus and when ReadExit() is .T. (true)
Return	Terminates READ when the last GET has input focus
Page Down, Page Up and Ctrl+W	Terminates READ and assigns the currently edited value

Info

See also: @...GET, @...SAY, CLEAR GETS, Get(), LastKey(), ReadModal(), TopBarMenu()
Category: Get system, Input commands
Source: rtl\getsys.prg, rtl\tgetlist.prg
LIB: xhb.lib
DLL: xhbdll.dll

RECALL

Removes a deletion mark from one or more records.

Syntax

```
RECALL [<Scope>] ;
[WHILE <lWhileCondition>] ;
  [FOR <lForCondition>]
```

Arguments

`<scope>`

This option defines the number of records to unmark for deletion. It defaults to the current record. The NEXT <*nCount*> scope removes the deletion mark from the next <*nCount*> records, while the RECORD <*nRecno*> scope processes only one record having the record number <*nRecno*>. The option ALL marks all records for deletion.

The default scope becomes ALL when a FOR condition is specified.

`WHILE <lWhileCondition>`

This is a logical expression indicating to continue unmarking records for deletion while the condition is true. The RECALL command stops as soon as <*lWhileCondition*> yields .F. (false).

`FOR <lForCondition>`

This is a logical expression which is evaluated for all records in the current work area. Those records where <*lForCondition*> yields .T. (true) are unmarked for deletion.

Description

The RECALL command removes the deletion mark for one or more records in the current work area. A record is marked for deletion with the DELETE command.

When a database is open in SHARED mode, the current record must be locked with the RLock() function before RECALL may be called. Failure to do so generates a runtime error. If multiple records should be unmarked for deletion, the entire file must be locked using FLock(), or the file must be open in EXCLUSIVE mode.

Note: The RECALL command can remove a deletion flag only until PACK is executed.

Info

See also:	DbEval(), DbRecall(), DELETE, Deleted(), FLock(), RLock(), PACK, SET DELETED, ZAP
Category:	Database commands
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates the effect of a deletion mark.

   PROCEDURE Main
      USE Customer NEW EXCLUSIVE

      DELETE RECORD 20
      ? Deleted(), Recno()             // result: .T.   20

      RECALL
```

```
   ? Deleted(), Recno()              // result: .F.  20

   CLOSE Customer
RETURN
```

REINDEX

Rebuilds all open indexes in the current work area.

Syntax

```
REINDEX [EVAL  <bBlock>] ;
        [EVERY <nInterval>]
```

Arguments

`EVAL <bBlock>`

This parameter is a code block that must return a logical value. The indexing operation continues as long as EVal(*<bBlock>*) returns .T. (true). The operation is stopped when Eval(*<bBlock>*) returns .F. (false).

The code block is evaluated for each record unless the EVERY option is specified. It is recommened to use EVERY in conjunction with EVAL.

`EVERY <nInterval>`

This option is a numeric value specifying the number of records after which the EVAL block is evaluated. This can improve the indexing operation especially for large databases. EVERY is ignored when no EVAL block is supplied.

Description

The REINDEX command rebuilds all open indexes in the current work area. During the operation the FOR condition, UNIQUE flag and ASCENDING/DESCENDING status defined with the INDEX command are maintained.

REINDEX requires a database be opened in EXCLUSIVE mode, otherwise a runtime error is generated. When the operation is complete, the first index becomes the controlling index and the record pointer is positioned on the first logical record.

Important: REINDEX does not repair an index file header. If an index file gets corrupted, use the INDEX command to re-create index files from scratch.

Info

See also:	INDEX, OrdCondSet(), OrdCreate(), OrdListRebuild(), PACK, SET INDEX, USE
Category:	Database commands, Index commands
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// This example reindexes the indexes open in the current work area:

   PROCEDURE Main
      USE Customer INDEX Cust01, Cust02, Cust03 EXCLUSIVE

      REINDEX

      CLOSE Customer
   RETURN
```

RELEASE

Deletes or resets dynamic memory variables.

Syntax

```
RELEASE <memVarNames,...>
RELEASE ALL [LIKE | EXCEPT <mask>]
```

Arguments

`<memVarNames,...>`

A comma separated list of symbolic names of PRIVATE and/or PUBLIC variables to release.

`ALL [LIKE|EXCEPT <mask>]`

Mask to define the variable names of visible dynamic memory variables to include or exclude from releasing. <*mask*> can be specified using the wildcard characters (*) and (?).

Description

The RELEASE command is used in two different ways. Either by specifying the exact symbolic name(s) of one or more dynamic memory variable(s), or by using a variable name's skeleton together with wildcard characters. The asterisk (*) matches multiple characters while the question mark matches any character at the same position.

As a general rule, the RELEASE command assigns NIL to the specified variables, it does not release their symbolic names immediately. This happens only when the releasing routine returns control to the calling routine.

The option ALL releases all currently visible dynamic variables. When combined either with LIKE (include) or with EXCEPT (exclude), followed by a skeleton specifying a group of variable names using wild card characters, the intended group of variables is selectively released.

GLOBAL, LOCAL and STATIC memory variables are not affected by CLEAR MEMORY.

Info

See also: CLEAR MEMORY, PRIVATE, PUBLIC, SAVE, RESTORE
Category: Memory commands
Source: vm\memvars.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates the effect of releasing PIVATE and
// PUBLIC variables in a main and a sub-routine.

    PROCEDURE Main
      PRIVATE cPrivA := 11 , ;
              cPrivB := 22

      PUBLIC  cPubA  := "Hello", ;
              cPubB  := "World"

      ? cPrivA                  // result: 11
      ? cPubA                   // result: Hello

      TestProc()

      ? cPrivA                  // result: NIL
```

```
   ? cPrivB                  // result: 22

   ? cPubB                   // result: World
   ? cPubA                   // runtime error

RETURN

PROCEDURE TestProc()
   MEMVAR cPrivA, cPrivB

   PRIVATE cPrivA := 1000, ;
           cPrivB := 2000

   ? cPrivA                  // result: 1000
   ? cPrivB                  // result: 2000

   RELEASE cPrivB
   ? cPrivA                  // result: 1000
   ? cPrivB                  // result: NIL

   RELEASE ALL
   ? cPrivA                  // result: NIL
   ? cPrivB                  // result: NIL

   ? cPubA                   // result: Hello

   RELEASE cPubA

   ? Type("cPubA")           // result: U
RETURN
```

RENAME

Changes the name of a file.

Syntax

```
RENAME <cOldFile> TO <cNewFile>
```

Arguments

`<cOldFile>`

This is name of the file to rename. It must include path and file extension and can be specified as a literal file name or as a character expression enclosed in parentheses. The path can be omitted from *<cOldFile>* when the file resides in the current directory.

`TO <cNewFile>`

This is the new file name including file extension. Drive and/or path are optional. *<cNewFile>* can be specified either as a literal string or as a character expression enclosed in parentheses.

Description

The RENAME command changes the name of a file. The file *<cOldFile>* is searched in the current directory only, unless a full qualified file name including drive and path is specified. Directories specified with SET DEFAULT and SET PATH are ignored by RENAME.

If *<cNewFile>* is specified as a full qualified file name and its directory differs from the one of *<cOldFile>*, the source file is moved to the new directory and stored under the new file name.

When the new file either exists or is currently open, RENAME aborts the operation. Use the File() function to test for the existence of *<cNewFile>*.

A file must be closed before attempting to rename it.

Important: Database files with memo fields have accompanying memo files. Memo files must be renamed along with their database files.

Info

See also:	COPY FILE, CurDir(), ERASE, FErase(), FError(), File(), FRename()
Category:	File commands
Source:	rtl\philes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows a user defined function that renames a database file
// along with its memo file.

   PROCEDURE Main

      ? File( "Customer.dbf" )      // result: .T.
      ? File( "Customer.fpt" )      // result: .T.

      ? File( "Cust2005.dbf" )      // result: .F.
      ? File( "Cust2005.fpt" )      // result: .F.

      DbRename( "Customer.dbf", "Cust2005.dbf" )

      ? File( "Customer.dbf" )      // result: .F.
      ? File( "Customer.fpt" )      // result: .F.
```

```
   ? File( "Cust2005.dbf" )      // result: .T.
   ? File( "Cust2005.fpt" )      // result: .T.

RETURN

FUNCTION DbRename( cOldFile, cNewFile )
   LOCAL cOldName := SubStr( cOldFile, 1, Rat( ".", cOldFile ) )
   LOCAL cNewName := SubStr( cNewFile, 1, Rat( ".", cNewFile ) )
   LOCAL cMemoExt, n
   LOCAL aMemoExt := { "DBT", "FPT", "SMT" }

   IF ! File( cOldFile ) .OR. File( cNewFile )
      RETURN .F.
   ENDIF

   FOR n:=1 TO 3
      IF File( cOldName + aMemoExt[n] )
         cMemoExt := aMemoExt[n]
         EXIT
      ENDIF
   NEXT

   RENAME (cOldFile) TO (cNewFile)

   IF ! Empty(cMemoExt) .AND. File( cOldName+cMemoExt )
      RENAME (cOldName+cMemoExt) TO (cNewName+cMemoExt)
   ENDIF
RETURN .T.
```

REPLACE

Assigns values to field variables.

Syntax

```
REPLACE <fieldName1> WITH <expression1> ;
     [, <fieldNameN> WITH <expressionN>] ;
  [NEXT <nCount>|ALL] ;
 [WHILE <lWhileCondition>] ;
   [FOR <lForCondition>]
```

Arguments

`<fieldName>`

This is the symbolic name of the field variable to assign a new value to. The field variable is searched in the current work area unless prefixed with an alias name of another work area.

`WITH <expression>`

The value of *<expression>* is assigned to *<fieldName>*. Note that *<expression>* must yield a value of the same data type as the field variable. Memo fields must be assigned values of data type Character.

`NEXT <nCount>`

This option defines the number of records to process for assignment operations. It defaults to the current record. The NEXT *<nCount>* scope performs assignments to the fields of the next *<nCount>* records.

`ALL`

The option ALL processes all records. It becomes the default scope when a FOR condition is specified.

`WHILE <lWhileCondition>`

This is a logical expression indicating to continue processing records while the condition is true. The REPLACE command stops as soon as *<lWhileCondition>* yields .F. (false).

`FOR <lForCondition>`

This is a logical expression which is evaluated for all records in the current work area. Those records where *<lForCondition>* yields .T. (true) are processed.

Description

The REPLACE command assigns values to one or more field variables. The field variables are identified with their symbolic field names and are searched in the current work area, unless being prefixed with the alias name of a different work area.

The scope of the REPLACE command is the current record when no further condition of scope is specified. otherwise, all records matching scope and condition are processed.

When a database is open in SHARED mode, the current record must be locked with the RLock() function before REPLACE may be called. Failure to do so generates a runtime error. If multiple records should be processed, the entire file must be locked using FLock(), or the file must be open in EXCLUSIVE mode. This applies to files open in all work areas praticipating in the assignment operation.

Note: When field variables that are part of an index expression of open indexes are changed, the corresponding index is automatically updated, unless the index is created with the CUSTOM attribute. In the latter case a custom index must be updated manually.

It is recommended to SET ORDER TO 0 before changing field variables that are part of index expressions. This makes sure that the relative record pointer position does not change automatically while indexes are updated.

Info

See also: COMMIT, DbRlock(), FLock(), RLock()
Category: Database commands

Example

```
// The example demonstrates how to assigne values to a database
// open in SHARED mode.

   PROCEDURE Main
      LOCAL cFirstname, cLastname

      USE Customer INDEX CustA, CustB ALIAS Cust NEW SHARED
      cFirstname := Cust->Firstname
      cLastname  := Cust->Lastname

      CLS
      @ 5,1 SAY "First name:" GET cFirstname
      @ 6,1 SAY "Last name :" GET cLastname
      READ

      IF .NOT. Empty( cFirstname + cLastname ) .AND. RLock()
         REPLACE Firstname WITH cFirstname, ;
                 Lastname  WITH cLastname
         COMMIT
         UNLOCK
      ENDIF

      CLOSE Cust
   RETURN
```

RESTORE

Loads dynamic memory variables from disk into memory.

Syntax

```
RESTORE FROM <memFilename> [ADDITIVE]
```

Arguments

FROM <memFilename>

This is the name of the memory file to open. It can include path and file extension. When no path is given, the file is searched in the current directory. The default file extension is MEM. *<memFilename>* can be specified as a literal file name or as a character expression enclosed in parentheses.

ADDITIVE

This option specifies that the dynamic variables should be loaded in addition to existing ones. If the option is omitted, all dynamic memory variables currently available are released first.

Description

The RESTORE command loads dynamic memory variables previously stored with the SAVE command from a memory file. The default extension for memory files is MEM.

Dynamic memory variables are PRIVATE and PUBLIC variables. Their symbolic names exist at runtime of an application. Thus, the symbolic names are re-created from a memory file with the RESTORE command. If dynamic variables of the same name are visible when the command is executed, they are overwritten with the values stored in the memory file. Variables that do not exist are created new as PRIVATE variables.

When the ADDITIVE clause is not specified, all currently existing dynamic variables are released befor the memory file is loaded.

GLOCBAL, LOCAL and STATIC variables are not affected by the RESTORE command.

Info

See also:	GLOBAL, HB_DeSerialize(), LOCAL, PRIVATE, PUBLIC, RESTORE, STATIC
Category:	Memory commands
Source:	vm\memvars.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates saving and restoring of dynamic memory
// variables.

   PROCEDURE Main
      PRIVATE cString1, cString2

      PUBLIC dDate   := Date()
      PUBLIC lLogic  := .F.


      SAVE TO AllVars

      SAVE ALL LIKE c* TO StringVar

      SAVE ALL EXCEPT cString? TO SomeVars
```

```
   RELEASE ALL
   CLEAR MEMORY

   RESTORE FROM AllVars
   RESTORE FROM StringVar
   RESTORE FROM SomeVars
RETURN
```

RUN

Executes an operating system command.

Syntax

```
RUN <CommandLine>
```

Arguments

<CommandLine>

This is the command to be executed on the operating system level. It can be specified as literal or as character string in parentheses.

Description

The RUN command opens a new command shell and executes the operating system commands specified with <*CommandLine*>.

RUN does not return until <*CommandLine*> is completed by the operating system. When RUN is complete, the new command shell is closed.

Info

See also:	( ), FUNCTION, PROCEDURE
Category:	Statements
Source:	vm\run.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example executes the DIR command, directs its output into
// a file and displays the result with Notepad.exe

   PROCEDURE Main
      LOCAL cCommand := "Notepad.exe files.lst"

      CLS

      ? "DIR command"
      RUN dir > files.lst

      ? "Executing Notepad.exe"
      RUN (cCommand)

      ? "Done"
   RETURN
```

SAVE

Saves dynamic memory variables to a memory file.

Syntax

```
SAVE TO <memFilename> [ALL [LIKE | EXCEPT <mask>]]
```

Arguments

`FROM <memFilename>`

This is the name of the memory file to create. It can include path and file extension. When no path is given, the file is created in the current directory. The default file extension is MEM. *<memFilename>* can be specified as a literal file name or as a character expression enclosed in parentheses.

`ALL [LIKE|EXCEPT <mask>]`

Mask to define the variable names of visible dynamic memory variables to include or exclude from saving. *<mask>* can be specified using the wildcard characters (*) and (?).

Description

The SAVE command writes dynamic memory variables currently visible to a memory file. The default extension for memory files is MEM. Only variables of data type Character, Date, Logic and Numeric can be written to a MEM file. Other data types are not supported. The vaiables can later be loaded into memory using the RESTORE command-

Dynamic memory variables are PRIVATE and PUBLIC variables. Their symbolic names exist at runtime of an application. Thus, the symbolic names are written into a memory file along with their values.

The option ALL stores all currently visible dynamic variables. When combined either with LIKE (include) or with EXCEPT (exclude), followed by a skeleton specifying a group of variable names using wild card characters, the intended group of variables is selectively stored.

GLOBAL, LOCAL and STATIC memory variables cannot be written to a memory file with the SAVE command.

Info

See also:	GLOBAL, HB_Serialize(), LOCAL, PRIVATE, PUBLIC, RESTORE, STATIC
Category:	Memory commands
Source:	vm\memvars.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// Refer to the RETSORE example
```

SEEK

Searches a value in an index.

Syntax

```
SEEK <expression> [LAST] [SOFTSEEK]
```

Arguments

`<expression>`

This is an arbitrary expression whose value is searched in the controlling index. The data type of the value must match with the data type of the index expression.

`LAST`

This option determines the record to begin the search with if there are multiple records having the same index value. By default, the search starts with the first of multiple records.

`SOFTSEEK`

This option controls the position of the record pointer when the searched value is not found. If omitted, the record pointer is positioned on the ghost record (Lastrec()+1). When SOFTSEEK is specified and the value is not found, the record pointer is positioned on the record with the next higher index value.

Description

The SEEK command searches a value in the controlling index. When the value is found, the record pointer is positioned on this record and function Found() returns .T. (true). When the value is not found in the index, the record pointer is positioned on Lastrec()+1 (without SOFTSEEK) or on the record having the next higher index value (with SOFTSEEK). In either case, function Found() returns .F. (false). If SOFTSEEK is used and no higher index value is found, the record pointer is positioned on Lastrec()+1, and function Eof() returns .T. (true).

The LAST option is useful if the searched value is found in multiple records. By default, the record pointer is positioned on the first of multiple records matching the search value. With the LAST option specified, the record pointer is positioned on the last of multiple records matching the search value.

Note: When the index expression yields a value of data type Character, the search value can be a character string that contains one character up to the length of the index value. When the searched value is shorter than the index value, the search includes only Len(<*expression*>) characters.

Info

See also:	DbSeek(), Eof(), Found(), LOCATE, SET SCOPE, SET SOFTSEEK, Set()
Category:	Database commands, Index commands
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates different search conditions and values based
// on the names "Jane Doe" and "John Doe"

   PROCEDURE Main
      USE Test
      INEX ON Upper(Lastname+FirstName) TO Test

      ? Len( FIELD->LASTNAME)          // result: 15
```

```
   SEEK "A"
   ? Found(), Eof(),Trim(Lastname)  // result: .F., .T., ""

   SEEK "A" SOFTSEEK
   ? Found(), Eof(),Trim(Lastname)  // result: .F., .F., "Doe"

   SEEK "DOE"
   ? Found(), Eof(),Trim(Firstname) // result: .T., .F., "Jane"

   SEEK "DOE" LAST
   ? Found(), Eof(),Trim(Firstname) // result: .T., .F., "John"
RETURN
```

SELECT

Changes the current work area.

Syntax

```
SELECT <nWorkArea> | <cAliasName>
```

Arguments

`<nWorkArea>`

This is a numeric value between 0 and 65535. It specifies the number of the work area to become the current one. All database commands and unaliased functions are executed in the current work area.

The value 0 has a special meaning: it selects the next unused work area disregarding its work area number.

`<cAliasName>`

This is the symbolic alias name of a used work area. The alias name is specified with the USE command.

Note: both work area number or alias name can be specified as literal value or as expression enclosed in parentheses.

Description

The SELECT command is used to select the current work area. A work area is a logical entity where database files are open along with their accompanying files, like memo or index files. The maximum number of work areas is restricted to 65535.

Work areas can be selected using their numeric identifier. When a work area is occupied by an open database, it can be selected using the alias name specified with the USE command. The alias name is a powerful means for executing expressions in a work area that is not the current one. All that is required is to enclose an expression in parentheses and prefix it with the alias name and alias operator.

Info

See also:	Alias(), DbSelectArea(), Select(), SET INDEX, USE, Used()
Category:	Database commands
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates how different operatations can be coded
// for multiple work areas.

   PROCEDURE Main
      SELECT 1
      USE Customer ALIAS Cust
      ? Select()                       // result: 1

      USE Invoice ALIAS Inv NEW
      ? Select()                       // result: 2

      USE Sales NEW
      ? Select()                       // result: 3

      SELECT Cust
```

```
    ? LastRec()                        // result: 200
    ? Inv->(LastRec())                 // result: 369

    CLOSE ALL
RETURN
```

SET ALTERNATE

Records the console output in a text file.

Syntax

```
SET ALTERNATE TO [<fileName> [ADDITIVE]]

or

SET ALTERNATE on | OFF | ( <lOnOff> )
```

Arguments

`TO <fileName>`

This is the name of a standard ASCII text file that receives console output. The default extension for <*fileName*> is .TXT. The file name can be specified as a string literal or a character expression enclosed in parentheses. When the file name is specified without drive letter and path, the file is created in the current directory. An existing file with the same name is overwritten without warning.

`ADDITIVE`

The ADDITIVE option causes console output be appended to an existing file. Without this option, a new and empty file is created before output is written to the file.

`ON | OFF | ( <lOnOff> )`

This option toggles if output is written to a file. With ON or .T. (true). console output is written to the file. OFF or .F. (false) switch this mode off.

Description

The SET ALTERNATE command writes console output to an ASCII text file with the default extension .TXT. The file is created as a new file unless the ADDITIVE option is used. Output to the file starts when SET ALTERNATE ON is executed and it ends with SET ALTERNATE OFF. The latter does not close the file but stops recording output. The file can be closed with CLOSE ALTERNATE, CLOSE ALL or SET ALTERNATE TO with no argument.

Some commands which display output to the console view have a TO FILE clause. It has the same result as the SET ALTERNATE command. A full-screen command such as @...SAY can not be echoed to a file. Instead, use SET PRINTER TO <*xcFile*> with SET DEVICE TO PRINTER to enable this.

Note: alternate files are not related to work areas. Therefore, only one file can be open with SET ALTERNATE at any time.

Info

See also:	CLOSE, FCreate(), FOpen(), FWrite(), Set(), SET CONSOLE, SET PRINTER
Category:	Console commands
Source:	rtl\console.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// This example creates an alternate file and writes the results
// of the ? command to the file:

   PROCEDURE Main
      SET ALTERNATE TO MyFile
```

```
    SET ALTERNATE ON
    USE Employees NEW

    DO WHILE !EOF()
       ? Employees->Name
       SKIP
    ENDDO

    SET ALTERNATE OFF
    CLOSE ALTERNATE
    CLOSE Employees
 RETURN
```

SET AUTOPEN

Toggles automatic opening of a structural index file.

Syntax

```
SET AUTOPEN ON | off | (<lOnOff>)
```

Arguments

`ON | off | (<lOnOff>)`

The option toggles if a structural index file is automatically opened with the USE command. With ON or .T. (true), an index file is automatically opened. OFF or .F. (false) switch this mode off.

Description

Some replaceable database drivers support automatic opening of index files with the USE command when the index file has the same file name as the database file (without extension). An example is the DBFCDX driver. SET AUTOPEN toggles this behavior.

When SET AUTOPEN is set to ON, which is the default, the USE command automatically opens an index file having the same name as the database file and the file extension returned from OrdBagExt().

Note: if an index file is automatically opened, a controlling index is not activated. The default index order is zero, i.e. records are accessible in physical order in the work area. To select a controlling index, call OrdSetFocus() or use SET AUTORDER for a default controlling index.

Info

See also:	OrdListAdd(), OrdSetFocus(), Set(), SET AUTORDER, SET AUTOSHARE, USE
Category:	Database commands, SET commands, xHarbour extensions
Source:	rtl\set.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates the effect of SET AUTOPEN with the
// DBFCDX driver.

   REQUEST DBFCDX

   PROCEDURE Main

      RddSetDefault( "DBFCDX" )
      SET AUTOPEN OFF

      USE Customer
      INDEX ON CustID                    TAG ID   TO Customer
      INDEX ON Upper(LastName+FirstName) TAG Name TO Customer

      USE Customer
      ? OrdCount(), OrdKey()        // result: 0  ""

      SET AUTOPEN ON

      USE Customer
      ? OrdCount(), OrdKey()        // result: 2  ""

      SET AUTORDER TO 1
```

```
    USE Customer
    ? OrdCount(), OrdKey()          // result: 2  CUSTID
    ? Ordkey( 2 )                   // result: Upper(LastName+FirstName)

    USE
RETURN
```

SET AUTORDER

Defines the default controlling index for automatically opened index files.

Syntax

```
SET AUTORDER TO <nOrder>
```

Arguments

<nOrder>

This is a numeric value specifying the ordinal position of the index to select as the controlling index when SET AUTOPEN is set to ON. The default is zero.

Description

When SET AUTOPEN is set to ON and the RDD supports automatic opening of structural indexes, the SET AUTORDER command specifies the index to activate as the controlling index. The default value for *<nOrder>* is zero, i.e. no controlling index is activated when an index file is automatically opened with the USE command. In this case, records of the database are accessible in physical order in the work area.

Note: refer to SET AUTOPEN for an example of automatic selection of the controlling index.

Info

See also:	OrdSetFocus(), Set(), SET AUTOPEN, SET AUTOSHARE, USE
Category:	Database commands, SET commands, xHarbour extensions
Source:	rtl\set.c
LIB:	xhb.lib
DLL:	xhbdll.dll

SET AUTOSHARE

Defines network detection for shared file access.

Syntax

```
SET AUTOSHARE TO [<nMode>]
```

Arguments

<nMode>

A numeric value 0, 1 or 2 can be specified for *<nMode>*. The default is 0. If omitted, the network detection mode is switched off.

Description

SET AUTOSHARE is a compatibility command useful for changing a multi-user application to a stand-alone application by changing one line of code in the start routine of a program. This requires changing only the value of *<nMode>*:

Values for SHARE mode detection

Value	Description
0 *)	Disables SHARE mode detection
1	Opens database SHARED in a network, and EXCLUSIVE if no network is detected
2	Always opens databases EXCLUSIVE

**) default*

To take advantage of SET AUTOSHARE, an application must be programmed for multi-user access, respecting the rules for network programming. For example, record locks must be obtained with RLock() before changing field variables. This way, a multi-user application is created. To change this application to a single user version, SET AUTOSHARE TO 2 is coded in the start routine.

A developer can SET AUTOMODE TO 1 on the development machine. In this case, performance advantages from EXCLUSIVE file access are available during the development cycle, while SHARED file access is granted in a multi-user environment.

Info

See also: Set(), SET AUTOPEN, SET AUTORDER, USE
Category: Database commands, SET commands, xHarbour extensions
LIB: xhb.lib
DLL: xhbdll.dll

SET BACKGROUND TASKS

Enables or disables the activity of background tasks.

Syntax

```
SET BACKGROUND TASKS on | OFF | (<lOnOff>)
```

Arguments

`on | OFF | (<lOnOff>)`

The option toggles the activity of background tasks defined with function HB_BackGroundAdd(). With ON or .T. (true), background task processing is enabled. OFF or .F. (false), which is the default, disable the activity of background tasks.

Description

SET BACKGROUND TASKS toggles the activity of background task processing. Backgound tasks are defined with function HB_BackGroundAdd() and run concurrently to regular program routines. Use function HB_IdleAdd() to process concurrent tasks during idle states only.

To enable background task processing, SET BACKGROUND TASKS must be set to ON.

Info

See also:	HB_BackGroundAdd(), Set(), SET BACKGROUNDTICK
Category:	Background processing, SET commands, xHarbour extensions
Source:	rtl\set.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates how the time can be displayed
// continuously while MemoEdit() is active.

   PROCEDURE Main
      LOCAL nTask

      HB_IdleAdd( {|| HB_BackGroundRun() } )

      nTask := HB_BackGroundAdd( {|| ShowTime( MaxRow(), MaxCol()-7 ) }, 1000 )

      SET BACKGROUND TASKS ON

      MemoEdit( MemoRead( "Test.prg" ), 1, 0, MaxRow()-2, MaxCol() )

      HB_BackGroundDel( nTask )
   RETURN


   PROCEDURE ShowTime( nRow, nCol )
      DispoutAt( nRow, nCol, Time() )
   RETURN
```

SET BACKGROUNDTICK

Defines the processing interval for background tasks.

Syntax

```
SET BACKGROUNDTICK <nInterval>
```

Arguments

`<nInterval>`

This is a numeric value specifying the number of instructions to be processed by the xHarbour virtual machine before a new cycle of background task processing is started. The default value is 1000.

Description

SET BACKGROUNDTICK can be used to "fine tune" background task processing. The default value of 1000 is usually adequate for a good balance between regular and background task processing.

Background tasks run concurrently with the main program. When *<nInterval>* is enlarged, the main program gets more CPU time since less time is required for background task checking. Reducing *<nInterval>* improves the response time for background tasks.

Info

See also:	HB_BackGroundAdd(), Set(), SET BACKGROUND TASKS
Category:	Background processing, SET commands, xHarbour extensions
Source:	rtl\set.c
LIB:	xhb.lib
DLL:	xhbdll.dll

SET BELL

Toggles automatic sounding of the bell in the GET system.

Syntax

```
SET BELL on | OFF | (<lOnOff>)
```

Arguments

`on | OFF | (<lOnOff>)`

The option toggles the bell during user input in the GET system. The default setting is OFF.

Description

SET BELL is a compatibility command that toggles the automatic sounding of the bell during user input with the GET system. When set to ON, the bell sounds if the last position in a GET entry field is reached, or when invalid data is entered.

Info

See also:	@...GET, Chr(), SET CONFIRM, SET SCOREBOARD, Set(), Tone()
Category:	Environment commands, SET commands
Source:	rtl\set.c
LIB:	xhb.lib
DLL:	xhbdll.dll

SET CENTURY

Sets the date format to include two or four digits.

Syntax

```
SET CENTURY on | OFF | ( <lOnOff> )
```

Arguments

```
on | OFF | ( <lOnOff> )
```

ON activates the display of century digits, and OFF deactivates them. Alternatively, a logical expression <*lOnOff*> can be specified in parentheses. .T. (true) represents ON, while .F. (false) stands for OFF.

Description

The SET CENTURY command enables or disables the display of century digits. By using the four digit notation for a year, dates of any century can be input. While SET CENTURY ON displays the four digit notation, SET CENTURY OFF will hide the century digits and will only display the year without century.

The century information is not removed from date values. Only the display and input of date values is affected. Dates from 01/01/0100 to 12/31/2999 are supported.

Info

See also: CtoD(), CSetCent(), Date(), DtoC(), DtoS(), SET DATE, SET EPOCH, Set(), StoD()
Category: SET commands
Source: rtl\set.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// This example shows the result of the SET CENTURY command:

   PROCEDURE Main
      SET CENTURY OFF
      ? Date()                  // Result: 12/06/05

      SET CENTURY ON
      ? Date()                  // Result: 12/06/2005
   RETURN
```

SET COLOR

Defines default colors for text-mode applications.

Syntax

```
SET COLO[U]R TO ;
      [ [ <standard>] ;
        [,<enhanced>] ;
        [,<border>] ;
        [,<background>] ;
        [,<unselected>] ;
      ]

or:

SET COLO[U]R TO ( <cColor> )
```

Arguments

`TO <standard>[, ...]`

A comma separated list of up to five color values can be specified that define the default colors for screen display in text-mode applications

`TO ( <cColor> )`

Alternatively, a color string enclosed in parentheses can be used to define the default colors.

Description

SET COLOR is a compatibility command used to define screen colors in text-mode applications. The command is entirely superseeded by function SetColor(). Refer to this function for an explanation how color values are coded.

Info

See also:	@...GET, @...SAY, Set(), SetColor()
Category:	Output commands, SET commands
LIB:	xhb.lib
DLL:	xhbdll.dll

SET CONFIRM

Determines how a GET entry field is exited.

Syntax

```
SET CONFIRM on | OFF | (<lOnOff>)
```

Arguments

`on | OFF | (<lOnOff>)`

The option determines if an exit key must be pressed to end editing in a GET entry field. With ON or .T. (true), the Return key must be pressed to leave a GET. OFF or .F. (false), which is the default, changes focus to the next GET once the user types past the last position in the edit buffer.

Description

SET CONFIRM is a compatibility command for the GET system in text-mode applications. When SET CONFIRM is OFF, the cursor moves to the next GET if the user types past the last position in the edit buffer of the current GET. With SET CONFIRM ON, the user is required to type an exit key (E.g. the Return key) to advance the cursor to the next GET.

Info

See also: @...GET, READ, ReadModal(), SET BELL, SET SCOREBOARD, Set()
Category: Input commands, SET commands
LIB: xhb.lib
DLL: xhbdll.dll

SET CONSOLE

Sets if console display is shown on the screen.

Syntax

```
SET CONSOLE ON | off | ( <lOnOff> )
```

Arguments

`ON | off | ( <lOnOff> )`

Alternatively to the options ON or OFF, a logical expression *<lOnOff>* in parentheses can be specified. .T. (true) represents ON, while .F. (false) stands for OFF.

Description

The SET CONSOLE command determines whether or not output from console commands is sent to the screen. The console commands display output on the screen without referencing row and/or column position of the cursor. Console commands can simultaneously send output to a printer and/or ASCII text file. Output can be directed to a printer by using the TO PRINTER clause, if it is supported by console commands, or the SET PRINTER ON command. Echoing the output to a file is achieved by the TO FILE clause or the SET ALTERNATE or SET PRINTER TO commands.

When SET CONSOLE is OFF, no output is displayed on the screen. Output to a file or printer however, is not affected by this setting.

Note: all commands that use explicit row and column positions for output are not affected by the SET CONSOLE setting. Their output is controlled with the SET DEVICE command which selects the output device.

Info

See also:	Set(), SET ALTERNATE, SET DEVICE, SET PRINTER
Category:	Console commands
Source:	rtl\set.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
This example uses REPORT FORM to output records to the printer while no
ouput is send to the screen:

  USE Employees NEW
  SET CONSOLE OFF
  REPORT FORM Name TO PRINTER

  SET CONSOLE ON
```

SET CURSOR

Toggles the display of the screen cursor in text-mode applications

Syntax

```
SET CURSOR on | OFF | (<lOnOff>)
```

Arguments

`on | OFF | (<lOnOff>)`

The option determines if the screen cursor is visible or not. With ON or .T. (true), the cursur is visible. OFF or .F. (false), hides the screen cursor.

Description

SET CURSOR is used to display or hide the screen cursor in text-mode applications. Normally, the screen cursor must only be visible when a user enters data. The command is superseeded by the SetCursor() function which cannot only show/hide the cursor, but also define the cursor's shape.

Info

See also: CSetCurs(), SET CONSOLE, Set(), SetCursor(), SetPos()
Category: Output commands, SET commands
Source: rtl\set.c
LIB: xhb.lib
DLL: xhbdll.dll

SET DATE

Specifies the date format for input and display.

Syntax

```
SET DATE FORMAT [TO] <cDateFormat>

or

SET DATE [TO] AMERICAN |
                  ansi |
               British |
                French |
                German |
               Italian |
                 Japan |
                   usa
```

Arguments

TO <cDateFormat>

<cDateFormat> is a character expression specifying which type of date format will be used. The value of *<cDateFormat>* must be a string of up to 12 characters.

If specified, *<cDateFormat>* determines the format (placement and number of digits) of the day, month and year. Position of day, month and year digits is defined by the number of occurrences of the letters d, m and y. Any other characters are displayed in the date values.

If FORMAT is omitted, one of the following keywords can be used to specify the desired date format.

Keywords for SET DATE

Keyword	Date format
AMERICAN	mm/dd/yy
ANSI	yy.mm.dd
BRITISH	dd/mm/yy
FRENCH	dd/mm/yy
GERMAN	dd.mm.yy
ITALIAN	dd-mm-yy
JAPAN	yy/mm/dd
USA	mm-dd-yy

Description

The SET DATE command specifies the format for the display and input of date values. It is a global setting valid for an entire xHarbour application.

Info

See also: CtoD(), Date(), DtoC(), DtoS(), Set(), SET CENTURY, SET EPOCH
Category: SET commands
Source: rtl\set.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// In this example the FORMAT clause directly specifies the date format

   PROCEDURE Main
      ? Set( _SET_DATEFORMAT )          // result: mm/dd/yy
      ? Date()                          // result: 12/21/05

      SET DATE FORMAT TO "yy:mm:dd"

      ? Date()                          // result: 05:12:21
   RETURN
```

SET DBFLOCKSCHEME

Selects the locking scheme for shared database access.

Syntax

```
SET DBFLOCKSCHEME TO <nLockScheme>
```

Arguments

TO <nLockScheme>

This is a numeric value defining the locking scheme for record locks in shared database access. Values listed in the following table or #define constants from the DBINFO.CH file can be used for *<nLockScheme>*

Constants for SET DBFLOCKSCHEME

Constant	Value	Description
DB_DBFLOCK_DEFAULT	0	Default locking scheme
DB_DBFLOCK_CLIP	1	Clipper 5.2 locking scheme
DB_DBFLOCK_CL53	2	Clipper 5.3 locking scheme
DB_DBFLOCK_VFP	3	Visual FoxPro locking scheme
DB_DBFLOCK_CL53EXT	4	Emulated shared locking
DB_DBFLOCK_XHB64	5	Locking scheme for files > 4GB

Description

This setting defines the locking scheme used for record locks with shared database access. The default locking scheme is DBFLOCK_DEFAULT.

The locking scheme used as the default depends on the RDD used for opening a database. When the DBFCDX RDD is used, the default locking scheme is DBFLOCK_VFP. For DBF, DBFFPT, DBFDBT and DBFNTX the DBFLOCK_CLIP locking scheme is used as the default.

Note: In the NTX header file, there is a flag which informs that DBFLOCK_CL53 should be used.

The DBFLOCKSCHEME command needs to be set before opening a database file. Different locking schemes can be set for each work area, but one file should never be accessed with different locking schemes at the same time. Always use UNLOCK to release all locks before changing the locking scheme for a database file.

Setting the locking scheme to 1 will lock the database files like CA-Clipper 5.2 does. To use CA-Clipper 5.3's locking scheme, set DBFLOCKSCHEME to 2. This will emulate shared locks using exclusive locks. Visual FoxPro's locking scheme is selected with DBFLOCKSCHEME set to 3.

When using locking scheme 4, a shared locking will be emulated. This scheme is very useful with systems that do not support shared locks. Although there are no problems with xHarbour, be cautious when using this scheme with a Clipper application in a network environment. Note that the file size is up to 4GB which makes this locking scheme the finest for use with a FAT32 file system. It can also be used for DBFNTX and DBFCDX.

When using files larger than 4GB, use scheme 5. This locking scheme requires large file support by the operating system. Note that it does not reduce the size of the file. This scheme was tested successfully on aLinux systems.

Note: The DBFLOCK_CL53 locking scheme uses the same locking scheme as the locking scheme that COMIX for CA-Clipper uses.

On POSIX (Linux and other *nixes) platforms SHARED locks are used when possible in all locking schemes. So it's not necessary to set DBFLOCK_CL53 which cannot work when 32bit file IO is used (maximum lock offset has 31bit).

In DOS/Windows the DBFLOCK_CL53 will be probably the most efficient for multi-user applications.

The locking scheme selected for an open database file can be checked with function DbInfo(DBI_LOCKSCHEME).

Info

See also: Set(), DbInfo(), DbRLock(), RddSetDefault(), RLock()
Category: Database commands, SET commands, xHarbour extensions
Header: DbInfo.ch
Source: rdd\dbcmd.c, rtl\set.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
#include "DbInfo.ch"

REQUEST DBFCDX

PROCEDURE Main()
   LOCAL aStruct := { { "FIELD1", "C", 30, 0 }, ;
                      { "FIELD2", "N", 10, 2 } }

   RddSetDefault( "DBFCDX" )

   SET DBFLOCKSCHEME TO DB_DBFLOCK_CL53
   DbCreate( "test", aStruct, "DBFCDX")

   USE Test SHARED
   APPEND BLANK

   IF ! NetErr()
      FIELD1->"This is a test"
      FIELD2->100
      ? "Append operation completed"
      COMMIT
   ELSE
      ? "Append operation failed"
   ENDIF
RETURN
```

SET DECIMALS

Defines the number of decimal places for displaying numeric values on the screen.

Syntax

```
SET DECIMALS TO [<nDecimals>]
```

Arguments

<nDecimals>

This is a numeric value specifying the number of decimal places for screen output of numbers. The default value is 2. If *<nDecimals>* is omitted, the number of decimal places is set to zero.

Description

SET DECIMALS defines the number of decimal places for the display of numbers in text-mode applications. Note that the command affects only the display and not the accuracy of calculations with numeric values. If a number has more decimal places than *<nDecimal>*, the number is rounded for display.

Note: to activate a fixed number of decimal places for screen display, SET FIXED must be set to ON.

Info

See also:	@...GET, @...SAY, SET FIXED, Set(), Str(), Transform()
Category:	Output commands, SET commands
Source:	rtl\set.c
LIB:	xhb.lib
DLL:	xhbdll.dll

SET DEFAULT

Sets the default drive and directory.

Syntax

```
SET DEFAULT TO [<cPath>]
```

Arguments

`TO <cPath>`

The *<cPath>* parameter specifies the default drive and directory for an xHarbour application to create and/or open files. Directories are separated by backslashes and the drive letter is separated with a colon. *<cPath>* can be specified as a literal path or as a character expression enclosed in parentheses.

If *<cPath>* is omitted, the current drive and directory of the operating system is used as default directory

Description

The SET DEFAULT command specifies the drive and directory where the application creates and saves files. The initial value of *<cPath>* is the directory from where the xHarbour application is started.

Use the SET PATH command to add additional directories to be searched for files.

Note: The SET DEFAULT command does not change the default directory of the operating system.

Info

See also:	CurDir(), DefPath(), File(), Set(), SET PATH
Category:	SET commands
Source:	rtl\set.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates the effect of changing the default
// drectory of an xHabour application.

   PROCEDURE Main
      SET PATH TO
      ? File( "Customer.dbf" )         // Result: .F.

      SET DEFAULT TO C:\xhb\data
      ? File( "Customer.dbf" )         // Result: .T.

      SET DEFAULT TO C:                // define default drive
      SET DEFAULT TO ..                // Change to parent directory
      SET DEFAULT TO \                 // Change to root directory

   RETURN
```

SET DELETED

Specifies visibility of records marked for deletion.

Syntax

```
SET DELETED on | OFF | ( <lOnOff> )
```

Arguments

`on | OFF | ( <lOnOff> )`

This option toggles whether records marked for deletion are visible or not. The default is OFF or .F. (false), i.e. all records are visible. To change the setting use ON or .T. (true) as parameter. The parameter can also be specified as a logical expression enclosed in parentheses.

Description

The SET DELETED setting controls the visibility of records marked for deletion. A deletion mark is set with the DELETE command and can be removed with RECALL. SET DELETED is a global setting valid for all work areas. To suppress visibility of deleted records in a single work area, SET DELETED to OFF and use a filter expression like SET FILTER TO Deleted().

Visibility of deleted records is suppressed while navigating the record pointer with SKIP, GO TOP or GO BOTTOM. During such relative navigation, all records carrying the deleted flag are ignored. Absolute navigation with GOTO <*nRecordNumber*>, however, allows for positioning the record pointer to a deleted record even when SET DELETED is ON.

Note: The SET DELETED setting has no effect on index creation with INDEX or REINDEX.

Info

See also:	DbDelete(), DbRecall(), DELETE, Deleted(), RECALL, Set(), SET FILTER
Category:	Database commands, SET commands
Source:	rtl\set.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// This example illustrates the effect of SET DELETED

   PROCEDURE Main
      USE Customer EXCLUSIVE
      DELETE NEXT 3                 // deletes record 1-3
      GOTO 1                        // go to record #1

      SET DELETED ON                // Ignore deleted records
      SKIP 3                        // skip 3 records
                                    // (first three are ignored)
      ? Recno(), Deleted()          // result: 6 .F.
      RECALL ALL                    // has no effect since
                                    // DELETED is ON
      GO TOP                        // first logical record
      ? Recno(), Deleted()          // result: 4 .F.

      GOTO 1                        // absolute navigation
      ? Recno(), Deleted()          // result: 1 .T.

      SET DELETED OFF
      RECALL ALL                    // removes all deletion flags
```

```
   GO TOP                              // first logical record
   ? Recno(), Deleted()                // result: 1 .F.

   CLOSE Customer
RETURN
```

SET DELIMITERS

Defines delimiting characters for GET entry fields and their visibility.

Syntax

```
SET DELIMITERS on | OFF | (<lOnOff>)

or

SET DELIMITERS TO [<cDelimiters> | DEFAULT]
```

Arguments

`on | OFF | (<lOnOff>)`

The option determines if GET entry fields are displayed within delimiting characters or not. With ON or .T. (true), the delimiters are displayed. OFF or .F. (false), hide the delimiters.

`TO <cDelimiters>`

A two character string can be specified for <*cDelimiters*>. The first character is used as left delimiter and the second character defines the right delimiter for GET entry fields. The DEFAULT option resets the delimiting characters to their default value which is two colons (::).

Description

SET DELIMITERS defines delimiting characters diplayed to the left and right of GET entry fields. The command exists forcompatibility reasons and is only used in text-mode applications.

Info

See also:	@...GET, SET INTENSITY, Set()
Category:	Output commands, SET commands
Source:	rtl\set.c
LIB:	xhb.lib
DLL:	xhbdll.dll

SET DESCENDING

Changes the descending flag of the controlling index at runtime.

Syntax

```
SET DESCENDING on | OFF | (<lOnOff>)
```

Arguments

```
on | OFF | (<lOnOff>)
```

This option toggles the descending flag of the controlling order in the current work area. The initial value of this setting is identical with the vlaue of the descending flag upon index creation. The value ON or .T. (true) enables the descending order, the value OFF or .F. (false) disables it. Note that when specifiying *<lOnOff>* as a logical expression, it must be enclosed in parentheses.

Description

The SET DESCENDING command is an equivalent to function OrdDescend(). It allows to change, or reverse, the ascending/descending order for navigating the record pointer of an indexed database.

Note that SET DESCENDING is only effective during runtime of an application. It does not change the DESCENDING flag in the file header of an index file. This flag can only be set with the INDEX command upon index creation.

Info

See also: INDEX, OrdCondSet(), OrdDescend()
Category: Index commands, SET commands
Source: rdd\dbcmd.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates the effect of SET DESCENDING at runtime

   PROCEDURE Main
      USE Customer                       // create an ascending index
      INDEX ON Upper(LastName) TO Cust01

      GO TOP                             // go to first logical record
      ? Recno(), Lastname                // result:  28 Abbey

      SET DESCENDING ON                  // change to descending order

      GO TOP                             // go to first logical record
      ? Recno(), Lastname                // result: 131 Zwillick

      CLOSE Customer
   RETURN
```

SET DEVICE

Selects the ouput device for @...SAY commands.

Syntax

```
SET DEVICE TO SCREEN | printer
```

Arguments

`TO SCREEN`

This outputs all @...SAY commands to the screen. The output is not influenced by the SET PRINTER and SET CONSOLE settings.

`TO PRINTER`

Outputs all @...SAY commands to the device set with SET PRINTER TO. Possible devices are local printer, network printer or a file.

Description

The SET DEVICE command sends the output of @...SAY commands either to the screen, or a printer. When SET DEVICE TO PRINTER is used, no output is displayed on the screen. The @...SAY command respects the settings set by SET MARGIN.

When the output is send to the printer, an automated EJECT is performed when the current printhead row position is less than the last print row position. When such an EJECT is executed, PCol() and PRow() values are set to zero. To The SetPRC() function can then be used to initialize PCol() and PRow() with new values.

To direct the output of @...SAY to a file, use SET PRINTER TO <*fileName*> and SET DEVICE TO PRINTER.

Info

See also:	@...SAY, EJECT, PCol(), PRow(), Set(), SET PRINTER, SetPrc()
Category:	Output commands
Source:	rtl\set.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates how to direct the output of @...SAYs
// to the printer and to a file:

   PROCEDURE Main
      SET DEVICE TO PRINTER
      @ 2,10 SAY "Hello there"
      EJECT

      SET PRINTER TO MyOutputFile.txt
      SET DEVICE TO PRINTER
      @ 10, 10 SAY "Current file is MyOutputFile.txt"

      SET PRINTER TO
      SET DEVICE TO SCREEN
   RETURN
```

SET DIRCASE

Specifies how directories are accessed on disk.

Syntax

```
SET DIRCASE LOWER | mixed | upper | <nDirCase>
```

Arguments

`LOWER | mixed | upper | <nDirCase>`

The case for directory access can be specified using a keyword or a numeric value for *<nDirCase>*.

Case sensitivity for directory access

Keyword	*<nDirCase>*	Description
MIXED *)	0	Mixed case is allowed
LOWER	1	Directories are converted to lower case
UPPER	2	Directories are converted to upper case
**) default*		

Description

SET DIRCASE defines how character strings programmed in xHarbour are converted before they are passed to the operating system for directory access. The command is only relevant when the operating system treats directory names differently by case. Windows is not case sensitive.

Info

See also:	Directory(), Set(), SET DIRSEPARATOR, SET FILECASE
Category:	Environment commands, SET commands, xHarbour extensions
Source:	rtl\set.c
LIB:	xhb.lib
DLL:	xhbdll.dll

SET DIRSEPARATOR

Specifies the default separator for directories.

Syntax

```
SET DIRSEPARATOR <cSeparator>
```

Arguments

<cSeparator>

A single character can be specified that is used a separator for directories. The default is a backslash.

Description

SET DIRSEPARATOR defines the separating character for directories when they are accessed by the operating system. The default separator is "\". Some operating systems use the forwar slash "/" as separator.

Info

See also:	Set(), SET DIRCASE, SET FILECASE
Category:	Environment commands, SET commands, xHarbour extensions
Source:	rtl\set.c
LIB:	xhb.lib
DLL:	xhbdll.dll

SET EOL

Defines the end-of-line character(s) for ASCII text files.

Syntax

```
SET EOL <cEol>
```

Arguments

<cEol>

A character string holding the end-of-line character(s) for ASCII text files. The default is Chr(13)+Chr(10).

Description

SET EOL is an environment command defining the new-line delimiter(s) for files in ASCII format. This applies to regular text files as well as files in DELIMITED or SDF format. The end-of-line character is operating system dependent. Windows uses Chr(13)+Chr(10) as end-of-line characters.

Note: the operating system dependent end-of-line characters can also be queried with function HB_OsNewLine().

Info

See also:	Chr(), HB_OsNewLine(), Set()
Category:	Environment commands, SET commands, xHarbour extensions
LIB:	xhb.lib
DLL:	xhbdll.dll

SET EPOCH

Determines the interpretation of date values without century digits.

Syntax

```
SET EPOCH TO <nYear>
```

Arguments

`TO <nYear>`

This is a numeric value specifying the reference year to which all character strings representing a date without century are compared.

Description

The SET EPOCH command determines the interpretation of date strings that do not carry century digits. Conversion functions such as CtoD() use the value of <*nYear*> as reference for a century. When decade and year of the string value to convert are larger than or equal to the last two digits of <*nYear*>, the value is assumed to fall into the same century like <*nYear*>. When the value is smaller, it is assumed to fall into the next century.

By default, SET EPOCH is 1900 so dates without century digits will fall into the twentieth century. xHarbour supports dates from 01/01/0100 to 12/31/2999.

Info

See also:	CtoD(), Date(), DtoC(), DtoS(), Set(), SET CENTURY, SET DATE
Category:	SET commands
Source:	rtl\set.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates the effect or a reference year set with
// SET EPOCH:

   PROCEDURE Main
      SET DATE FORMAT TO "mm/dd/yyyy"
      ? CtoD("05/27/1903")           // Result: 05/27/1903
      ? CtoD("05/27/55")             // Result: 05/27/1955

      SET EPOCH TO 1950
      ? CtoD("05/27/1910")           // Result: 05/27/1910
      ? CtoD("05/27/69")             // Result: 05/27/1969
      ? CtoD("05/27/06")             // Result: 05/27/2006
   RETURN
```

SET ERRORLOG

Defines the default error log file.

Syntax

```
SET ERRORLOG TO <cLogFile> [ADDITIVE]
```

Arguments

<cLogFile>

This is the name of the file that stores information about runtime errors. The default file is named "Error.log"

ADDITIVE

If this option is specified, new error information is appended to the log file. Otherwise, *<cLogFile>* is overwritten when a new error occurs.

Description

The SET ERRORLOG command is used in customized error handling routines where error information about runtime errors is written to a user-defined file. By default, runtime error information is stored in the Error.log file.

Note: xHarbour's default error handling routine is programmed in the file ERRORSYS.PRG.

Info

See also:	BEGIN SEQUENCE, Error(), ErrorBlock(), Set(), SET ERRORLOOP, TRY...CATCH
Category:	Environment commands, SET commands, xHarbour extensions
Source:	rtl\set.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates how a particular type of errors can be logged
// in a separate file. The file receives information about file open errors.

   PROCEDURE Main

      ? OpenDatabases()

   RETURN

   FUNCTION OpenDatabases()
      LOCAL aErrLog := Set( _SET_ERRORLOG )
      LOCAL oError
      LOCAL lSuccess := .T.

      SET ERRORLOG TO DbOpenError.log ADDITIVE

      USE Customer
      SET INDEX TO Cust01, Cust02

      SET ERRORLOG TO ( aErrLog[1] )
   RETURN lSuccess
```

SET ERRORLOOP

Defines the maximum recursion depth for error handling.

Syntax

```
SET ERRORLOOP TO <nRecursion>
```

Arguments

<nRecursion>

A numeric value specifying the maximum number of recursive calls within the xharbour error handling routine. The default value is 8.

Description

The SET ERRORLOOP command restricts recursive calls within the xHarbour error handling routine to a number of *<nRecursion>* calls. This is required when a runtime error occurs in an error handling routine. In this case, an endless recursion is suppressed by terminating the application when the error handling routine has not resolved a runtime error after *<nRecursion>* invocations.

Note: xHarbour's default error handling routine is programmed in the file ERRORSYS.PRG.

Info

See also:	Set()
Category:	Environment commands, SET commands, xHarbour extensions
LIB:	xhb.lib
DLL:	xhbdll.dll

SET ESCAPE

Sets the ESC key as a READ exit key.

Syntax

```
SET ESCAPE ON | off | ( <lOnOff> )
```

Arguments

`ON | off | ( <lOnOff> )`

This option toggles whether or not the ESC key is an exit key for the READ command. The default is ON or .T. (true), i.e. ESC terminates READ. To change the setting use OFF or .F. (false) as parameter. The parameter can also be specified as a logical expression enclosed in parentheses.

Description

The SET ESCAPE command enables or disables the ESC key as exit key for the READ command. When enabled, the ESC key voids all changes made in the buffer of the current GET object and ends the user input. With SET ESCAPE OFF, the ESC key is ignored during the READ command.

Note that SET KEY allows for associating a procedure with a key. This may override the SET ESCAPE setting.

Info

See also:	READ, ReadExit(), Set(), SET KEY, SetCancel(), SetKey()
Category:	SET commands
Header:	set.ch
Source:	rtl\set.c
LIB:	xhb.lib
DLL:	xhbdll.dll

SET EVENTMASK

Sets which events should be returned by the Inkey() function.

Syntax

```
SET EVENTMASK TO <nEventMask>
```

Arguments

```
TO <nEventMask>
```

<nEventMask> is a numeric value specifying the type of events to return by the Inkey() function. The argument can be a combination , or sum, of #define constants taken from the file Inkey.ch. It can also be specified as a numeric expression enclodes in parentheses.

Constants for *<nEventMask>*

Constant	Value	Events returned by Inkey()
INKEY_MOVE	1	Mouse pointer moved
INKEY_LDOWN	2	Left mouse button pressed
INKEY_LUP	4	Left mouse button released
INKEY_RDOWN	8	Right mouse button pressed
INKEY_RUP	16	Right mouse button released
INKEY_MMIDDLE	32	Middle mouse button pressed
INKEY_MWHEEL	64	Mouse wheel turned
INKEY_KEYBOARD	128	Key pressed
INKEY_ALL	255	All events are returned

The default value for *<nEventMask>* is INKEY_KEYBOARD, i.e. the Inkey() function returns only keyboard events and ignores the mouse.

Description

The SET EVENTMASK command determines which type of events should be returned by the Inkey() function. By default, only keyboard events are returned by Inkey(). This is a compatibility setting and should be changed to INKEY_ALL at program start. The reaction to events coming from the mouse can then be programmed in a DO CASE or SWITCH structure.

Info

See also: HB_KeyPut(), Inkey(), Lastkey(), Nextkey(), MCol(), MRow(), SET KEY, Set()
Category: Input commands, SET commands
Header: Inkey.ch
Source: rtl\set.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example changes the event mask for Inkey() to ALL events
// and displays the mouse cursor position.

   #include "Inkey.ch"

   PROCEDURE Main
      LOCAL nEvent

      SET EVENTMASK TO INKEY_ALL
```

```
    DO WHILE Lastkey() <> K_ESC
       ? nEvent := Inkey(0)

       IF nEvent > 999
          // display current mouse cursor position
          ?? MRow(), MCol()
       ENDIF
    ENDDO

 RETURN
```

SET EXACT

Determines the mode for character string comparison.

Syntax

```
SET EXACT on | OFF | ( <lOnOff> )
```

Arguments

`on | OFF | ( <lOnOff> )`

This option toggles if an exact comparison is performed with character strings. or not. The default is OFF or .F. (false), i.e. characters are compared up the length of the shorter string. To change the setting use ON or .T. (true) as parameter. The parameter can also be specified as a logical expression enclosed in parentheses.

Description

The SET EXACT command determines the mode for character string comparison with the comparison operators (=, >, <, =>, =<). The following rules are applied for comparison with SET EXACT set to ON:

1) When the right operand is a null string, the result is always .T. (true)
2) When the right operand is longer than the left operand, the result is always .F. (false)
3) When the right operand contains less or the same number of characters as the left operand, the result is only true if the left operand begins with the exact same characters as the right operand.

With SET EXACT set to OFF, the same rules apply except that trailing spaces are ignored in the comparison.

Notes: The exact equal operator == always performs an exact comparison irrespectively of the SET EXACT setting.

Search commands and functions such as SEEK, DbSeek() or SET RELATEION are not affected by SET EXACT.

Info

See also:	=, ==, Set()
Category:	SET commands
Source:	rtl\set.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates comparison results with SET EXACT

   PROCEDURE Main
      SET EXACT OFF
      ? "AB" = "ABCD"             // result: .F.
      ? "ABCD" = "AB"             // result: .T.
      ? "AB" = ""                 // result: .T.
      ? "" = "AB"                 // result: .F.
      ? "AB" = "AB   "            // result: .F.

      SET EXACT ON
      ? "AB" = "ABCD"             // result: .F.
      ? "ABCD" = "AB"             // result: .F.
      ? "AB" = ""                 // result: .F.
```

```
    ? "" = "AB"                    // result: .F.
    ? "AB" = "AB  "                // result: .T.

    ? "AB" == "AB  "               // result: .F.
RETURN
```

SET EXCLUSIVE

Sets the global EXCLUSIVE open mode for databases.

Syntax

```
SET EXCLUSIVE ON | off | ( <lOnOff> )
```

Arguments

`ON | off | ( <lOnOff> )`

This option toggles whether databases are opened in EXCLUSIVE mode or not. The default is ON or .T. (true), i.e. all databases opened with the USE command are exclusively accessible for the xHarbour application only. To change the setting use OFF or .F. (false) as parameter. The parameter can also be specified as a logical expression enclosed in parentheses.

Description

The SET EXCLUSIVE command changes the global setting for the default open mode with the USE command for databases. The default setting is ON. This setting is valid for all work areas and can be overriden for inividual work areas by specifying the option EXCLUSIVE or SHARED with the USE command.

Opening a database in EXCLUSIVE mode reserves access to this database to the xHarbour application that opened the database. Other applications in a network environment are denied access as long as the database is open for exclusive use. It is, therefore, recommended to use databases in SHARED mode, unless a database operation must be performed that requires exclusive access. This is necessary for PACK, REINDEX and ZAP operations.

Databases opened for exclusive use do not require record or file locks for changing data. This is required when a database is open in SHARED mode.

Info

See also:	FLock(), Neterr(), RLock(), Set(), UNLOCK, USE
Category:	Database commands, SET commands
Source:	rtl\set.c
LIB:	xhb.lib
DLL:	xhbdll.dll

SET FILECASE

Specifies how files are accessed on disk.

Syntax

```
SET FILECASE LOWER | mixed | upper | <nFileCase>
```

Arguments

`LOWER | mixed | upper | <nFileCase>`

The case for file access can be specified using a keyword or a numeric value for *<nFileCase>*.

Case sensitivity for file access

Keyword	*<nFileCase>*	Description
MIXED *)	0	Mixed case is allowed
LOWER	1	File names are converted to lower case
UPPER	2	File names are converted to upper case

**) default*

Description

SET FILECASE defines how character strings programmed in xHarbour are converted before they are passed to the operating system for file access. The command is only relevant when the operating system treats file names differently by case. Windows is not case sensitive.

Info

See also:	FCreate(), File(), FOpen(), Set(), SET DIRCASE
Category:	Environment commands, SET commands, xHarbour extensions
Source:	rtl\set.c
LIB:	xhb.lib
DLL:	xhbdll.dll

SET FILTER

Defines a condition for filtering records in the current work area.

Syntax

```
SET FILTER TO [<lExpression>]
```

Arguments

`TO <lExpression>`

This is logical expression used as filter condition. All records where the expression evaluates to .F. (false) are not visible in the current work area. When no argument is passed to SET FILTER, an existing filter condition is removed from the current work area.

Description

The SET FILTER command defines a logical expression for the current work area. The expression is evaluated for each record during relative database navigation with SKIP, GO TOP or GO BOTTOM. All records where the filter expression evaluates to .F. (false) are ignored and the record pointer is advanced until the expression yields .T. (true) or the end of file is reached.

As a result, all records where <*lExpression*> yields .F. (false) are logically filtered and not visible during relative database navigation. Absolute navigation with GOTO <*nRecordNumber*>, however, allows for positioning the record pointer to a record where <*lExpression*> yields .F. (false).

To activate a filter after SET FILTER is issued, the record pointer must be moved relatively. This is usually achieved with GO TOP.

Optimization: when database navigation appears to be slow with a filter condition, try to create an index with an expression that matches the filter and SET OPTIMIZE to ON.

Index: The SET FILTER condition has no effect on index creation with INDEX or REINDEX.

Info

See also:	DbClearFilter(), DbFilter(), DbSetFilter(), SET DELETED, SET OPTIMIZE, SET SCOPE
Category:	Database commands, SET commands
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates the effect of a filter condition

   PROCEDURE Main
      USE Customer
      INDEX ON Upper(LastName) TO Cust01

      GO TOP
      ? Recno(), Lastname              // result: 28 Abbey

      SET FILTER TO Upper(LastName) > "L"

      GO TOP
      ? Recno(), Lastname              // result: 182 MacDonald

      CLOSE Customer
   RETURN
```

SET FIXED

Toggles fixed formatting for displaying numbers in text-mode.

Syntax

```
SET FIXED on | OFF | (<lOnOff>)
```

Arguments

`ON | off | (<lOnOff>)`

The option toggles if numeric values are displayed with a fixed number of decimal places, or not. The default is OFF, or .F. (false). When set to ON or .T. (true), numbers are displayed with SET DECIMALS number of decimal places.

Description

SET FIXED is used in text-mode applications for displaying numeric values with a fixed number of decimal places. The number of decimals defaults to 2 and can be changed with SET DECIMALS. If a number has more decimal places, the number is rounded for display, not truncated. Note that SET FIXED affects only the display and not the accuracy of calculations with numeric values.

Info

See also:	Exp(), Log(), Round(), SET DECIMALS, Set(), Sqrt(), Val()
Category:	Environment commands, SET commands
Source:	rtl\set.c
LIB:	xhb.lib
DLL:	xhbdll.dll

SET FUNCTION

Associates a character string with a function key.

Syntax

```
SET FUNCTION <nFunctionKey> TO <cString>
```

Arguments

`<nFunctionKey>`

This is the numeric code for the function key to associate *<cString>* with. See table below.

`TO <cString>`

A character string to write into the keyboard buffer when the function key is pressed.

Description

SET FUNCTION is a compatibility command that associates a character string with a particular function key (F1..F10/F12). The function keys are numbered from 1 to 40 according to the following table:

Numeric codes for function keys

Number	Actual key
1 - 10	F1 - F10
11 - 20	Shift+F1 - Shift+F10
21 - 30	Ctrl+F1 - Ctrl+F10
31 - 40	Alt+F1 - Alt+F10

When a function key, or key combination, is associated with *<cString>*, this string is written into the keyboard buffer via SET KEY.

Info

See also: KEYBOARD, SET KEY, SetKey()
Category: Input commands, SET commands
Source: rtl\setfunc.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates how the string associated with a function
// key can be read from the keyboard buffer.

    #include "Inkey.ch"

    PROCEDURE Main
       LOCAL cStr
       LOCAL nKey, bBlock

       SET FUNCTION 2 TO "Edit"

       DO WHILE Lastkey() <> K_ESC
          nKey := Inkey(0)

          IF ( bBlock := SetKey( nKey ) ) <> NIL
             Eval( bBlock )
             ?
```

```
            DO WHILE Nextkey() <> 0  // displays: Edit
               ?? Chr( Inkey() )     // when F1 is pressed.
            ENDDO
         ENDIF

      ENDDO
   RETURN
```

SET HARDCOMMIT

Toggles immediate committing of changes to record buffers.

Syntax

```
SET HARDCOMMIT ON | off | (<lOnOff>)
```

Arguments

`ON | off | (<lOnOff>)`

The option toggles if data of a changed record is immediately commited to disk with the REPLACE command. The value ON or .T. (true), which is the default, commits changed data immediately. OFF or .F. (false) switch this mode off.

Description

Some replaceable database drivers support a delayed committing of records when their data is changed with the REPLACE command. The SET HARDCOMMIT setting influences this behavior. The default is ON.

Setting HARDCOMMIT to OFF can optimize record updates in databases but requires an explicit COMMIT command to indicate that data must be written to the database.

Info

See also:	COMMIT, REPLACE, Set(), USE
Category:	Database commands, SET commands, xHarbour extensions
Source:	rtl\set.c
LIB:	xhb.lib
DLL:	xhbdll.dll

SET INDEX

Opens one or more index files in the current work area.

Syntax

```
SET INDEX TO [<cIndexFile,...>] [ADDITIVE]
```

Arguments

`TO <cIndexFile,...>`

The names of the index files to open in the current work area can be specified as a comma separated list of literal file names or character expressions enclosed in parentheses. When this option is omitted, all index files open in the current work area are closed.

`ADDITIVE`

This option is only useful when *<cIndexFile,...>* is specified. In this case, the files are opened in addition to already open files. If ADDITIVE is omitted, all open index files are closed prior to opening new ones.

Description

The SET INDEX command opens index files in the current work area. The first index file specified becomes the controlling index. If the file contains multiple indexes, the first index in the multiple index file becomes the controlling index.

All open indexes are closed when SET INDEX is issued, unless the ADDITIVE option is used.

After the indexes are opened, the record pointer is positioned on the first logical record of the controlling index, unless the ADDITIVE option is used. ADDITIVE neither changes the controlling index nor the record pointer.

The command SET ORDER can later be used to change the controlling index.

Info

See also: CLOSE, DbClearIndex(), DbSetIndex(), INDEX, OrdListAdd(), REINDEX, SET ORDER, USE
Category: Database commands, Index commands
Source: rdd\dbcmd.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example creates three index files holding one index each.

   PROCEDURE Main
      USE Customer
      INDEX ON Upper(FirstName)           TAG FName TO Cust01
      INDEX ON Upper(LastName+Firstname)  TAG LName TO Cust02
      INDEX ON Upper(City)                TAG City  TO Cust03

      ? IndexOrd(), IndexKey()     // result: 1 Upper(City)

      SET INDEX TO Cust01, Cust02
      ? IndexOrd(), IndexKey()     // result: 1 Upper(FirstName)

      SET ORDER TO TAG LName
      ? IndexOrd(), IndexKey()     // result: 2 Upper(LastName+FirstName)
```

```
   SET INDEX TO Cust03 ADDITIVE
   ? IndexOrd(), IndexKey()     // result: 2 Upper(LastName+FirstName)

   SET ORDER TO 3
   ? IndexOrd(), IndexKey()     // result: 3 Upper(City)

   CLOSE Customer
RETURN
```

SET INTENSITY

Toggles usage of enhanced colors for GET and PROMPT

Syntax

```
SET INTENSITY ON | off | (<lOnOff>)
```

Arguments

`ON | off | (<lOnOff>)`

The option toggles whether or not the commands GET and PROMPT use the enhanced color for displaying the current entry fieldor menu item. The default is ON, or .T. (true). When set to OFF or .F. (false), GET and PROMPT use only the standard color for display.

Description

SET INTENSITY is a compatibility command for text-mode applications. It determines if @...GET and @...PROMPT use only the standard color for display or highlight the current entry field or menu item with the enhanced color. Refer to function SetColor() for standard and enhanced color.

Info

See also:	@...GET, @...PROMPT, @...SAY, Set(), SetColor()
Category:	Output commands, SET commands
Source:	rtl\set.c
LIB:	xhb.lib
DLL:	xhbdll.dll

SET KEY

Associates a key with a procedure.

Syntax

```
SET KEY <nInkeyCode> TO [<ProcedureName>]
```

Arguments

<nInkeyCode>

<nInkeyCode> is the numeric Inkey() value of the key to associate with a procedure. Refer to the file INKEY.CH for numeric key codes.

TO <ProcedureName>

The name of the procedure to execute must be specified as a literal name or as a macro expression. When the parameter is omitted, the procedure association is removed from the key.

Description

The SET KEY command associates a key with a procedure. The procedure is executed automatically when the key is pressed during a wait state. A wait state is employed by functions and commands that wait for user input, except for the Inkey() function. This includes the AChoice(), DBEdit(), MemoEdit() functions, and the ACCEPT, INPUT, READ and WAIT commands.

The associated procedure receives as parameters the return values of ProcName(), ProcLine() and ReadVar().

Notes: SET KEY is a compatibility command superseded by the SetKey() function.

The F1 key is automatically associated with a procedure named HELP at program start. If such a procedure is linked, it is executed when F1 is pressed during a wait state.

Info

See also:	Inkey(), KEYBOARD, LastKey(), PROCEDURE, ProcLine(), SetKey()
Category:	SET commands
Header:	Inkey.ch
Source:	rtl\setkey.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// Refer to the SetKey() function for an example.
```

SET LOG STYLE

Selects a style for logging data to log channels.

Syntax

```
SET LOG STYLE <nStyle>
```

Arguments

<nStyle>

This is a numeric value specifying the default formatting of log entries. #define constants are available in HbLog.ch that can be used for *<nStyle>*. To combine different formatting instructions, pass the sum of the #define constants for *<nStyle>*

Formatting constants for log messages

Constant	Value	Description
HB_LOG_ST_DATE *)	0x0001	Includes the date
HB_LOG_ST_TIME *)	0x0002	Includes the time as hh:mm:ss
HB_LOG_ST_SECS	0x0004	Includes seconds
HB_LOG_ST_LEVEL *)	0x0008	Includes priority level
HB_LOG_ST_ISODATE	0x0010	Includes ISO date
HB_LOG_ST_NAME	0x0020	Includes EXE file

**) default style*

!END

Description

The SET LOG STYLE command defines the output format of a log entry created with the LOG command. Default information like date and time stamps, or the priority level, can be included or excluded from a log entry.

Info

See also: CLOSE LOG, INIT LOG, LOG, TraceLog()
Category: Log commands, xHarbour extensions
Header: hblog.ch
Source: rtl\hblog.prg, rtl\hblognet.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays log messages to the console and
// demonstrates different formatting styles for log messages.

   #include "Hblog.ch"

   PROCEDURE Main
      INIT LOG Console()

      LOG "DEFAULT STYLE"

      SET  LOG STYLE HB_LOG_ST_DATE
      LOG "HB_LOG_ST_DATE"

      SET  LOG STYLE HB_LOG_ST_TIME
      LOG "HB_LOG_ST_TIME"
```

```
   SET  LOG STYLE HB_LOG_ST_SECS
   LOG "HB_LOG_ST_SECS"

   SET  LOG STYLE HB_LOG_ST_LEVEL
   LOG "HB_LOG_ST_LEVEL"

   SET  LOG STYLE HB_LOG_ST_ISODATE
   LOG "HB_LOG_ST_ISODATE"

   SET  LOG STYLE HB_LOG_ST_NAME
   LOG "HB_LOG_ST_NAME"

   SET  LOG STYLE HB_LOG_ST_NAME + HB_LOG_ST_ISODATE + HB_LOG_ST_SECS
   LOG "HB_LOG_ST_NAME + HB_LOG_ST_ISODATE + HB_LOG_ST_SECS"

   CLOSE LOG
RETURN
```

SET MARGIN

Defines the left margin for text-mode print output.

Syntax

```
SET MARGIN TO [<nSpaces>]
```

Arguments

<nSpaces>

This is a numeric value defining the number of blank spaces the print head should be advanced on the left side of a page before print output starts. Omitting the parameter sets the left margin to zero.

Description

SET MARGIN is a compatibility command for printing in text-mode. A left margin is produced by advancing the print head by *<nSpaces>* blank spaces. After this, data sent to the printer is output in the current print row. Note that PCol() reflects the number of blank spaces "printed" as left margin.

Info

See also:	@...SAY, PCol(), PRow(), SET DEVICE, SET PRINTER, Set(), SetPrc()
Category:	Output commands, SET commands
Source:	rtl\set.c
LIB:	xhb.lib
DLL:	xhbdll.dll

SET MEMOBLOCK

Defines the default block size for memo files.

Syntax

```
SET MEMOBLOCK TO <nBlockSize>
```

Arguments

`TO <nBlockSize>`

<nBlockSize> is a numeric value defining the new memo block size.

Description

The SET MEMOBLOCK command defines the memo block size for RDDs that support variable block sizes for memo fields. The block size is the minimum amount of space allocated in memo files when storing character strings in memo fields. When large strings are stored in memo fields, they occupy the number of Int(Len(cString)/*<nBlockSize>*) plus one blocks in the memo file.

The initial memo block size depends on the replaceable database driver used to open a database in a work area. The following table lists the default sizes for memo blocks of RDDs shipped with xHarbour:

Default memo block sizes

Memo type	Block size	Changeable
DBT	512	No
FPT	64	Yes
SMT	64	Yes

When the memo block size is changed for an RDD, this setting is valid for database files that are created new. It is not possible to change the memo block size for existing databases.

Info

See also: DbInfo(), RddSetDefault(), Set()
Category: Database commands, SET commands
Header: Set.ch
Source: rtl\set.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example deonstrates changing the memo block size for a newly
// created database

   #include "dbInfo.ch"

   REQUEST Dbfcdx

   PROCEDURE Main
      LOCAL aStruct

      RddSetDefault( "DBFCDX" )

      USE Customer SHARED ALIAS Cust VIA "DBFCDX"
      aStruct := DbStruct()
```

```
    ? DbInfo( DBI_MEMOEXT )          // result: .fpt
    ? DbInfo( DBI_MEMOBLOCKSIZE )    // result: 64

    SELECT 0
    SET MEMOBLOCK TO 128

    DbCreate( "Test.dbf", aStruct, "DBFCDX" )

    USE TEST
    ? DbInfo( DBI_MEMOEXT )          // result: .fpt
    ? DbInfo( DBI_MEMOBLOCKSIZE )    // result: 128

    CLOSE ALL
RETURN
```

SET MESSAGE

Defines the screen row for @...PROMPT messages.

Syntax

```
SET MESSAGE TO [<nRow> [CENTER | CENTRE]]
```

Arguments

<nRow>

This is anumeric value specifying the row coordinate on the screen where @...PROMPT messages are displayed. If *<nRow>* is set to zero, the display of messages is suppressed.

CENTER

When this option is used, messages are displayed centered on the screen. Otherwise, they are displayed left justified in the specified screen row.

Description

SET MESSAGE determines where and if messages are displayed for text-mode menus defined with the MESSAGE option of the @...PROMPT command. Text mode menus are activated with the MENU TO command.

Info

See also:	@...GET, @...PROMPT, MENU TO, SET WRAP, Set()
Category:	Output commands, SET commands
LIB:	xhb.lib
DLL:	xhbdll.dll

SET OPTIMIZE

Toggles filter optimization with indexed databases.

Syntax

```
SET OPTIMIZE ON | off | (<lToggle>)
```

Arguments

`ON | off | (<lOnOff>)`

This option toggles the filter optimization for database navigation in the current work area. The initial value of this setting is defined by the RDD used to open database ans index files. The value ON or .T. (true) enables the optimization, the value OFF or .F. (false) disables it. Note that when specifiying *<lOnOff>* as a logical expression, it must be enclosed in parentheses.

Description

The SET OPTIMIZE command determines whether or not to optimize filters in the current work area. Optimization is based on index expressions of indexes open in the current work area. When a filter condition matches with an index expression, the RDD, such as DBFCDX, compares values stored in the index rather than the database. This leads to an enhanced performance since less disk I/O is required during database navigation.

Info

See also:	Set(), SET FILTER, SET INDEX
Category:	Database commands, SET commands
Source:	rtl\set.c
LIB:	xhb.lib
DLL:	xhbdll.dll

SET ORDER

Selects the controlling index.

Syntax

```
SET ORDER TO <nIndexPos>
SET ORDER TO TAG <cIndexName> [IN <cIndexFile]
```

Arguments

`TO <nIndexPos>`

This is a numeric value specifying the ordinal position of the index to select as controlling index. Indexes are numbered in the sequence they are opened, beginning with 1.

When no argument is passed or when the order is 0, all indexes remain open, but database navigation follows the physical order of records. That is, there is no controlling index.

`TO TAG <cIndexName>`

This is the symbolic name of the index to select as the controlling index. It can be specified as a literal name or a character expression enclosed in parentheses. The TAG option is usually supported by RDDs capable of maintaining multiple-index files, such as DBFCDX.

`IN <cIndexFile>`

<cIndexFile> is the name of the file containing the index to select. If not specified, all index files open in the current work area are searched for the index with the name *<cIndexTag>*. The file name can be specified as a literal name or a character expression enclosed in parentheses. When the file extension is omitted, it is determined by the database driver that opened the file.

Description

The SET ORDER TO command selects the controlling index from a list of indexes open in the current work area. Indexes are opened with the SET INDEX command or the INDEX option of the USE command. The latter, however, is not recommended in a network environment.

The controlling index can either be specified by its ordinal position *<nIndexPos>*, or by its symbolic name *<cIndexName>*. The former is determined by the order in which indexes are opened (beginning with 1), while the latter is specified with the TAG option of the INDEX command when the index is created.

A controlling index affects the logical order of records in a work area. Relative database navigation with SKIP, GO TOP and GO BOTTOM follows this order, rather than the physical order how records are stored in a database.

The controlling index can be disabled temporarily by setting the index order to 0. This leaves all indexes open but database navigation follows the physical order.

Info

See also:	DbSetOrder(), INDEX, IndexKey(), IndexOrd(), OrdNumber(), OrdSetFocus(), SEEK, SET INDEX, USE
Category:	Database commands, Index commands
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example creates three index files holding one index each.
```

```
PROCEDURE Main
   USE Customer
   INDEX ON Upper(FirstName)           TAG FName TO Cust01
   INDEX ON Upper(LastName+Firstname)  TAG LName TO Cust02
   INDEX ON Upper(City)                TAG City  TO Cust03

   SET INDEX TO Cust01, Cust02, Cust03
   ? OrdNumber(),OrdKey()     // result: 1 Upper(FirstName)

   SET ORDER TO TAG LName
   ? OrdNumber(),OrdKey()     // result: 2 Upper(LastName+FirstName)

   SET ORDER TO 3
   ? OrdNumber(),OrdKey()     // result: 3 Upper(City)

   SET ORDER TO 0             ** No controlling index
   ? OrdNumber(),OrdKey()     // result: 0 ""
                              ** indexes are still open
   ? OrdKey(1)                // result: Upper(FirstName)
   ? OrdKey(2)                // result: Upper(LastName+FirstName)
   ? OrdKey(3)                // result: Upper(City)
   CLOSE Customer
RETURN
```

SET PATH

Set the search path for opening files.

Syntax

```
SET PATH TO [<cDirectoryList>]
```

Arguments

`TO <cDirectoryList>`

The *<cDirectoryList>* parameter specifies one or more directories to search for files when they are not found. A single directory in the list may contain a drive letter, followd by a colon, and subdirectories prefixed with a backslash. individual directories in *<cDirectoryList>* are separated with semicolons. *<cDirectoryList>* can be specified as a string literal or as a character expression enclosed in parentheses.

If *<cDirectoryList>* is omitted, the current drive and directory of the operating system is used as default.

Description

The SET PATH command defines an additional search path for opening files when they cannot be found. This affects databases and associated files.

Files are searched in the following order:

1) The current directory of the operating system.
2) The SET DEFAULT directory.
3) The directories specified with SET PATH.

Note that low-level file functions like FOpen() do not use SET PATH or SET DEFAULT for searching files.

Info

See also:	CurDir(), Set(), SET DEFAULT
Category:	SET commands
Source:	rtl\set.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates how to define a search path

   // single directory
   SET PATH TO D:\xhb\apps\

   // multiple directories
   SET PATH TO "D:\xhb\apps\;D:\xhb\data\;"
```

SET PRINTER

Enables or disables output to the printer or redirects printer output.

Syntax

```
SET PRINTER on | OFF | ( <lOnOff> )

or

SET PRINTER TO [<cDevice> | <cFile> [ADDITIVE]]
```

Arguments

`on | OFF | ( <lOnOff> )`

This option toggles whether console output is also directed to the printer or not. The default is OFF or .F. (false), i.e. the printer does not receive console output. To change the setting use ON or .T. (true) as parameter. The parameter can also be specified as a logical expression enclosed in parentheses.

`TO <cDevice>`

<cDevice> is the name of the device where all console output will be copied to. The device can be local or on a network. If the device name does not exist, a file with the name *<cDevice>* is created. The device name can be specified as a literal character string or as a character expression enclosed in parentheses.

`TO <cFile>`

<cFile> is the name of the output file. The name of the file can be specified as a literal character string or as a character expression enclosed in parentheses. The default file extension is PRN.

`ADDITIVE`

This option causes print output to be appended to the file *<cFile>*. When ADDITIVE is omitted, the file *<cFile>* is created as a new, empty file.

If SET PRINTER TO is specified without any argument, the currently specified device or file is closed and the default destination is reselected.

Description

The SET PRINTER command has two different functionalities.

The ON | OFF form of SET PRINTER determines if the console output is echoed to the printer. Commands that send output to the console are called console commands. Console commands, in general, do not specify row and column coordinates. All these commands (with exception of the ? and ?? command) have a TO PRINTER clause which echoes the console output to the printer. The output from the console commands is displayed on the console window, unless CONSOLE is OFF.

The SET PRINTER command determines destination of output from all commands and functions that send output to the printer. This includes the @...SAY command if device is set to printer. If the destination is a device, the names LPT1, LPT2, LPT3 (all parallel ports), COM1 and COM2 (serial ports), CON and PRN are valid. Default device is PRN.

When the destination is a file, it is created in the current default directory. When a file with the same name already exists, it is overwritten by the new one. No warning is generated when overwriting the file. Until the file is closed with SET PRINTER TO (with no argument), all output is echoed to this file.

Notes

The @...SAY command is not affected by SET PRINTER ON. To send the output of this command to the printer, use the SET DEVICE TO PRINTER instead.

Using the Windows operating system, xHarbour supports the SET PRINTER TO \\ServerName\PrinterName syntax.

When the file is closed, no end of file marker is added. To add the end of file marker, use ?? Chr(26) just before the closing the file with SET PRINTER TO and no argument.

Info

See also:	@...SAY, EJECT, PCol(), PRow(), Set(), SET ALTERNATE, SET CONSOLE, SET DEVICE, SetPrc()
Category:	Console commands, Printer commands, SET commands
Source:	rtl\console.c, rtl\set.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example echoes the output of the ? command to printer:

   PROCEDURE Main
      USE Employees NEW
      SET PRINTER ON

      DO WHILE !EOF()
         ? Employees->Name, Employees->Phone
         SKIP
      ENDDO

      EJECT
      SET PRINTER OFF

      CLOSE
   RETURN
```

SET RELATION

Synchronizes the record pointers in one or more work areas.

Syntax

```
SET RELATION TO [<expression1>|<nRecord1> INTO <cAlias1> [SCOPED]] ;
         [, [TO] <expressionN>|<nRecordN> INTO <cAliasN> [SCOPED]]
         [ADDITIVE]
```

Arguments

`TO <expression>`

This is the expression used to relate the current work area with a secondary one. The current work area becomes the parent of the secondary, which is also referred to as child work area. The effect of such a relation is that the record pointer in the child work area is synchronized with the parent work area. The database in the child work area *<cAlias>* must have a controlling index whose index expression matches *<expression>*.

The record pointer in the child work area is synchronized using a SEEK operation. The value of *<expression>* is passed to SEEK.

When no argument is passed, all active relations in the current work area are cleared.

`TO <nRecord>`

This is a numeric expression. The record pointer in the child work area is synchronized either with a GOTO operation or with a SEEK operation. GOTO is used when there is no controlling index in the child work area. SEEK is used when the controllng index in the child work area is of numeric data type.

`INTO <cAlias>`

The parameter specifies the alias name of the child work area. It can be specified as a literal alias name, or a character expression enclosed in parentheses.

`SCOPED`

When the SCOPED option is used, database navigation in the child work area is restricted (scoped) to the records that relate to the current record in the parent work area.

`ADDITIVE`

This option is only meaningful when relations are already defined in the current work area. In this case, all relations remain active, and new ones are defined. If ADDITIVE is omitted, all existing relations are cleared prior to defining new ones.

Description

The SET RELATION command relates a parent work area with one or more child work areas. This causes the record pointer in a child work area to be synchronized with the record pointer in the parent work area.

Synchronization of the record pointer in a child work area is accomplished either relative via an index, or absolute via a record number.

Relative synchronization

This requires a controlling index in the child work area. Each time the record pointer moves in the parent work area, the value of *<expression>* is SEEKed in the child work area. As a consequence, the data type of *<expression>* must match the data type of the controlling index in the child work area.

Absolute synchronization

When the child work area has no controlling index, or when the type of the index expression is not numeric and the relation expression is numeric, the child work area is synchronized via GOTO *<nRecord>*.

Notes

The record pointer in the child work area is positioned on Lastrec()+1 when there is no match with the relation expression.

It is illegal to relate a parent work area directly or indirectly with itself.

The SET RELATION command does not support SOFTSEEK. It always acts as if SOFTSEEK is set to OFF.

When relating two work areas based on matching record numbers, use the Recno() function for the SET RELATION TO expression. Make sure that the child work area has no controlling index.

Info

See also:	DbClearRelation(), DbRelation(), DbRSelect(), DbSetRelation(), Found(), RecNo(), SET INDEX, SET ORDER, SET SOFTSEEK
Category:	Database commands
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example lists data from a customer and an invoice database

   PROCEDURE Main
      USE Customer ALIAS Cust

      USE Invoice  ALIA Inv NEW
      INDEX ON CustNo TO Invoice01

      SELECT Cust
      SET RELATION TO CustNo INTO Inv

      // Invoice data changes whith each SKIP in Customer database
      DO WHILE .NOT. Eof()
         ? Cust->LastName, Cust->FirstName, Inv->Total
         SKIP
      ENDDO

      CLOSE ALL
   RETURN
```

SET SCOPE

Changes the top and/or bottom scope for navigating in the controlling index.

Syntax

```
SET SCOPE TO [<topScope> [, <bottomScope>]]
```

Arguments

`TO <topScope>`

This is an expression whose value is used to define the top scope, or top boundary, for indexed database navigation. Its data type must match with the data type of the index expression of the controlling index. Alternatively, *<topScope>* can be a code block that must return a value of the correct data type.

If *<bottomScope>* is omitted and *<topScope>* is defined, it is used for both, the top and bottom scope.

SET SCOPE TO with no argument clears a defined scope.

`, <bottomScope>`

This is the same like *<topScope>*, except that the bottom scope, or bottom boundary is defined.

Description

The SET SCOPE command defines or clears the top and bottom scope for indexed database navigation. The top and bottom scope values define the range of values valid for navigating the record pointer within the controlling index.

When a scope is set, the GO TOP command is equivalent to SEEK *<topScope>*, while GO BOTTOM is the same as SEEK *<bottomScope>* LAST.

Attempting to move the record pointer before *<topScope>* does not change the record pointer but causes function BoF() to return .T. (true).

When the record pointer is moved beyond *<bottomScope>*, it is advanced to the ghost record (Lastrec()+1) and function EoF() returns .T. (true).

Info

See also:	BoF(), EoF(), Found(), GO, OrdKey(), OrdScope(), SET INDEX, SET ORDER, SKIP
Category:	Database commands, Index commands
Header:	Ord.ch
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates indexed database navigation
// restricted by scopes.

   #include "Ord.ch"

   PROCEDURE Main
      USE Customer

      INDEX ON Upper(LastName) TO Cust01
      GO TOP
      ? Recno(), Bof(), Eof(), Lastname // result:   25 .F. .F. Abbey
```

```
   GO BOTTOM
   ? Recno(), Bof(), Eof(), Lastname // result:  127 .F. .F. Zwillick

   SET SCOPE TO "E", "G"
   GO TOP
   ? Recno(), Bof(), Eof(), Lastname // result:  558 .F. .F. Earley

   SKIP -1
   ? Recno(), Bof(), Eof(), Lastname // result:  558 .T. .F. Earley

   GO BOTTOM
   ? Recno(), Bof(), Eof(), Lastname // result: 1660 .F. .F. Gwinnell

   SKIP
   ? Recno(), Bof(), Eof(), Lastname // result: 1996 .F. .T.
RETURN
```

SET SCOPEBOTTOM

Changes the bottom scope for navigating in the controlling index.

Syntax

```
SET SCOPEBOTTOM TO [<bottomScope>]
```

Arguments

`TO <bottomScope>`

This is an expression whose value is used to define the bottom scope, or bottom boundary, for indexed database navigation. Its data type must match with the data type of the index expression of the controlling index. Alternatively, *<bottomScope>* can be a code block that must return a value of the correct data type.

Omitting the argument clears the bottom scope.

Description

The SET SCOPEBOTTOM command defines or clears the bottom scope for indexed database navigation. Refer to SET SCOPE for more explanations.

Info

See also:	OrdScope(), SET SCOPE
Category:	Database commands, Index commands
Header:	Ord.ch
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

SET SCOPETOP

Changes the top scope for navigating in the controlling index.

Syntax

```
SET SCOPETOP TO [<topScope>]
```

Arguments

`TO <topScope>`

This is an expression whose value is used to define the top scope, or top boundary, for indexed database navigation. Its data type must match with the data type of the index expression of the controlling index. Alternatively, *<topScope>* can be a code block that must return a value of the correct data type.

Omitting the argument clears the top scope.

Description

The SET SCOPETOP command defines or clears the top scope for indexed database navigation. Refer to SET SCOPE for more explanations.

Info

See also:	OrdScope(), SET SCOPE
Category:	Database commands, Index commands
Header:	Ord.ch
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

SET SCOREBOARD

Toggles the display of messages from READ and MemoEdit().

Syntax

```
SET SCOREBOARD ON | off | (<lOnOff>)
```

Arguments

`ON | off | (<lOnOff>)`

The option toggles whether or not messages generated by READ or MemoEdit() are displayed on the screen. The default is ON, or .T. (true). When set to OFF or .F. (false), READ and MemoEdit() messages are suppressed.

Description

SET SCOREBOARD is a compatibility command that influences the display of messages generated by MemoEdit() or READ. Such messages inform the user about the Insert/Overstrike mode during editing, or if invalid data is input. The messages are displayed in the top screen row, if activated.

Info

See also:	@...GET, MemoEdit(), READ, Set()
Category:	Output commands, SET commands
LIB:	xhb.lib
DLL:	xhbdll.dll

SET SOFTSEEK

Enables or disables relative seeking.

Syntax

```
SET SOFTSEEK on | OFF | ( <lOnOff> )
```

Arguments

```
on | OFF | ( <lOnOff> )
```

This option toggles whether the SEEK command employs relative seeking or not. The default is OFF or .F. (false), i.e. SEEK searches for an exact match. To change the setting use ON or .T. (true) as parameter. The parameter can also be specified as a logical expression enclosed in parentheses.

Description

The SET SOFTSEEK command changes the global setting for searching relatively with the SEEK command. The default setting is OFF. This setting is valid for all work areas. It can be overridden for individual work areas by calling the DbSeek() function in place of the SEEK command.

When SOFTSEEK is set to OFF, the SEEK command searches for an exact match with the searched value. If the value is not found, the record pointer is placed on the ghost record (Lastrec()+1) and function EoF() returns .T. (true).

When SOFTSEEK is set to ON and the searched value is not found, the record pointer is positioned on the record with the next higher index value. As a result, Eof() yields .F. (false), unless there is no higher index value.

Irrespective of the SOFTSEEK setting, function Found() returns .F. (false) when the searched value is not found.

Note: The SOFTSEEK setting is ignored for automatic SEEK operations that are performed due to SET RELATION in a child work area.

Info

See also:	DbSeek(), Eof(), Found(), SEEK, SET INDEX, SET ORDER, SET RELATION
Category:	Index commands, SET commands
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates the effect of SOFTSEEK using a database
// that contains "Jane" and "John" as names, but not "Jim"

   PROCEDURE Main
      USE Test
      INEX ON Name TO Test

      SET SOFTSEEK OFF

      SEEK "Jane"
      ? Found(), Eof(), Name  // result: .T., .F., "Jane"

      SEEK "Jim"
      ? Found(), Eof(), Name  // result: .F., .T., ""
```

```
   SET SOFTSEEK ON

   SEEK "Jim"
   ? Found(), Eof(), Name  // result: .F., .F., "John"

   SEEK "John"
   ? Found(), Eof(), Name  // result: .T., .F., "John"

   CLOSE Test
RETURN
```

SET STRICTREAD

Toggles read optimization for database access.

Syntax

```
SET STRICTREAD on | OFF | ( <lOnOff> )
```

Arguments

`on | OFF | ( <lOnOff> )`

The option toggles whether or not internal memory buffers are used for loading field variables into memory variables. The default is OFF, or .F. (false). When set to ON or .T. (true), performance optimization is suppressed.

Description

SET STRICTREAD is part of xHarbour's optimization for database access. This is accomplished by means of internal memory buffers used to hold the contents of field variables. If field variables must be accessed that are held in a buffer already, no file access is required when SET STRICTREAD is set to OFF. This is the default and optimizes field access.

When SET STRICTREAD is set to ON, the contents of field variables are always read from disk.

Info

See also: Set(), SET OPTIMIZE
Category: Database commands, SET commands, xHarbour extensions
Source: rtl\set.c
LIB: xhb.lib
DLL: xhbdll.dll

SET TIME

Specifies the time format for input and display.

Syntax

```
SET TIME FORMAT [TO] "hh[:mm][:ss][.ccc] [PM]"

or

SET TIME FORMAT [TO] (<cTimeFormat>)
```

Arguments

`TO "hh[:mm][:ss][.ccc] [AM|PM]"`

This is a literal time format string. It must be coded in the sequence Hour:Minute:Second where the colon is used as default delimiter. Optionally, the seconds value can be followed by a period and up to three digits (.ccc) indicating fractions of a second in milliseconds.

When the time should be formatted using a 12 hour clock, the suffix PM can be added. It must be separated with a blank space.

`TO <cTimeFormat>`

Instead of a literal time format string, a character expression enclosed in parentheses can be specified.

Description

The SET TIME command specifies the format for the display of the Time part of DateTime() values. It is a global setting valid for an entire xHarbour application.

Info

See also:	DateTime(), Set(), SET CENTURY, SET DATE, SET EPOCH
Category:	SET commands, xHarbour extensions
Source:	rtl\set.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates different Date and Time formatting
// of a DateTime value

   PROCEDURE Main

      LOCAL dDateTime := {^ 2007/04/26 16:31:24.789 }

      ? Set(_SET_TIMEFORMAT)           // result:  hh:mm:ss.cc

      ? dDateTime                      // result:  04/26/07 16:31:24.78

      SET TIME FORMAT TO "hh:mm pm"

      ? dDateTime                      // result:  04/26/07 04:31 PM

      SET DATE TO ITALIAN
      SET CENTURY ON
      SET TIME FORMAT TO "hh|mm|ss"
```

```
   ? dDateTime                        // result:  26-04-2007 16|31|24
RETURN
```

SET TRACE

Toggles output of function TraceLog().

Syntax

```
SET TRACE ON | off | (<lOnOff>)
```

Arguments

`ON | off | (<lOnOff>)`

The option toggles whether or not output of function TraceLog() is directed into the trace log file. The default is ON, or .T. (true). When set to OFF or .F. (false), function TraceLog() produces no output.

Description

The SET TRACE command is part of xHarbour's tracing system which allows for logging function, method or procedure calls in an arbitrary way. The default file receiving entries from the TraceLog() function is named Trace.log. It can be changed with Set(_SET_TRACEFILE).

Info

See also: Set(), TraceLog()
Category: Environment commands, SET commands, xHarbour extensions
Source: rtl\set.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example produces a trace.log file with an log entry
// for PROCEDURE Test()

   PROCEDURE Main

      SET TRACE OFF
      TraceLog()

      ? "Hello World"

      SET TRACE ON
      Test( "Hi there" )

   RETURN


   PROCEDURE Test( cMsg )

      Tracelog( cMsg )
      ? cMsg

   RETURN
```

SET TYPEAHEAD

Dimensions the size of the keyboard buffer.

Syntax

```
SET TYPEAHEAD TO <nKeyboardSize>
```

Arguments

`TO <nKeyboardSize>`

This is a numeric value specifiying the maximum number of key strokes that can accumulate in the internal keyboard buffer. The value must lie within the range of 0 and 4096.

Description

The SET TYPEAHEAD command sets the number of keystrokes that the internal keyboard buffer can hold. This command does not affect the amount of key strokes that can be added to the keyboard buffer programmatically using the KEYBOARD command. Before the new buffer size is set, the buffer is cleared first.

When *<nKeyboardSize>* is set to zero, keyboard polling is suspended. Explicit requests for keyboard input by using the NextKey() function, for example, enables the buffer temporarily.

Info

See also:	CLEAR TYPEAHEAD, Inkey(), KEYBOARD, SetCancel(), Set()
Category:	SET commands
Source:	rtl\set.c
LIB:	xhb.lib
DLL:	xhbdll.dll

SET UNIQUE

Includes or excludes non-unique keys to/from an index.

Syntax

```
SET UNIQUE on | OFF | ( <lOnOff> )
```

Arguments

`on | OFF | ( <lOnOff> )`

This option toggles whether records resulting in duplicate index entries are included in an index or not. The default is OFF or .F. (false), i.e. all records are included in an index. To change the setting use ON or .T. (true) as parameter. The parameter can also be specified as a logical expression enclosed in parentheses.

Description

The SET UNIQUE command determines how many records of a database, that result in the very same index value, are included in an index. With SET UNIQUE OFF, all records are included. When the setting is ON, a record is only included in an index when the index value of the record does not exist in the index. That is, an index value exists only once in an index.

SET UNIQE is a global setting valid for all work areas. It can be overridden for individual work areas with the UNIQUE option of the INDEX command.

Note: It is important to understand that the SET UNIQUE setting affects indexes, not databases. It is possible to add, for example, the name "Miller" ten times to an address database. When the database is indexed on "LASTNAME" and SET UNIQUE is ON, only the first record containing "Miller" is added to the index. If this index is later selected as the controlling index, the remaining nine "Miller" records are not visible.

Info

See also:	DbCreateIndex(), INDEX, OrdCreate(), PACK, REINDEX, SEEK, Set()
Category:	Index commands, SET commands
Source:	rdd\dbcmd.c, rtl\set.c
LIB:	xhb.lib
DLL:	xhbdll.dll

SET VIDEOMODE

Changes the current video mode of the application.

Syntax

```
SET VIDEOMODE TO <nMode>
```

Arguments

`TO <nMode>`

<nMode> is a numeric value representing a certain video mode.

Description

The SET VIDEOMODE command changes the current display to text mode and different graphic modes. Two modes are supported for which the #define constant LLG_VIDEO_TEXT and LLG_VIDEO_VGA_640_480_16 are used.

Note: When changing from LLG_VIDEO_TEXT to LLG_VIDEO_VGA_640_480_16, all displayed text lines are converted to the equivalent graphic display. This conversion is not applied when switching back.

Info

See also:	Set()
Category:	Output commands, SET commands
Header:	Set.ch
Source:	rtl\set.c
LIB:	xhb.lib
DLL:	xhbdll.dll

SET WRAP

Toggles wrapping of the highligh bar in text-mode menus.

Syntax

```
SET WRAP on | OFF | ( <lOnOff> )
```

Arguments

`on | OFF | ( <lOnOff> )`

The option toggles whether or not the highlight bar of a MENU TO menu wraps when the user reaches the first or last menu item. The default is OFF, or .F. (false), i.e. wrapping of the highlight bar is disabled.

Description

SET WRAP is a compatibility command defining the behavior of text-mode menus activated with MENU TO. When SET WRAP is set to ON, the highlight bar wraps to the last menu item if the user reaches the first menu item, and vice versa.

Info

See also:	@...PROMPT, MENU TO, SET MESSAGE, Set()
Category:	Output commands, SET commands
Source:	rtl\set.c
LIB:	xhb.lib
DLL:	xhbdll.dll

SKIP

Moves the record pointer to a new position.

Syntax

```
SKIP [<nRecords>] [ALIAS <cAlias> | <nWorkArea>]
```

Arguments

`<nRecords>`

This is a numeric expression specifying the number of records to move the record pointer from its current position. Positive values for *<nRecords>* move the record pointer downwards and negative values move it upwards.

When no argument is passed, *<nRecords>* defaults to 1.

`ALIAS <cAlias> | <nWorkArea>`

The ALIAS option specifies the work area in which to move the record pointer. It can be specified as a symbolic alias name or as the numeric work area number.

Description

The SKIP command moves the record pointer of a work area to a new position. The new position is specified as a "distance" from the current record pointer, i.e. in number of records. A negative "distance" moves the record pointer towards the begin-of-file while positive value for *<nRecords>* moves it towards the end-of-file. A SKIP command issued with no value for *<nRecords>* advances the record pointer to the next record towards the end.

SKIP performs a logical navigation of the record pointer. That is, it navigates database records according to the controlling index, active filters and scopes (see INDEX, SET FILTER and SET SCOPE). A physical navigation is performed when no index and filter is active. In this case, SKIP navigates the record pointer in the order how records are stored in a database.

The command allows for navigating the record pointer of work areas other than the current work area by using the ALIAS option.

The record pointer cannot be moved outside physical or logical boundaries. Both are reached with GO TOP or GO BOTTOM.

When SKIP moves the record pointer using a negative distance and reaches the TOP, the record pointer remains on TOP and function BoF() returns .T. (true).

When SKIP moves the record pointer using a positive value for *<nRecords>* that goes beyond BOTTOM, the record pointer is positioned on the ghost record (Lastrec()+1) and function EoF() returns .T. (true).

Networking: Use SKIP 0 to make changes to a record visible to other work stations while a record is locked.

Info

See also: Bof(), COMMIT, DbSkip(), Eof(), GO, LOCATE, RecNo(), SEEK
Category: Database commands
Source: rdd\dbcmd.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The the example demonstrates the effect of SKIP

   PROCEDURE Main
      USE Customer

      ? LastRec()                      // result: 1995
      ? RecNo()                        // result: 1

      SKIP 200
      ? RecNo()                        // result: 201

      SKIP -500
      ? RecNo()                        // result: 1
      ? Bof()                          // result: .T.

      GO BOTTOM
      SKIP 10
      ? RecNo()                        // result: 1996
      ? Eof()                          // result: .T.

      CLOSE Customer
   RETURN
```

SORT

Creates a new, physically sorted database.

Syntax

```
    SORT TO <cDatabase> ;
         ON <fieldName1> [/[A | D][C]] ;
         [, <fieldNameN> [/[A | D][C]]] ;
           [<Scope>] ;
     [WHILE <lWhileCondition>] ;
       [FOR <lForCondition>] ;
  [CODEPAGE <cCodePage>] ;
[CONNECTION <nConnection>] ]
```

Arguments

`TO <cDatabase>`

This is the name of the database file to create. It can include path and file extension. When no path is given, the file is created in the current directory. The default file extension is DBF. *<cDatabase>* can be specified as a literal file name or as a character expression enclosed in parentheses.

`ON <fieldName>`

The field names to use from the current work area for the sort operation must be specified as literal names, or as character expressions enclosed in parentheses.

`/A | /D`

These flags define the sorting order for a field. /D means Descending. /A stands for Ascaneding and is the default.

`/C`

This flag is only meaningful for fields of data type Character. If the flag is used, sorting of character values is case-insensitive.

`<Scope>`

This option defines the number of records to sort. It defaults to ALL. The NEXT *<nCount>* scope sorts the next *<nCount>* records, while the REST scope sorts records beginning with the current record down to the end of file.

`WHILE <lWhileCondition>`

This is a logical expression indicating to continue sorting while the condition is true. The SORT command stops processing as soon as *<lWhileCondition>* yields .F. (false).

`FOR <lForCondition>`

This is a logical expression which is evaluated for all records in the current work area. Those records where *<lForCondition>* yields .T. (true) are added to the target database.

`CODEPAGE <cCodePage>`

This is a character string specifying the code page to use for character strings stored in the new database. It defaults to the return value of HB_SetCodePage().

`CONNECTION <nConnection>`

This option specifies a numeric server connection handle. It is returned by a server connection function which establishes a connection to a database server, such as SR_AddConnection() of the xHarbour Builder SQLRDD. When CONNECTION is used, the SORT TO command creates a database on the server.

Description

The SORT command creates a new database file and adds to it records from the current work area in sorted order. Fields of data type Character are sorted according to their ASCII values, unless the /C flag is used. In this case, the sorting of character values is case-insensitive. Numeric fields are sorted by value, Date fields in chronological order and Logical fields with .T. (true) being the high value. Memo fields cannot be sorted.

The default sorting order is ascending (/A flag). It can be changed to descending using the /D flag.

The SORT command requires a file lock in the current work area (see function FLock()). Alternatively, the database must be used in EXCLUSIVE mode.

Records marked for deletion are not included in the target database when SET DELETED is set to ON.

Info

See also: ASort(), FLock(), INDEX, OrdCreate(), USE
Category: Database commands
Source: rdd\dbcmd.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example creates a physically sorted database.

   PROCEDURE Main
      USE Customer

      SORT TO Cust01 ON Lastname, Firstname

      USE Cust01

      Browse()

      CLOSE Cust01
   RETURN
```

STORE

Assigns a value to one ore more variables.

Syntax

```
STORE <expression> TO <varName,...>

or

<varName1> := [ <varName2> := ...] <expression>
```

Arguments

`<expression>`

The value of *<expression>* is assigned to the variables specified with *<varName>*.

`TO <varName,...>`

The symbolic names of variables to assign a value to is specified as a comma separated list. Variables that do not exist are created as PRIVATE variables.

Description

The STORE command exists for compatibility reasons. It is superseded by the inline-assignment operator (:=).

Info

See also:	:=, GLOBAL, FIELD, LOCAL, STATIC, MEMVAR
Category:	Memory commands
LIB:	xhb.lib
DLL:	xhbdll.dll

SUM

Calculates the sum of numeric expressions in the current work area.

Syntax

```
   SUM <expressions,...> ;
    TO <resultVarNames,...> ;
      [<Scope>] ;
[WHILE <lWhileCondition>] ;
  [FOR <lForCondition>]
```

Arguments

`<expressions,...>`

This is a comma separated list of numeric expressions that are evaluated for the records of the current work area.

`TO <resultVarNames,...>`

A comma separated list of variable names that are assigned the results of the calculation. The number of *<resultVarNames,...>* must match exactly the number of *<expressions,...>*.

`<Scope>`

This option defines the number of records to sum. It defaults to ALL. The NEXT *<nCount>* sums the next *<nCount>* records, while the REST scope sums records beginning with the current record down to the end of file.

`WHILE <lWhileCondition>`

This is a logical expression indicating to continue calculation while the condition is true. The SUM operation stops as soon as *<lWhileCondition>* yields .F. (false).

`FOR <lForCondition>`

This is a logical expression which is evaluated for all records in the current work area. Those records where *<lForCondition>* yields .T. (true) are included in the calculation.

Description

The SUM command is used to calculate sum values for one or more numeric expressions. The expressions are evaluated in the current work area. The number of records to include in the calculation can be restricted using a FOR and/or WHILE condition or by explicitly defining a scope.

Records marked for deletion are not included in the calculation when SET DELETED is set to ON.

Info

See also:	AVERAGE, DbEval(), TOTAL
Category:	Database commands
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example adds daily sales for the month of December.

   PROCEDURE Main
      LOCAL nSales := 0

      USE Invoice
```

```
   SUM FIELD->Total TO nSales FOR ;
       Month(PayDate) == 12 .AND. Year(PayDate)=Year(Date())

   ? "December sales:", nSales

   CLOSE Invoice
RETURN
```

TEXT

Displays a block of literal text.

Syntax

```
TEXT [TO PRINTER] [TO FILE <fileName>]
   <text>
ENDTEXT

or

TEXT INTO <varName>
   <text>
ENDTEXT
```

Arguments

`TO PRINTER`

The text is additionally output to the printer.

`TO FILE <fileName>`

The text is additionally output to the file <*fileName*>. It can be specified as a literal file name or as a character expression enclosed in parentheses. The default extension is TXT.

`INTO <varName>`

This is the symbolic name of the variable to assign <*text*> to.

`<text>`

This is a literal text block enclosed in TEXT and ENDTEXT. The text is output to the screen and displayed exactly as written in the PRG source code file.

Description

The TEXT...ENDTEXT command outputs a block of literal text to the console. Output can be directed additionally to a printer or a file. When SET CONSOLE is set to OFF, the screen output is suppressed.

ALternatively, the text is assigned as character string to the variable <*varName*> when the INTO option is used.

Info

See also:	? | ??, @...SAY, MLCount(), MemoLine(), SET ALTERNATE, SET CONSOLE, SET DEVICE, SET PRINTER
Category:	Console commands
Source:	rtl\text.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

TOTAL

Creates a new database summarizing numeric fields by an expression.

Syntax

```
     TOTAL ON <expression> ;
     [FIELDS <fieldName,...>] ;
          TO <xcDatabase> ;
            [<Scope>] ;
      [WHILE <lWhileCondition>] ;
        [FOR <lForCondition>]
   [CODEPAGE <cCodePage> ]  ;
 [CONNECTION <nConnection>] ]
```

Arguments

ON <expression>

This is an expression whose value identifies groups of records to summarize. Each time the value of *<expression>* changes, a new record is added to the target database and a new calculation begins. Records in the current work area should be indexed or sorted by *<expression>*.

FIELDS <fieldNames,...>

The names of the numeric fields to summarize can be specified as a comma separated list of literal field names or character expressions enclosed in parentheses. When this option is omitted, no calculation is performed. Instead, only the records that match *<expression>* first are added to the target database.

TO <cDatabase>

This is name of the database file to create. It can include path and file extension. When no path is given, the file is created in the current directory. The default file extension is DBF. *<cDatabase>* can be specified as a literal file name or as a character expression enclosed in parentheses.

<Scope>

This option defines the number of records to process. It defaults to ALL. The NEXT *<nCount>* processes the next *<nCount>* records, while the REST scope totals records beginning with the current record down to the end of file.

WHILE <lWhileCondition>

This is a logical expression indicating to continue calculation while the condition is true. The TOTAL operation stops as soon as *<lWhileCondition>* yields .F. (false).

FOR <lForCondition>

This is a logical expression which is evaluated for all records in the current work area. Those records where *<lForCondition>* yields .T. (true) are included in the calculation.

CODEPAGE <cCodePage>

This is a character string specifying the code page to use for character strings stored in the new database. It defaults to the return value of HB_SetCodePage().

CONNECTION <nConnection>

This option specifies a numeric server connection handle. It is returned by a server connection function which establishes a connection to a database server, such as SR_AddConnection() of the xHarbour Builder SQLRDD. When CONNECTION is used, the TOTAL command creates a database on the server.

Description

The TOTAL command is used to calculate numeric totals for groups of records. A record group is identified by the value of <*expression*>. Each time this value changes, a new record is added to the target database and the values of numeric fields listed in <*fieldName,...*> are summarized until <*expression*> changes again. As a consequence, the records in the current work area must be indexed or sorted by <*expression*> for a correct calculation.

The target database has the same structure as the source database. This requires the field length of numeric fields be large enough to hold total values. If the result is too large, a runtime error is generated.

Records marked for deletion are not included in the calculation when SET DELETED is set to ON.

Info

See also: AVERAGE, INDEX, SORT, SUM
Category: Database commands
Source: rdd\dbcmd.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example calculates total sales from an invoice database
// by part number

   PROCEDURE Main
      USE Invoice
      INDEX ON PartNo TO Temp

      TOTAL ON PartNo TO TotalSales

      USE TotalSales
      Browse()

      CLOSE ALL
   RETURN
```

UNLOCK

Releases file and/or record locks in the current or all work areas

Syntax

```
UNLOCK [ALL]
```

Arguments

`ALL`

This option causes all locks be released in all work areas. Without this option, only the locks in the current work area are released.

Description

The UNLOCK command releases file and record locks in the current or all work areas previously set with RLock() or FLock().

Locking is required when a database is open in SHARED mode and changed data must to be written to a database file.

Info

See also:	DbRUnlock(), DbUnlock(), DbUnlockAll(), FLock(), RLock(), SET RELATION, USE
Category:	Database commands
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

UPDATE

Updates records in the current work area from a second work area.

Syntax

```
UPDATE FROM <cAlias> ON <expression> ;
    [RANDOM] ;
    REPLACE <fieldName1> WITH <value1> ;
          [,<fieldNameN> WITH <valueN> ]
```

Arguments

`FROM <cAlias>`

This is the alias name of the second work area used update the current work area. It can be specified as a literal alias name or a character expression enclosed in parentheses.

`ON <expression>`

This is an expression whose value is searched in the second work area to find the records for updating the current work area.

`RANDOM`

The option allows records in the FROM work area to be in any order. If this option is specified, the current work area must be indexed on <*expression*>.

`REPLACE <fieldName>`

This is the symbolic name of a field variable in the current work area to assign a new value to.

`WITH <value>`

The result of of <*value*> is assigned to <*fieldName*>. Note that <*value*> must yield a value of the same data type as the field valriable. Memo fields must be assigned values of data type Character.

Description

The UPDATE command replaces fields in the current work area with values from a second work area based on <*expression*>. An update occurs when <*expression*> provides the same value in both work areas. This requires the current work area be indexed on <*expression*> and that no multiple key values exist in this index. When there are multiple records with the same key value, only the first record with this key value is updated. The FROM work area, however, can have duplicate key values.

If RANDOM is not specified both work areas must be indexed on <*expression*>. If RANDOM is specified, only the current work area needs to be indexed on <*expression*>.

In a multi-user application in network operation, the file in the current work area must be exclusively open or a file lock with FLock() must be applied.

Records marked for deletion are not processed in either work area when SET DELETED is set to ON.

Info

See also: AVERAGE, DbEval(), INDEX, OrdCreate(), REPLACE, SORT, SUM, TOTAL
Category: Database commands
Source: rdd\dbcmd.c
LIB: xhb.lib
DLL: xhbdll.dll

USE

Opens a database and its associated files in a work area.

Syntax

```
        USE [<cDatabase> ;
      [INDEX <cIndexFiles,...>] ;
      [ALIAS <cAlias>] ;
             [EXCLUSIVE | SHARED] ;
             [NEW] ;
             [READONLY] ;
        [VIA <cRDD>] ;
   [CODEPAGE <cCodePage>] ;
 [CONNECTION <nConnection>]]
```

Arguments

`<cDatabase>`

This is name of the database file to open. It can include path and file extension. The default file extension is DBF. <*cDatabase*> can be specified as a literal file name or as a character expression enclosed in parentheses.

If no argument is specified, all files open in the current work area are closed.

`INDEX <cIndexFiles,...>`

The names of the index files to open along with the database can be specified as a comma separated list of literal file names or character expressions enclosed in parentheses. When this option is omitted, the datbase is opened without indexes.

`ALIAS <cAlias>`

This is the symbolic alias name of the work area. It can be specified as a literal alias name or a character expression enclosed in parentheses. It defaults to the file name of <*cDatabase*> without extension.

`EXCLUSIVE`

This option opens the database file for exclusive access. All other users in a network environment cannot access the database until the file is closed.

`SHARED`

This option opens the database file for shared use in a network environment.

`NEW`

NEW selects the next free work area and then opens the database. Without the option, all files open in the current work area are closed before <*cDatabase*> is opened in the current work area.

`READONLY`

This option is required if the file <*cDatabase*> is marked as read-only in the file system.

`VIA <cRDD>`

The option specifies the replaceable database driver (RDD) to use for maintaining the database and its associated files. It defaults to RddSetDefault() and must be a character expression.

`CODEPAGE <cCodePage>`

This is a character string specifying the code page to use for character strings stored in the database. It defaults to the return value of HB_SetCodePage().

`CONNECTION <nConnection>`

This option specifies a numeric server connection handle. It is returned by a server connection function which establishes a connection to a database server, such as SR_AddConnection() of the xHarbour Builder SQLRDD. When CONNECTION is used, the USE command opens a database on the server.

Description

The USE command opens a database file along with its associated memo files in the current work area. Optionally, index files belonging to the database can be opened as well. Only one database can be open at any time in one work area. The maximum number of work areas is 65535.

USE operates in the current work area, which can be selected using the SELECT command. As an alternative, the NEW option of USE selects the next free work area as current and then opens the file. When the database is open, its record pointer is positioned on the first logical record, if indexes are opened, or on record number 1.

Index files can be opened along with the database using the INDEX option. It is, however, recommended to open first the database file alone, check the success of file opening with NetErr() and then open associated index files in a separate operation using SET INDEX.

File access to the database in a multi-user environment is controlled with the options SHARED and EXCLUSIVE. Shared file access requires record locking for updating records during database operations. This is accomplished with function RLock().

Info

See also:	CLOSE, DbSetIndex(), DbUseArea(), NetErr(), OrdListAdd(), RddSetDefault(), SELECT, SET DEFAULT, SET INDEX, SET PATH, Used()
Category:	Database commands
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates the recommended procedure of opening a databse
// and its indexes in a multi-user environment:

   PROCEDURE Main
      USE Customer SHARED NEW

      IF .NOT. NetErr()
         SET INDEX TO Cust01, Cust02, Cust03
      ELSE
         ? "Unable to open database"
         BREAK
      ENDIF

      <database operations>

      CLOSE ALL
   RETURN
```

WAIT

Suspend program execution until a key is pressed.

Syntax

```
WAIT [<msgVar>] [TO <varName>]
```

Arguments

`<msgExpr>`

An expression returning a message to display while the program waits for user input can be specified. If omitted, the user is prompted with the default message: "Press any key to continue..."

`TO <varName>`

Optionally, the name of a variable to receive the pressed key as a character value can be specified. When *<varName>* does not exist or is not visible, it is created as a private variable.

Description

WAIT is a simple console input command that employs a wait state until the user presses a key. The command displays a message on the screen in a new line. The message can be suppressed by passing a null string ("") for *<msgExpr>*.

The key pressed by the user can be assigned as character value to a variable for further processing.

Note: WAIT is a compatibility command that should be replaced with function Inkey().

Info

See also:	@...GET, ACCEPT, Inkey(), INPUT, MENU TO
Category:	Input commands
Source:	rtl\wait.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows how to query for a single key stroke.

   PROCEDURE Main
      ? "Program started..."
      ?
      WAIT "Press Q to QUIT, any other key to continue" TO cKey

      IF cKey $ "qQ"
         ? "Program aborted"
         QUIT
      ENDIF

      ? "Normal program termination".
   RETURN
```

ZAP

Delete all records from the current database file

Syntax

```
ZAP
```

Description

The ZAP command deletes all records from a database, leaving an empty database file consisting of the file header only. All associated files like memo file or open index files are emptied as well.

The file operations performed by ZAP require a database file be open in EXCLUSIVE mode. The operation is irreversible, i.e. all data stored in a ZAPped file is lost. It is impossible to RECALL data when the ZAP command is complete.

Use DELETE to mark a record for deletion without losing stored data. Records marked for deletion can be recalled.

Info

See also:	DELETE, PACK, RECALL, USE
Category:	Database commands
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows how to prepare for a ZAP operation in a network

   PROCEDURE Main
      USE Customer EXCLUSIVE

      IF .NOT. NetErr()
         SET INDEX TO Cust01, Cust02, Cust03
         ZAP
      ELSE
         ? "ZAP failed"
      ENDIF

      CLOSE Customer
   RETURN
```

Function Reference

AAdd()

Adds one element to the end of an array.

Syntax

```
AAdd( <aArray>, <xValue> ) --> xValue
```

Arguments

`<aArray>`

A variable holding an array to add an element to.

`<xValue>`

An arbitrary value to add to *<aArray>*.

Return

The function returns the value added to the array.

Description

The AAdd() function increases the length of the array *<aArray>* by one element and assigns the value *<xValue>* to the new element. It is the last element of *<aArray>*.

AAdd() dynamically grows an array one element at a time. This is convenient when less than 100 elements should be added to *<aArray>*. To collect more elements in a loop, for example, it is recommended to dimension the array adequately or to grow multiple elements at a time using the ASize() function.

When *<xValue>* is of data type Array, a multi-dimensional array is created.

Info

See also:	ADel(), AEval(), AFill(), AIns(), ASize()
Category:	Array functions
Source:	vm\arrayshb.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// This example adds 10 numbers to an array and displays a message
// after adding each element.

   PROCEDURE MAIN()
      LOCAL n, aArray := {}

      ? Len( aArray )                  // result: 0

      FOR n := 1 to 10
         AAdd( aArray, n )
         ? "Added value", n, "to the array."
      NEXT

      ? Len( aArray )                  // result: 10
   RETURN NIL
```

Abs()

Returns the absolute value of a numeric expression.

Syntax

```
Abs( <nExpression> ) --> nPositive
```

Arguments

<nExpression>

A numeric expression whose absolute value is calculated.

Return

The function returns the absolute value of the passed argument. The result is always greater than or equal to zero.

Description

The function removes the minus sign from negative numeric values and leaves non-negative values unchanged. This allows for determining the magnitude of numbers regardless of their sign.

Info

See also:	Exp(), Int(), Log(), Max(), Min()
Category:	Numeric functions
Source:	rtl\abs.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates results from Abs()

   PROCEDURE Main()
      LOCAL nVal1 := 200
      LOCAL nVal2 := 900

      ? nVal1 - nVal2                  // result: -700

      ? Abs( nVal1 - nVal2 )           // result:  700
      ? Abs( nVal2 - nVal1 )           // result:  700

      ? Abs( 0 )                       // result: 0
      ? Abs( -700 )                    // result: 700
   RETURN NIL
```

AChoice()

Displays a list of items to select one from.

Syntax

```
AChoice( [<nTop>]       , ;
         [<nLeft>]      , ;
         [<nBottom>]    , ;
         [<nRight>]     , ;
         <acItems>      , ;
         [<aSelectable>], ;
         [<cUserFunc>]  , ;
         [<nFirstItem>] , ;
         [<nFirstRow>]    ) --> nSelectedItem
```

Arguments

<nTop> and <nLeft>

Numeric values indicating the screen coordinates for the upper left corner of the AChoice() window. The default value for both parameters is zero.

<nBottom>

Numeric value indicating the bottom row screen coordinate. It defaults to MaxRow().

<nRight>

Numeric value indicating the right column screen coordinate. It defaults to MaxCol().

<acItems>

A one dimensional array containing characters strings must be passed for *<acItems>*. It represents a list of items to choose one from. The character strings are displayed in the AChoice() window.

Note: No element in *<acItems>* may contain a null-string (""). A null-string is treated as End-of-list marker.

<aSelectable>

This parameter is optional. Either a logical value or an array containing logical values or character strings can be specified. It defaults to .T. (true) which means that all items in *<acItems>* can be selected. If .F. (false) is specified for *<aSelectable>*, the Ahoice() window is displayed but the function returns immediately, allowing no selection.

When an array is passed, it must have at least the same number of elements as *<acItems>* and is treated as parallel array indicating individual elements that can be selected. If an element of *<aSelectable>* contains .T. (true), the corresponding item in *<acItems>* is selectable, otherwise it is displayed but not selectable.

As an alternative, elements in *<aSelectable>* may contain character strings. They are treated as macro-expressions and are evaluated with the macro-operator. The result of the macro-expression must be a logical value.

<cUserFunc>

This is an optional character string holding the symbolic name of a user-defined function. The user function is called each time a key is pressed while AChoice() is active. The return value of the user function influences the AChoice() behavior. Note that the user function must be specified without parentheses, only the symbolic name may be passed.

`<nFirstItem>`

Optionally, the first item to start the selection with can be specified as a numeric value. It indicates the ordinal position of the element in *<acItems>* and defaults to 1, the first element.

`<nFirstRow>`

<nFirstRow> specifies the offset from the first window row to position the hilight bar in at initial display. It defaults to zero, i.e. the highlight bar is displayed in the *<nTop>* row when AChoice() begins.

Return

AChoice() returns a numeric value. It indicates the position of the selcted item in *<acItems>*. When the user makes no selection, the return value is zero.

Description

AChoice() is a console window function that presents to the user a list of items to select one from. The function maintains a window region and displays in it the items specified as the *<acItems>* array containing character strings. The colors for display are taken from the current SetColor() string. That is, normal items are displayed in Standard color, the highlight in Enhanced color and unselectable items with the Unselected color

The user can navigate through the list using cursor keys. AChoice() automatically scrolls the list within the defined window region if the user attempts to navigate the highlight bar outside the window. The user selects an item by pressing the Return key or by double clicking an item, and aborts selection with the Esc key.

Direct navigation to an item as accomplished by pressing letter keys. In this case, AChoice() moves the highlight to the next item whose first letter matches the pressed key.

Navigation without user function

AChoice() recognizes in its default navigation mode the keys listed in following table. The default mode is present when no user function *<cUserFunc>* is specified.

AChoice() default navigation

Key	Navigation
Up arrow	Go to previous item
Down arrow	Go to next item
Home	Go to first item in menu
End	Go to last item in menu
Ctrl+Home	Go to first item in window
Ctrl+End	Go to last item in window
PgUp	Go to previous page
PgDn	Go to next page
Ctrl+PgUp	Go to the first item in menu
Ctrl+PgDn	Go to last item in menu
Return	Select current item
Esc	Abort selection
Left arrow	Abort selection
Right arrow	Abort selection
First letter	Go to next item beginning with first letter
Left click *)	Go to clicked item
Left double click *)	Select clicked item

**) Mouse keys must be activated with SET EVENTMASK*

Navigation with user function

AChoice() processes some very basic navigation keys automatically when a user function is specified. This nvigation occurs before the user function is called. All keys not listed in the following table are passed on to the user function which may instruct AChoice() to take appropriate action.

AChoice() restricted navigation

Key	Navigation
Down arrow	Go to next item
Ctrl+Home	Go to first item in window
Ctrl+End	Go to last item in window
PgUp	Go to previous page
PgDn	Go to next page
Ctrl+PgUp	Go to the first item in menu
Ctrl+PgDn	Go to last item in menu
Left click *)	Go to clicked item

**) Mouse keys must be activated with SET EVENTMASK*

After default action is taken, AChoice() invokes the user function, passing three parameters for further processing:

1) The current AChoice() mode (see table below).
2) The current element in the array of items.
3) The relative row position of the highlight bar within the window region.

AChoice() modes passed to the user function indicate the internal state. They are encoded as numeric values for which #define constants exist in the ACHOICE.CH file.

AChoice() modes

Constant	Mode	Description
AC_IDLE	0	Idle
AC_HITTOP	1	Attempt to move cursor past top of list
AC_HITBOTTOM	2	Attempt to move cursor past bottom of list
AC_EXCEPT	3	Unknown key pressed
AC_NOITEM	4	No selectable items

The user function can process the passed parameters. It then instructs AChoice() how to proceed by returning one of the following numeric values. All user-function return values are also available as #define constants.

User-function return values for AChoice()

Constant	Value	Action
AC_ABORT	0	Abort selection
AC_SELECT	1	Make selection
AC_CONT	2	Continue AChoice()
AC_GOTO	3	Go to the next item whose first character matches the key pressed

Info

See also: @...PROMPT, DbEdit(), MENU TO, TBrowseNew()
Category: Array functions, UI functions
Header: achoice.ch
Source: rtl/achoice.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example uses AChoice() to display names of files in the current
// directory. Only files with the TXT or PRG extension can be selected.
// A selected file is displayed using MemoEdit(). Note how the return
// value of the user function triggers the AChoice() action.

   #include "Achoice.ch"
   #include "Inkey.ch"

   MEMVAR  aFiles

   PROCEDURE Main
      LOCAL nSelect
      LOCAL aItems  := Directory()
      LOCAL i, imax := Len( aItems )
      LOCAL aSelect := Array( imax )
      LOCAL cScreen

      SetColor( "W/N" )
      CLS
      SetColor( "N/BG,W+/B,,,W/BG" )

      PRIVATE aFiles := AClone( aItems )

      FOR i:=1 TO imax
         aItems[i]  := Upper( aItems[i,1] )
         aSelect[i] := ( ".TXT" $ aItems[i] .OR. ;
                         ".PRG" $ aItems[i] )
      NEXT

      DO WHILE LastKey() <> K_ESC
         nSelect := Achoice( 2, 2, MaxRow()-5, 30, ;
                             aItems, aSelect, "USERFUNC" )

         IF nSelect <> 0
            // Display selected file
            SAVE SCREEN TO cScreen
            MemoEdit( MemoRead( aItems[nSelect] ) )
            RESTORE SCREEN FROM cScreen
            KEYBOARD Chr(255)      // sets Lastkey() to 255
            Inkey()
         ENDIF

      ENDDO
   RETURN


   FUNCTION UserFunc( nMode, nElement, nRow )
      LOCAL nKey := LastKey()
      LOCAL nRet := AC_CONT
      LOCAL cMsg
```

```
    DO CASE
    CASE nMode == AC_IDLE
       // do some idle processing
       cMsg := " File: " +       aFiles[nElement,1]   + ;
               " Size: " +  Str( aFiles[nElement,2] ) + ;
               " Date: " + DtoC( aFiles[nElement,3] ) + ;
               " Time: " +       aFiles[nElement,4]

       DispOutAt( MaxRow(), 0, Padr(cMsg, MaxCol()+1), "W+/R" )

    CASE nMode == AC_EXCEPT
       // key handling for unknown keys
       IF nKey == K_ESC
          nRet := AC_ABORT

       ELSEIF nKey == K_RETURN .OR. nKey == K_LDBLCLK
          nRet := AC_SELECT

       ELSEIF nKey > 31 .AND. nKey < 255
          nRet := AC_GOTO

       ENDIF
    ENDCASE

 RETURN nRet
```

AClone()

Creates a deep copy of an array.

Syntax

```
AClone( <aArray> ) --> aClone
```

Arguments

<aArray>

A variable holding the array to duplicate.

Return

The function returns a reference to the created array.

Description

AClone() is an array function that duplicates an array. Unlike ACopy(), which operates only on one-dimensional arrays, AClone() performs a deep copy and iterates through all dimenstions of an array. If an array element contains an array, this sub-array is duplicated as well. As a result, all array references contained in one array are duplicated, resulting in a deep copy of the source array.

Note Array elements holding values of complex data types other than type Array are not duplicated. That is, Hashes, Code blocks and Objects are not duplicated. AClone() transfers only their reference to the result array. Source and result array then contain the exact same value of the complex data type.

Info

See also:	ACopy(), ADel(), AIns(), ASize()
Category:	Array functions
Source:	vm/arrayshb.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// This example demonstrates the difference between ACopy() and AClone()
//   -> aTgt1 is a shallow copy of the source array
//   -> aTgt2 is a deep copy of the source array

   PROCEDURE Main
      LOCAL aSrc  := { 1, { "A", "B" } }
      LOCAL aTgt1 := ACopy( aSrc, Array(2) )
      LOCAL aTgt2 := AClone( aSrc )

      // check first element of sub-array
      ? aTgt1[2,1]                     // result: A
      ? aTgt2[2,1]                     // result: A

      // change first element of sub-array in source array
      aSrc[2,1] := "no clone"

      // re-check first element of sub-array
      ? aTgt1[2,1]                     // result: no clone
      ? aTgt2[2,1]                     // result: A
   RETURN
```

ACopy()

Copy elements from a source array into a target array.

Syntax

```
ACopy( <aSource>      , ;
       <aTarget>      , ;
      [<nSourceStart>], ;
      [<nCount>]      , ;
      [<nTargetStart>]  ) --> aTarget
```

Arguments

<aSource>

The source array whose elements are copied.

<aTarget>

The target array recieving copied values.

<nSourceStart>

This is a numeric expression indicating the first element in the source array to begin copying with. It defaults to 1, the first element of *<aSource>*.

<nCount>

A numeric expression specifying the number of elements to copy from *<aSource>* to *<aTarget>*. It defaults to the maximum possible number that can be copied. This number depends on the size of source and target array, and the start elements of both arrays.

<nTargetPos>

A numeric expression indicating the first element in the target array that receives copied values. It defaults to 1.

Return

The function returns a reference to the *<aTarget>* array.

Description

ACopy() is an array function that iterates over the elements of a source array and copies values stored in these elements to a target array. Copying begins with the element *<nSourceStart>* of the source array. The value of this element is copied to the element *<nTargetStart>* of the target array. The function then proceeds to the next element in both arrays. The copy process is complete when either *<nCount>* elements are copied, or the last element in one of both arrays is reached. If the source array contains more elements than the target array, copying stops when the last element of the target array is reached, and vice versa. The number of elements in both arrays is not changed by ACopy().

The function operates in the first dimension of an array. Only values of the simple data types Character, Date, Logic, Numeric and NIL are actually copied. Values of the complex data types Array, Code block, Hash and Object are represented by references to their values. Hence, ACopy() copies the references to complex values from source to target array. As a result both arrays contain references to the same complex values. This is especially important when copying multi-dimensional array with ACopy() since both arrays contain references to the same sub-arrays when copying is complete.

Note: use function AClone() to obtain a true "deep copy" of multi-dimensional arrays.

Info

See also: AClone(), ADel(), AEval(), AFill(), AIns(), ASort()
Category: Array functions
Source: vm/arrayshb.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// This example copies the values from one array to another and
// displays the results on the screen.

   PROCEDURE Main()
      LOCAL aSource := {"xHarbour", "dot", "com"}
      LOCAL aTarget := {"xHarbour", "dot", "net"}

      ACopy( aSource, aTarget, 3, 1, 3)

      ? aSource[1], aSource[2], aSource[3]
      ? aTarget[1], aTarget[2], aTarget[3]

      AAdd( aSource, "is")
      AAdd( aSource, "great")

      ACopy(aSource, aTarget, , , 2)

      ? aSource[1], aSource[2], aSource[3], aSource[4], aSource[5]
      ? aTarget[1], aTarget[2], aTarget[3]
      ? Len( aTarget )
   RETURN
```

ACos()

Calculates the arc cosine.

Syntax

```
ACos( <nRadians> ) --> nArcCosine
```

Arguments

<nRadians>

A numeric value between -1 and +1 specifying the cosine in radians.

Return

The function returns the arc cosine as a numeric value between 0 and Pi().

Info

See also: ASin(), ATan(), ATn2(), DtoR(), RtoD(), SetPrec(), Sin(), Tan()
Category: CT:Math, Mathematical functions, Trigonometric functions
Source: ct\trig.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays results of function ACos().

   PROCEDURE Main
      ? Str( ACos( -1.0 ), 18, 15)  // result: 3.141592653589794
      ? Str( ACos( -0.5 ), 18, 15)  // result: 2.094395102393196
      ? Str( ACos(    0 ), 18, 15)  // result: 1.570796326794897
      ? Str( ACos(  0.5 ), 18, 15)  // result: 1.047197551196598
      ? Str( ACos(  1.0 ), 18, 15)  // result: 0.000000000000000
   RETURN
```

AddASCII()

Adds a numeric value to the ASCII code of a specified character in a string.

Syntax

```
AddASCII( <cString>, <nValue>, [<nPos>] ) --> cNewString
```

Arguments

<cString>

This is a character string to manipulate.

<nValue>

A numeric value to add to a single character of *<cString>*.

<nPos>

Optionally, the ordinal position of the character to change can be specified as a numeric value. It defaults to Len(cString), the last character.

Return

The function adds *<nValue>* to the ASCII code of the specified character and returns the modified string.

Info

See also:	CharAdd()
Category:	CT:String manipulation, Character functions
Source:	ct\addascii.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows results of AddASCII()

   PROCEDURE Main
      LOCAL cString := "ABC"

      ? AddAscii( cString, 32 )     // result: "ABc"

      ? AddAscii( cString, 3, 2 )   // result: "AEC"
   RETURN
```

AddMonth()

Adds or subtracts a number of months to/from a Date value.

Syntax

```
AddMonth( <dDate>, <nMonths> ) --> dNewDate
```

Arguments

<dDate>

A Date value, except for an empty date, can be passed.

<nMonths>

An integer numeric value specifying the number of months to add to *<dDate>*. If *<nMonths>* is a negative value, it is subtracted from *<dDate>*.

Return

The function returns the modified date, or an empty date on error.

Info

See also:	Date(), Month()
Category:	CT:DateTime, Date and time
Source:	ct\dattime2.c
LIB:	xhb.lib
DLL:	xhbdll.dll

ADel()

Deletes an element from an array.

Syntax

```
ADel( <aArray>, <nElement>, [<lShrink>] ) --> aArray
```

Arguments

`<aArray>`

A variable holding the array to delete an element from.

`<nElement>`

This is a numeric expression indicating the ordinal position of the array element to delete. It must be in the range between 1 and Len(<*aArray*>).

`<lShrink>`

Optionally, a logical value can be specified. If .T. (true) is passed, the length of the array is reduced by one element. The default value is .F. (false) leaving the number of elements in <*aArray*> unchanged.

Return

The return value is a reference to <*aArray*>.

Description

The array function ADel() deletes the element at position <*nElement*> from the array <*aArray*>. All subsequent elements are shifted up by one position so that the last element contains the value NIL when the function returns. This default behaviour leaves the number of elements in the array unchanged.

If the third parameter <*lShrink*> is specified as .T. (true), the last element is removed from the array and the number of elements is reduced by 1.

Info

See also:	AAdd(), ACopy(), AFill(), AIns(), ASize()
Category:	Array functions
Source:	vm/arrayshb.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// This example demonstrates two results of deleting an
// array element

   PROCEDURE Main()
      LOCAL aArray := { "A", "B", "C" }

      ADel( aArray, 2 )

      ? Len( aArray )                      // result: 3
      ? aArray[1], aArray[2], aArray[3]   // result: A C NIL

      ADel( aArray, 2, .T. )

      ? Len( aArray )                      // result: 2
```

```
   ? aArray[1], aArray[2]           // result: A NIL

RETURN
```

ADir()

Fills pre-allocated arrays with file and/or directory information.

Syntax

```
ADir( [<cFileMask>], ;
      [<aName>]    , ;
      [<aSize>]    , ;
      [<aDate>]    , ;
      [<aTime>]    , ;
      [<aAttr>]      ) --> nDirEnries
```

Arguments

<cFileMask>

<cFileMask> is a character string specifying the directory and file mask used to search for files. The file mask may contain wild card characters, like "*" or "?". If no path is givven, ADir() searches in the SET DEFAULT directory.

<aName>

A pre-allocated array to fill with the file names of the found files. Each element contains a character string holding a file name without path information, as reported by the operating system.

<aSize>

A pre-allocated array to fill with numeric values indicating the file size.

<aDate>

A pre-allocated array to fill with the "last change" date values.

<aTime>

A pre-allocated array to fill with the "last change" Time values. The values are of data type Character and contain the time formatted as "hh:mm:ss".

<aAttr>

A pre-allocated array to fill with file attributes. Recognized attributes are Normal, Hidden, System and Directory. The are combined in one character string. If *<aAttr>* is omitted, ADir() includes only normal files in the search result.

Return

The function returns a numeric value indicating the number of files meeting the search condidtion defined with *<cFileMask>*.

Description

ADir() searches a directory for files meeting a search pattern *<cFileMask>*, and optionally fills arrays with various file information. The number of files meeting the search condition is usually determined in an initial call to ADir(). The return value of the initial call is then used to pre-allocate one-dimensional arrays that are filled with the desired informationin a second call to ADir().

To include files with Hidden, Directory, or System attributes, the according attribute must be specified for the search.

Note: ADir() is a compatibility function. It is superseded by the Directory() function which returns all file information in a multi-dimensional array.

Info

See also: AChoice(), AEval(), AScan(), ASort(), Directory(), Len()
Category: Array functions, File functions
Source: rtl\adir.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// This example demonstrates the two-step approach for obtaining file
// information about .LOG files. The initial call determines the number
// of files found. This result is used to pre-allocate arrays to fill
// in a second call to ADir().

   PROCEDURE MAIN()
      LOCAL aName, aSize, aDate, aTime, aAttr
      LOCAL nLen, i

      nLen := ADir( "*.log" )

      IF nLen > 0
         aName := Array( nLen )
         aSize := Array( nLen )
         aDate := Array( nLen )
         aTime := Array( nLen )
         aAttr := Array( nLen )

         ADir( "*.log", aName, aSize, aDate, aTime, aAttr )

         FOR i := 1 TO nLen
           ? aName[i], aSize[i], aDate[i], aTime[i], aAttr[i]
         NEXT
      ELSE
         ? "This directory does not contain .log files."
      ENDIF

   RETURN
```

AEval()

Evaluates a code block for each array element.

Syntax

```
AEval( <aArray>, <bBlock>, [<nStart>], [<nCount>] ) --> aArray
```

Arguments

<aArray>

This is the array to iterate.

<bBlock>

The code block evaluated for the elements of *<aArray>*. It receives two parameters: the value of the current array element and its numeric position in the array.

<nStart>

This is a numeric expression indicating the first element in the array to begin with. It defaults to 1, the first element of *<aArray>*.

<nCount>

A numeric expression specifying the number of elements to pass to the code block. It defaults to 1+Len(*<aArray>*)-*<nStart>*.

Return

The function returns a reference to *<aArray>*.

Description

The array function AEval() traverses an array beginning with the element *<nStart>* and passes the values of individual array elements to the code block *<bBlock>*. The code block is evaluated and AEval() proceeds to the next array element until either *<nCount>* elements are processed or the end of the array is reached. The return value of the code block is ignored.

With each iteration, the code block receives as parameters the value stored in the current array element and a numeric value indicating the ordinal position of the current array element.

Note that AEval() is appropriate for iterating an array once. Operations that require a more sophisticated iteration or that alter array elements are programmed in FOR...NEXT or FOR EACH loops.

Info

See also:	DbEval(), Eval(), FOR, FOR EACH, HEval(), QOut() | QQOut()
Category:	Array functions, Code block functions
Source:	vm\arrayshb.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The first part of the example displays the contents of an array while the
// second part first increases the value of the elements and then displays it
// on the screen.

   PROCEDURE Main()
      LOCAL aArray := {"A", "B", "C", "D", "E", "F" }

      ? "Simple iteration: "
```

```
    AEval( aArray, {|x| QQOut(x) } )        // result: ABCDEF

    ? "Fill array with consecutive numbers: "
    AEval( aArray, {|x,n| aArray[n] := n } )

    // Note: LOCAL variable is visible within code block
    // since aArray and code block are created in the
    // same procedure.

    AEval( aArray, {|x| QQOut(LTrim(Str(x))) } )
                                            // result: 123456
RETURN
```

AFields()

Fills pre-allocated arrays with structure information of the current work area.

Syntax

```
AFields( [<aFieldName>], ;
         [<aFieldType>], ;
         [<aFieldLen>] , ;
         [<aFieldDec>]   ) --> nCount
```

Arguments

<aFieldNames>

A pre-allocated array to fill with the field names of the work area. Each element contains a character string.

<aFieldTypes>

A pre-allocated array to fill with the field types of the work area. Each element contains a single character.

<aFieldLen>

A pre-allocated array to fill with the field lengths of the work area. Each element contains a numeric value.

<aFieldDec>

A pre-allocated array to fill with the decimal places of the fields of the work area. Each element contains a numeric value.

Return

The function returns a numeric value indicating the number of array elements filled. If no parameter is passed or if the work area is not used, the return value is zero.

Description

AField() is used to obtain structural information about a database open in a work area. The function operates in the current work area unless it is prefixed with the alias operator.

One or more arrays must be pre-allocated to receive the desired information. Use function FCount() to dimension the arrays large enough to hold all avialable information.

Note that AFields exists for compatibility reasons. It is superseded by the DbSTruct() function, which retrieves all structural information in a multi-dimensional array.

Info

See also:	AChoice(), AEval(), AScan(), DbCreate(), DbStruct()
Category:	Array functions, Field functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example fills an array with field names and displays the
// names and field contents within AEval().

   PROCEDURE Main
      LOCAL aNames
```

```
    USE Customer
    aNames := Array( FCount() )

    AFields( aNames )
    AEval( aNames, {|cName| QOut(cName, &cName) } )

    CLOSE ALL
RETURN
```

AFill()

Fills an array with a constant value.

Syntax

```
AFill( <aArray>, <expression>, [<nStart>], [<nCount>] ) --> aArray
```

Arguments

`<aArray>`

This is the variable holding the array to fill with a value.

`<expression>`

An expression yielding a value of arbitrary data type. The expression is evaluated once and the resulting value is assigned to each element of *<aArray>*.

`<nStart>`

This is a numeric expression indicating the first element in the array to fill. It defaults to 1.

`<nCount>`

A numeric expression specifying the number of elements to fill. It defaults to 1+Len(*<aArray>*)-*<nStart>*.

Return

The function returns a reference to *<aArray>*.

Description

The array function AFill() traverses the passed array and assigns the value of *<expression>* to the specified number of elements, beginning at position *<nStart>*. The array is only traversed in its first dimension. Milti-dimensional arrays cannot be filled using AFill().

Note: *<expression>* is evaluated only once, i.e. all elements are assigned the same value. If the value is of complex data type that is a referenceto to a value (e.g. Array or Object), all elements contain the same reference to the same value.

Info

See also:	AAdd(), AEval(), AIns(), DbStruct(), Directory()
Category:	Array functions
Source:	vm/arrayshb.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example creates an array filled with "0" and  demonstrates the
// effect of using a complex data type (Array) to fill an array with.

   PROCEDURE Main()
      LOCAL aArray[3]

      // using a simple data type
      AFill( aArray, "A" )
      ? aArray[1], aArray[2], aArray[3]  // result: A A A

      // using a complex data type
      AFill( aArray, { "A", "B" } )
```

```
   ? aArray[1,1], aArray[1,2]         // result: A B
   ? aArray[2,1], aArray[2,2]         // result: A B
   ? aArray[3,1], aArray[3,2]         // result: A B

   // changing one element
   aArray[3,2] := "DON'T"

   ? aArray[1,1], aArray[1,2]         // result: A DON'T
   ? aArray[2,1], aArray[2,2]         // result: A DON'T
   ? aArray[3,1], aArray[3,2]         // result: A DON'T

   // One assignment changes all elements
   // since aArray contains three times
   // the same array reference.
RETURN
```

AfterAtNum()

Extracts the remainder of a string after the last occurrence of a search string.

Syntax

```
AfterAtNum( <cSearch>    , ;
            <cString>    , ;
           [<nCount>]    , ;
           [<nSkipChars>] ) --> cResult
```

Arguments

<cSearch>

This is the character string to search for in <*cString*>.

<cString>

This is the character string to find <*cSearch*> in.

<nCount>

A numeric value indicating the number of occurrences of finding <*cSearch*> in <*cString*> before the function returns. It defaults to the last occurrence.

<nSkipChars>

This numeric parameter defaults to 0 so that the function begins searching with the first character of <*cString*>. Passing a value > 0 instructs the function to ignore the first <*nSkipChars*> characters in the search.

Return

The function returns the remainder of <*cString*> after the <*nCount*>th occurrence of <*cSearch*> in the string.

Description

AfterAtNum() begins searching <*cSearch*> in <*cString*> at the first plus <*nSkipChars*> characters. It terminates searching after <*nCount*> occurrences or after the last occurrence of <*cSearch*> and returns the remainder of <*cString*>. The function recognizes the CSetAtMuPa() and SetAlike() settings.

Info

See also: AtNum(), AtRepl(), BeforAtNum(), NumAt(), CSetAtMuPa(), SetAtLike()
Category: CT:String manipulation, Character functions
Source: ct\atnum.c
LIB: xhb.lib
DLL: xhbdll.dll

AIns()

Inserts an element into an array.

Syntax

```
AIns( <aArray>, <nElement>, [<xValue>], [<lGrow>] ) --> aArray
```

Arguments

<aArray>

A variable holding the array to insert an element into.

<nElement>

This is a numeric expression indicating the ordinal position where to insert the new array element. It must be in the range between 1 and Len(*<aArray>*).

<xValue>

A value of arbitrary data type to assign to the new array element. It defaults to NIL.

<lGrow>

Optionally, a logical value can be specified. If .T. (true) is passed, the length of the array is enlarged by one element. The default value is .F. (false) leaving the number of elements in *<aArray>* unchanged.

Return

The function returns a reference to *<aArray>*.

Description

The array function AIns() inserts a new element at position *<nElement>* into the array *<aArray>*. All subsequent elements are shifted down by one position so that the last element is lost when the function returns. This default behaviour leaves the number of elements in the array unchanged.

The new array element is initialized with NIL, but can be assigned a value when parameter *<xValue>* is not NIL.

If the fourth parameter *<lGrow>* is specified as .T. (true), the function first adds a new element and then shifts all elements down, beginning at the specified position. This way, the last element is preserved and the number of elements is enlarged by one.

Info

See also:	AAdd(), ACopy(), ADel(), AEval(), AFill(), ASize()
Category:	Array functions
Source:	vm\arrayshb.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// This example demonstrates two results of inserting an
// array element

   PROCEDURE Main()
      LOCAL aArray := { "A", "B", "C" }

      AIns( aArray, 2 )
```

```
   ? Len( aArray )                     // result: 3
   ? aArray[1], aArray[2], aArray[3]  // result: A NIL B

   AIns( aArray, 2, "C", .T. )

   ? Len( aArray )                     // result: 4
   ? aArray[1], aArray[2], aArray[3], aArray[4]
                                       // result: A C NIL B
RETURN
```

ALenAlloc()

Determines for how much array elements memory is pre-allocated.

Syntax

```
ALenAlloc( <aArray> ) --> nElements
```

Arguments

`<aArray>`

This is the array to query pre-allocated memory for.

Return

The function returns a numeric value specifying the number of elements memory is pre-allocated for.

Description

The function is used to determine for how much array elements memory is pre-allocated. This is accomplished with function ASizeAlloc() which is available for memory optimization of dynamically growing arrays.

Info

See also:	AAdd(), Array(), ASizeAlloc()
Category:	Array functions, xHarbour extensions
Source:	vm\arrayshb.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Alert()

Displays a text-mode dialog box with a message.

Syntax

```
Alert( <xMessage>, [<aOptions>], [<cColor>], [<nDelay>] ) --> nChoice
```

Arguments

<xMessage>

This is the message to display in the dialog box. It can be of any data type but is usually a character string or an array filled with strings. If *<xMessage>* is a character string, the semicolon is treated as a new line character. When it is an array, each element is displayed in a new line of the dialog box.

<aOptions>

<aOptions> is an array filled with character strings. Each element is displayed as prompt for the user to make a selection. If no parameter is passed, *<aOptions>* defaults to { "Ok" }.

<cColor>

This is a color string holding two color values. The Normal color is used to display the dialog box along with *<xMessage>*. The options to select from are displayed in the Enhanced color. The default is "W+/R,W+/B".

<nDelay>

Optionally, a numeric value can be specified, indicating the number of seconds to wait for a key press before the dialog box closes automatically. It defaults to zero, which means that the dialog box does not close automatically.

Return

The function returns a numeric value indicating the ordinal position within *<aOption>* of the prompt selected by the user. When the ESC key is pressed, the user aborted the selection and the return value is 0.

If *<nDelay>* is specified and this number of seconds has elapsed without a key press, the return value is 1.

Description

Alert() displays a simple text-mode dialog box and lets the user select an option. The user can move a highlight bar using arrow keys or the TAB key. To select an option, the user can press ENTER, SPACE or the first letter of an option.

Info

See also:	@...PROMPT, AChoice(), DispBox(), DispOut(), MENU TO
Category:	UI functions
Source:	rtl\alert.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays a message to terminate or resume the program.
// After 5 seconds without user response, the program quits automatically.

   PROCEDURE Main()
      LOCAL cMessage, aOptions, nChoice
```

```
    cMessage := "Quit program?;(auto-quit in 5 seconds)"
    aOptions := { "Yes", "No" }

    nChoice := Alert( cMessage, aOptions, , 5)

    DO CASE
    CASE nChoice == 0
       ? "Program not interrupted ..."
    CASE nChoice == 1
       QUIT
    CASE nChoice == 2
       ? "Program resumed ..."
    ENDCASE

  RETURN
```

Alias()

Returns the alias name of a work area.

Syntax

```
Alias( [<nWorkArea>] ) --> cAlias
```

Arguments

```
<nWorkArea>
```

<nWorkArea> is the numeric ordinal position of a work area. Work areas are numbered from 1 to 65535.

Return

The function returns the alias name of the work area specified with *<nWorkArea>*. If no parameter is passed, the alias name of the current work area is returned. When no database is open in the work area, the return value is a null string ("").

Description

The alias name of a work area exists while a database is open in it. Alias names are used to address work areas via a symbolic name and are specified with the USE command or the DbUseArea() function. Alias names are always created in upper case letters.

Info

See also:	DbInfo(), DbUseArea(), OrdName(), OrdNumber(), SELECT, Select(), USE
Category:	Database functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates usage of functions Alias() and Select(),
// both ofwhich are used to address work areas.

   PROCEDURE Main
      USE Customer NEW ALIAS Cust
      USE Invoice  NEW ALIAS Inv

      ? Alias()                          // result: INV
      ? Select()                         // result: 2

      ? Alias(1)                         // result: CUST
      ? Select( "CUST" )                 // result: 1

      CLOSE ALL
   RETURN
```

AllTrim()

Removes leading and trailing blank spaces from a string.

Syntax

```
AllTrim( <cString> ) --> cTrimmed
```

Arguments

`<cString>`

<*cString*> is any character string.

Return

The function returns the passed string without leading and trailing blank spaces.

Description

This function removes leading and trailing spaces from a string. Use function LTrim() or RTrim() to remove spaces on the left or right end of <*cString*>.

Info

See also:	LTrim(), PadC() | PadL() | PadR(), RTrim()
Category:	Character functions
Source:	rtl\trim.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// This example removes all leading and trailing spaces from strings.

   PROCEDURE Main()

      ? "<" + AllTrim( "  xHarbour " ) + ">"  // result: <xHarbour>

      ? "<" + AllTrim( "xHarbour " ) + ">"    // result: <xHarbour>

      ? "<" + AllTrim( "  xHarbour" ) + ">"   // result: <xHarbour>

   RETURN NIL
```

AltD()

Activates and/or invokes the "classic" debugger.

Syntax

```
AltD( [<nAction>] ) --> NIL
```

Arguments

<nAction>

If no parameter is passed, the debugger is invoked, when it is activated. Passing numeric 1 activates the debugger and numeric zero deactivates it.

Return

The return value is always NIL.

Description

Function AltD() toggles the activity of the "classic" debugger which runs in text mode only. To activate the debugger, AltD(1) must be called in a program. A subsequent call to AltD() with no parameter invokes the debugger and displays the PRG source code file with the current line being executed. A programmer can then inspect variables or execute a program step by step. Calling AltD(0) disables the debugger.

When the debugger is activated, the key combination Alt+D invokes the debugger as well. This way, a program can run until a certain situation is reached that must be debugged.

Info

See also:	Set()
Category:	Debug functions
Source:	debug\dbgaltd.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates how to invoke the "classic" debugger.
// To run the example, create the executable with "classic" debug
// information or call xbuild on the command line:
//
// --> xbuild -B -CLASSIC test.prg

   PROCEDURE Main
      LOCAL dDate := Date()
      LOCAL nKey

      AltD(1)                // activates debugger
      AltD()                 // invokes debugger unconditionally

      ? "Today is:", dDate

      IF Day( dDate ) == 1
         AltD()              // invokes debugger conditionally
      ENDIF

      nKey := Inkey(0)       // debugger is invoked by pressing Alt+D
```

```
    ? "Last key:", nKey
RETURN
```

AmPm()

Converts a time string into am/pm format.

Syntax

```
AmPm( <cTime> ) --> cFormattedTime
```

Arguments

`<cTime>`

<*cTime*> is a character string returned by the Time() function.

Return

The return value is the formatted time string.

Description

AmPm() is a utility function that converts the 24h time string returned by the Time() function to a 12h am/pm time string.

Info

See also:	Time(), Transform()
Category:	Conversion functions
Source:	rtl\ampm.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates formatting of Time() string

   PROCEDURE Main
      LOCAL cMidnight := "00:00:00"
      LOCAL cMorning  := "07:30:45"
      LOCAL cNoon     := "12:00:00"
      LOCAL cEvening  := "19:30:45"

      ? AmPm( cMidNight )              // result: 12:00:00 am
      ? AmPm( cMorning  )              // result: 07:30:45 am
      ? AmPm( cNoon     )              // result: 12:00:00 pm
      ? AmPm( cEvening  )              // result:  7:30:45 pm
   RETURN
```

AnsiToHtml()

Inserts HTML character entities into an ANSI text string,

Syntax

```
AnsiToHtml( <cAnsiString> ) --> cHtmlString
```

Arguments

<cAnsiString>

This is an ANSI character string.

Return

The function returns a HTML formatted text string.

Description

Function AnsiToHtml() inserts HTML character entities into *<cAnsiString>* and returns the HTML formatted text string. Note that the function processes only HTML character entities (e.g. >, < or &) that are interchangeable in the OEM and ANSI character sets.

The following table lists all HTML character entities recognized by the function.

HTML character entities.

Character	HTML entity	Description
&	&	ampersand
<	<	less-than sign
>	>	greater-than sign
¢	¢	cent sign
£	£	pound sign
¥	¥	yen sign
¦	¦	broken bar
§	§	section sign
©	©	copyright sign
®	®	registered sign
°	°	degree sign
¿	¿	inverted question mark
À	À	Latin capital letter a with grave
Á	Á	Latin capital letter a with acute
Â	Â	Latin capital letter a with circumflex
Ã	Ã	Latin capital letter a with tilde
-	Ä	Latin capital letter a with diaeresis
Å	Å	Latin capital letter a with ring above
Æ	Æ	Latin capital letter ae
Ç	Ç	Latin capital letter c with cedilla
È	È	Latin capital letter e with grave
É	É	Latin capital letter e with acute
Ê	Ê	Latin capital letter e with circumflex
Ë	Ë	Latin capital letter e with diaeresis
Ì	Ì	Latin capital letter i with grave
Í	Í	Latin capital letter i with acute
Î	Î	Latin capital letter i with circumflex
Ï	Ï	Latin capital letter i with diaeresis
Ð	Ð	Latin capital letter eth
Ñ	Ñ	Latin capital letter n with tilde
Ò	Ò	Latin capital letter o with grave

Ó	Ó	Latin capital letter o with acute
Ô	Ô	Latin capital letter o with circumflex
Õ	Õ	Latin capital letter o with tilde
Ö	Ö	Latin capital letter o with diaeresis
Ø	Ø	Latin capital letter o with stroke
Ù	Ù	Latin capital letter u with grave
Ú	Ú	Latin capital letter u with acute
Û	Û	Latin capital letter u with circumflex
Ü	Ü	Latin capital letter u with diaeresis
Ý	Ý	Latin capital letter y with acute
Þ	Þ	Latin capital letter thorn
ß	ß	Latin small letter sharp s (German Eszett)
à	à	Latin small letter a with grave
á	á	Latin small letter a with acute
â	â	Latin small letter a with circumflex
ã	ã	Latin small letter a with tilde
ä	ä	Latin small letter a with diaeresis
å	å	Latin small letter a with ring above
æ	æ	Latin lowercase ligature ae
ç	ç	Latin small letter c with cedilla
è	è	Latin small letter e with grave
é	é	Latin small letter e with acute
ê	ê	Latin small letter e with circumflex
ë	ë	Latin small letter e with diaeresis
ì	ì	Latin small letter i with grave
í	í	Latin small letter i with acute
î	î	Latin small letter i with circumflex
ï	ï	Latin small letter i with diaeresis
ð	ð	Latin small letter eth
ñ	ñ	Latin small letter n with tilde
ò	ò	Latin small letter o with grave
ó	ó	Latin small letter o with acute
ô	ô	Latin small letter o with circumflex
õ	õ	Latin small letter o with tilde
ö	ö	Latin small letter o with diaeresis
ø	ø	Latin small letter o with stroke
ù	ù	Latin small letter u with grave
ú	ú	Latin small letter u with acute
û	û	Latin small letter u with circumflex
ü	ü	Latin small letter u with diaeresis
ý	ý	Latin small letter y with acute
þ	þ	Latin small letter thorn
ÿ	ÿ	Latin small letter y with diaeresis
^	ˆ	modifier letter circumflex accent
~	˜	small tilde

Info

See also: HtmlToAnsi(), HtmlToOem(), OemToHtml(), THtmlDocument()
Category: Conversion functions, HTML functions, xHarbour extensions
Source: tip\thtml.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example shows a result of AnsiToHtml()

   PROCEDURE Main
      LOCAL cText := 'Copyright © 2007 xHarbour Inc'

      ? AnsiToHtml( cText )  // result: Copyright &copy; 2007 xHarbour Inc

   RETURN
```

Array()

Creates an uninitialized array of specified length.

Syntax

```
Array( <nDim1> [, <nDimN,...>] ) --> aArray
```

Arguments

`<nDim>`

A comma separated list of numeric values can be passed. Each value indicates the number of elements to create for each dimension of the array. Passing a single numeric value creates a one-dimensional array.

Return

The function returns an array dimensioned according to the passed parameters. All elements of the array contain the value NIL.

Description

Array variables are usually initialized within a variable declaration. The Array() function allows for creating arrays programmatically outside a variable declaration. Array() can be part of an expression or can be called from within a code block.

The return value of Array() is a reference to an array of the specified dimensions. The number of elements in the array is limited by memory resources of the operating system.

Info

See also:	AAdd(), AClone(), ACopy(), ADel(), AEval(), AFill(), AIns(), Hash()
Category:	Array functions
Source:	vm\arrayshb.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates different approaches of creating
// uninitialized arrays.

   PROCEDURE Main()
     LOCAL xValue
     LOCAL aArray[2,3]                  // declaration

     xValue := Array( 3, 5 )            // programmatically

     ? Len( aArray ), Len( aArray[1] )  // result: 2 3

     ? Len( xValue ), Len( xValue[1] )  // result: 3 5
   RETURN NIL
```

Asc()

Returns the ASCII code for characters.

Syntax

```
ASC( <cCharacter> ) --> nAsciiCode
```

Arguments

`<cCharacter>`

<cCharacter> is any character expression.

Return

The functions returns the numeric ASCII code for *<cCharacter>*.

Description

This function returns the ASCII value of the leftmost character of any character expression.

Note: passing a null string ("") yields 0. It is the same result as passing Chr(0). A null string, however, is different from Chr(0).

Info

See also:	Chr(), Inkey(), Str(), Val()
Category:	Conversion functions
Source:	rtl\chrasc.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays the ASCII values of several characters and
// character strings

   PROCEDURE Main()

      ? Asc( " " )                    // result:  32
      ? Asc( "A" )                    // result:  65
      ? Asc( "a" )                    // result:  97
      ? Asc( "xHarbour" )             // result: 120
      ? Asc( "XHARBOUR" )             // result:  88

   RETURN
```

AScan()

Searches a value in an array beginning with the first element.

Syntax

```
Ascan( <aArray>  , ;
       <xbSearch>, ;
       [<nStart>], ;
       [<nCount>], ;
       [<lExact>], ;
       [<lASCII>]  ) --> nElement
```

Arguments

`<aArray>`

This is the array to iterate.

`<xbSearch>`

<xbSearch> is either the value to search in *<aArray>* or a code block containing the search condition. The code block receives a single parameter which is the value stored in the current array element. The code block must return a logical value. When the return value is .T. (true), the function stops searching and returns the position of the corresponding array element.

`<nStart>`

This is a numeric expression indicating the first element in the array to begin the search with. It defaults to 1, the first element of *<aArray>*.

`<nCount>`

A numeric expression specifying the number of elements to search. It defaults to 1+Len(*<aArray>*)-*<nStart>*.

`<lExact>`

This parameter influences the comparison for searching character strings in *<aArray>*. If .T. (true) is passed, an exact string comparison is performed. When omitted or if .F. (false) is passed, string comparison follows the SET EXACT rules.

`<lASCII>`

This parameter is only relevant for arrays that contain single characters in their elements. A single character is treated like a numeric ASCII value when *<lASCII>* is .T. (true). The default is.F. (false).

Return

AScan() returns the numeric ordinal position of the array element that contains the searched value. When no match is found, the return value is 0.

Description

The array function AScan() traverses the array *<aArray>* for the value specified with *<xbSearch>*. If *<xbSearch>* is not a code block, AScan() compares the values of each array element for being equal with the searched value. If this comparison yields .T. (true), the function stops searching and returns the numeric position of the array element containing the searched value.

If a code block is passed for *<xbSearch>*, it is evaluated and receives as parameter the value of the current array element. The code block must contain a comparison rule that yields a logical value. AScan() considers the value as being found when the codeblock returns .T. (true). Otherwise the function proceeds with the next array element.

Note: use function RAscan() to search an array beginning with the last element.

Info

See also: AEval(), Asc(), ASort(), ATail(), Eval(), IN, RAscan(), SET EXACT
Category: Array functions
Source: vm\arrayshb.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates simple and complex searching in arrays.

   PROCEDURE Main()
      LOCAL aArray := { "A", 1, Date(), NIL, .F. }

      // one dimensional array
      ? AScan( aArray, 1 )              // result: 2
      ? AScan( aArray, .F. )            // result: 5

      ? AScan( aArray, {|x| Valtype(x)=="D" } )
                                        // result: 3

      // two dimensional array
      aArray := Directory( "*.prg" )

      ? AScan( aArray, {|a| Upper(a[1]) == "TEST.PRG" } )
                                        // result: 28
   RETURN
```

ASCIISum()

Sums the ASCII codes of all characters in a string.

Syntax

```
ASCIISum( <cString> ) --> nASCIISum
```

Arguments

`<cString>`

This is the character string to process.

Return

The function returns the sum of the ASCII codes of all characters of <*cString*>.

Info

See also: Asc(), CheckSum()
Category: CT:String manipulation, Character functions
Source: ct\asciisum.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates the return value of AsciiSum().

   PROCEDURE Main
      LOCAL cString := "ABC"
      LOCAL cChar

      FOR EACH cChar in cString
         ? Asc( cChar )
      NEXT                   ** Result:
                             //         65
                             //         66
                             //         67

      ? AsciiSum( cString )  // result: 198
   RETURN
```

AscPos()

Determines the ASCII code of a specified character in a string.

Syntax

```
AscPos( <cString>, [<nPos>] ) --> nASCIICode
```

Arguments

<cString>

This is a character string to query for an ASCII code.

<nPos>

Optionally, the ordinal position of the character to query can be specified as a numeric value. It defaults to Len(cString), the last character.

Return

The function returns the ASCII code of the character at the specified position as a numeric value.

Note: the function exists for compatibility reasons. The [ ] operator can be applied to character strings in xHarbour and is more efficient for extracting a single character.

Info

See also:	[ ] (string), Asc()
Category:	CT:String manipulation, Character functions
Source:	ct\ascpos.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows two different ways of obtaining the
// ASCII code of a single character in a string.

   PROCEDURE Main
      LOCAL cString := "xHarbour"

      // compatibility
      ? AscPos( cString )          // result: 114
      ? AscPos( cString, 2 )       // result:  72

      // xHarbour [] operator
      ? Asc( cString[-1] )         // result: 114
      ? Asc( cString[2] )          // result:  72
   RETURN
```

ASin()

Calculates the arc sine.

Syntax

```
ASin( <nRadians> ) --> nArcSine
```

Arguments

`<nRadians>`

A numeric value between -1 and +1 specifying the sine in radians.

Return

The function returns the arc sine as a numeric value between plus/minus Pi()/2.

Info

See also: ACos(), ATan(), ATn2(), DtoR(), RtoD(), SetPrec(), Sin(), Tan()
Category: CT:Math, Mathematical functions, Trigonometric functions
Source: ct\trig.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays results of function ASin().

   PROCEDURE Main
      ? Str( ASin( -1.0 ), 18, 15)  // result: -1.570796326794897
      ? Str( ASin( -0.5 ), 18, 15)  // result: -0.523598775598299
      ? Str( ASin(    0 ), 18, 15)  // result:  0.000000000000000
      ? Str( ASin(  0.5 ), 18, 15)  // result:  0.523598775598299
      ? Str( ASin(  1.0 ), 18, 15)  // result:  1.570796326794897
   RETURN
```

ASize()

Changes the number of elements in an array.

Syntax

```
ASize( <aArray>, <nCount> ) --> aArray
```

Arguments

`<aArray>`

This is the array whose size is changed.

`<nCount>`

A numeric expression specifying the number of elements the result array should have.

Return

The function returns the reference to *<aArray>*.

Description

The array function ASize() is used to dynamically increase or decrease the size of *<aArray>* to *<nCount>* elements. When the array is enlarged, new elements are added to the end of *<aArray>* and are initialized with NIL. When the number of elements is decreased, they are removed from the end of *<aArray>*. The values in the removed elements are lost.

The size of arrays is limited by the memory resources of the operating system.

Info

See also:	AAdd(), ADel(), AFill(), AIns(), ASizeAlloc()
Category:	Array functions
Source:	vm\arrayshb.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// This example demonstrates the use of the ASize function.
// It increases and decreases the size of an array.

   PROCEDURE Main()
      LOCAL aArray := { 45 }

      ? Len( aArray )                      // result: 1

      ASize( aArray, 3 )
      ? Len( aArray )                      // result: 3
      ? aArray[1], aArray[2], aArray[3]   // result: 45 NIL NIL

      ASize( aArray, 1 )
      ? Len( aArray )                      // result: 1
      ? aArray[1]                          // result: 45

   RETURN
```

ASizeAlloc()

Pre-allocates memory for an array.

Syntax

```
ASizeAlloc( <aArray>, <nCount>  ) --> aArray
```

Arguments

`<aArray>`

This is the array to pre-allocate memory for.

`<nCount>`

A numeric value specifying the number of elements to reserve memory for.

Return

The function returns the array <*aArray*>.

Description

Function ASizeAlloc() pre-allocates memory for <*nCount*> number of elements of an array. This improves dynamically growing arrays when elements are added to the array using AAdd().

Note: the function does not change the number of elements of <*aArray*>, it reserves only memory for them, thus optimizing memory usage with dynamic arrays.

Info

See also:	AAdd(), Array(), ALenAlloc(), ASize()
Category:	Array functions, xHarbour extensions
Source:	vm\arrayshb.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example creates an empty array and pre-allocates
// memory for 1000 elements. Note that the examples ends
// with a runtime error, since element 100 does not exist.

   PROCEDURE Main
      LOCAL aArray := {}

      ASizeAlloc( aArray, 1000 )

      ? ALenAlloc( aArray )        // result: 1000

      AAdd( aArray, 10 )

      ? Len( aArray )              // result: 1

      ? aArray[100]                // runtime error
  RETURN
```

ASort()

Sorts an array.

Syntax

```
ASort( <aArray>, [<nStart>], [<nCount>], [<bSort>] ) --> aArray
```

Arguments

`<aArray>`

The array to be sorted.

`<nStart>`

This is a numeric expression indicating the first element in the array to begin sorting with. It defaults to 1, the first element of *<aArray>*.

`<nCount>`

A numeric expression specifying the number of elements to sort. It defaults to 1+Len(*<aArray>*)-*<nStart>*.

`<bSort>`

Optionally, a code block defining a sorting rule can be passed. If this parameter is omitted, all values stored in *<aArray>* are sorted in ascending order.

When a code block is specified, it must be declared with two parameters and must return a logical value. The code block receives the values of two adjacent array elements. When the code block returns .T. (true), the first code block parameter is considered smaller than the second. If .F. (false) is returned, the first code block parameter is larger.

Return

The function returns a reference to *<aArray>*.

Description

ASort() sorts an array entirely or partially by the values stored in its elements. If the code block *<bSort>* is omitted, the function expects *<aArray>* to be a one dimensional array containing simple data types. The values are sorted in ascending order. Character strings are sorted by their ASCII values, dates are sorted chronologically, Logical .F. (false) is smaller than .T. (true), and Numerics are sorted by their value.

If *<bSort>* is specified, it is used to define the comparison rule for "smaller". The code block must have two parameters which receive the values of two array elements. The first code block parameter is considered smaller, when the code block returns .T. (true), otherwise it is considered larger than the second code block parameter.

The definition of an appropriate comparison rule allows for sorting an array in descending order, and/or to sort multi-dimensional arrays.

Info

See also: AEval(), AScan(), Eval(), SORT
Category: Array functions
Source: vm\arrayshb.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates the sorting of one- and two-dimensional
// arrays in ascending and descending order.

   PROCEDURE Main()
      LOCAL i, aNum  := { 3, 1, 4, 42 }
      LOCAL aNames   := { { "gilbert", 1 }, ;
                          { "eats"   , 2 }, ;
                          { "grape"  , 3 }  }

      // one-dimensional array
      ? "Ascending: "
      ASort( aNum )

      FOR i := 1 TO Len( aNum )
        ?? aNum[i]                    // result: 1 3 4 42
      NEXT

      ? "Descending: "
      ASort( aNum ,,, {|x,y| x > y } )
      FOR i := 1 TO Len( aNum )       // result: 42 4 3 1
         ?? aNum[i]
      NEXT

      // two-dimensional array sorted by second column
      ? "Descending: "
      ASort( aNames ,,, {|x,y| x[2] > y[2] } )
      FOR i := 1 TO Len( aNames )
         ? aNames[i,1]
      NEXT

      // result:
      // grape
      // eats
      // gilbert
   RETURN
```

At()

Locates the position of a substring within a character string.

Syntax

```
At( <cSearch>, <cString>, [<nStart>], [<nEnd>] ) --> nPos
```

Arguments

<cSearch>

<cSearch> is the substring to search for.

<cString>

<cString> is the searched character string.

<nStart>

A numeric expression indication the position of the first character in *<cString>* to begin the search with. It defaults to 1.

<nEnd>

A numeric expression indicating the position of the last character in *<cString>* to include in the search. It defaults to Len(*<cString>*). If *<nEnd>* is a negative value, the position for the last character is counted from the end of *<cString>*.

Return

The function returns a numeric value which is the position in *<cString>* where *<cSubString>* is found. The return value is zero when *<cSubString>* is not found.

Description

The function At() searches the string *<cString>* from left to right for the character string *<cSearch>*. The search begins at position *<nStart>* and is case-sensitive. If *<cString>* does not contain *<cSearch>*, the function returns 0.

Note: use function RAt() to search *<cString>* from right to left.

Info

See also:	$, HB_ATokens(), AtSkipStrings(), HB_RegEx(), IN, Left(), RAt(), Right(), StrTran(), SubStr()
Category:	Character functions
Source:	rtl\at.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates various results of At().

   PROCEDURE Main()
      LOCAL cString := "xHarbour"

      ? At( "Ha", cString )           // result: 2

      ? At( "ARB" , cString )         // result: 0

      ? At( "ARB" , Upper(cString) )  // result: 3
```

```
   ? At( "r" , cString )          // result: 4

   ? At( "r" , cString, 5 )       // result: 8

   ? At( "r" , cString, 5, 7 )    // result: 0
RETURN
```

AtAdjust()

Justifies a character sequence within a string.

Syntax

```
AtAdjust( <cSearch>    , ;
          <cString>    , ;
          <nEndPos>    , ;
         [<nCount>]    , ;
         [<nSkipChars>], ;
         [<xInsChar>]    ) --> cJustified
```

Arguments

<cSearch>

This is a character string to search for in *<cString>*.

<cString>

This is the character string to find *<cSearch>* in.

<nEndPos>

This is a numeric value specifying the last character of *<cString>* to include in the search.

<nCount>

A numeric value indicating the number of occurrences of finding *<cSearch>* in *<cString>* before the function returns. It defaults to the last occurrence.

<nSkipChars>

This numeric parameter defaults to 0 so that the function begins searching with the first character of *<cString>*. Passing a value > 0 instructs the function to ignore the first *<nSkipChars>* characters in the search.

<xInsChar>

This parameter defaults to 32, i.e. Chr(32) is inserted into *<cString>* for justifying *<cSearch>*. *<xInsChar>* is either a numeric ASCII code or a single character of Valtype()="C".

Return

The function returns the justified string.

Description

AtAdjust() begins searching *<cSearch>* in *<cString>* at the first plus *<nSkipChars>* characters. It terminates searching after *<nCount>* occurrences or after the last occurrence of *<cSearch>*. It inserts *<xInsChar>* up to *<nEnpos>* and returns returns modified string. The function recognizes the CSetAtMuPa() and SetAlike() settings.

Info

See also: AtNum(), CSetAtMuPa(), SetAtLike()
Category: CT:String manipulation, Character functions
Source: ct\atadjust.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example justifies a serias of character strings by
// inserting dots into the result string.

   PROCEDURE Main
      LOCAL aNames := { "Hagen, Walter"   , ;
                        "McKenzie, Scott", ;
                        "Ho, Li-Minh"      }
      LOCAL cName

      FOR EACH cName IN aNames
         cName := AtAdjust( ",", cName, 12, 1, 0, "." )
         ? CharRem( ", ",  cName )
      NEXT

      ** result
      // Hagen......Walter
      // McKenzie...Scott
      // Ho.........Li-Minh
   RETURN
```

ATail()

Returns the last element of an array.

Syntax

```
ATail( <aArray> ) --> xValue
```

Arguments

`<aArray>`

<aArray> is the array to retrieve the last element from.

Return

The function returns the value stored in the last element of *<aArray>*.

Description

The function ATail() retrieves the value of the last element in the array named *<aArray>*. This function does not alter the size of the array.

Info

See also:	[ ] (array), AEval(), AScan(), ASort(), Len()
Category:	Array functions
Source:	vm\arrayshb.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows thee ways of obtaining the last element
// of an array

   PROCEDURE Main()
      LOCAL aArray := { "A", "B", "C" }

      // functional
      ? ATail( aArray )

      // positive subscript
      ? aArray[ Len(aArray) ]

      // negative subscript
      ? aArray[-1]
   RETURN
```

ATan()

Calculates the arc tangent.

Syntax

```
ATan( <nRadians> ) --> nArcTangent
```

Arguments

<nRadians>

A numeric value specifying the angle tangent value in radians.

Return

The function returns the arc tangent as a numeric value between plus/minus Pi()/2.

Info

See also: ACos(), ASin(), ATn2(), DtoR(), RtoD(), SetPrec(), Sin(), Tan()
Category: CT:Math, Mathematical functions, Trigonometric functions
Source: ct\trig.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays results of function ATan().

   PROCEDURE Main
      ? Str( ATan( -1.0 ), 18, 15)  // result: -0.785398163397449
      ? Str( ATan( -0.5 ), 18, 15)  // result: -0.463647609000806
      ? Str( ATan(    0 ), 18, 15)  // result:  0.000000000000000
      ? Str( ATan(  0.5 ), 18, 15)  // result:  0.463647609000806
      ? Str( ATan(  1.0 ), 18, 15)  // result:  0.785398163397449
   RETURN
```

ATn2()

Calculates the radians of an angle from sine and cosine.

Syntax

```
Atn2( <nSine> , <nCosine> ) --> nRadians
```

Arguments

<nSine>

A numeric value specifying the angle sine value in radians.

<nCosine>

A numeric value specifying the angle cosine value in radians.

Return

The function returns the angle sine in radians as a numeric value between plus/minus Pi().

Description

ATn2() accepts sine and cosine values of a given point in radians and calculates from it the angle sine in radians. This is equivalent to ATn2(x/y), where division by zero errors are taken care of.

Info

See also:	ACos(), ASin(), ATan(), DtoR(), RtoD(), SetPrec(), Sin(), Tan()
Category:	CT:Math, Mathematical functions, Trigonometric functions
Source:	ct\trig.c
LIB:	xhb.lib
DLL:	xhbdll.dll

AtNum()

Searches multiple occurrences of a substring within a string.

Syntax

```
AtNum( <cSearch>    , ;
       <cString>    , ;
      [<nCount>]    , ;
      [<nSkipChars>] ) --> nPos
```

Arguments

<cSearch>

This is a character string to search for in <*cString*>.

<cString>

This is the character string to find <*cSearch*> in.

<nCount>

A numeric value indicating the number of occurrences of finding <*cSearch*> in <*cString*> before the function returns. It defaults to the last occurrence.

<nSkipChars>

This numeric parameter defaults to 0 so that the function begins searching with the first character of <*cString*>. Passing a value > 0 instructs the function to ignore the first <*nSkipChars*> characters in the search.

Return

The function returns the ordinal position of the <*nCount*>th occurrence of <*cSearch*> in <*cString*> as a numeric value.

Note: if <*nCount*> is not specified, AtNum() returns the same result as function RAt().

Info

See also:	AfterAtNum(), AtRepl(), BeforAtNum(), NumAt(), CSetAtMuPa(), SetAtLike()
Category:	CT:String manipulation, Character functions
Source:	ct\atnum.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows results of AtNum()

   PROCEDURE Main
      LOCAL cString := "xHarbour Compiler"

      ? AtNum( "r", cString )       // result: 17

      ? AtNum( "r", cString, 1 )    // result:  4
      ? AtNum( "r", cString, 2 )    // result:  8
      ? AtNum( "r", cString, 3 )    // result: 17
      ? AtNum( "r", cString, 4 )    // result:  0
   RETURN
```

AtRepl()

Searches and replaces a substring within a string.

Syntax

```
AtRepl( <cSearch> , ;
        <cString>  , ;
        <cReplace> , ;
        [<nCount>] , ;
        [<lOneOnly>] ) --> cResult
```

Arguments

<cSearch>

This is a character string to search in <*cString*>.

<cString>

This is a character string to find <*cSearch*> in.

<cReplace>

This is a character string that replaces <*cSearch*> in <*cString*>.

<nCount>

A numeric value indicating the number of occurrences of finding <*cSearch*> in <*cString*> before the function returns. It defaults to the last occurrence.

<lOneOnly>

This parameter defaults to .F. (false) so that only the <*nCount*>th occurrence of <*cSearch*> is replaced. Passing .T. (true) instructs the function to replace all occurrences up to and including the <*nCount*>th occurrence.

Return

The function returns the modified string.

Info

See also: AtNum(), AtAdjust(), CSetAtMuPa(), SetAtLike()
Category: CT:String manipulation, Character functions
Source: ct\atrepl.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates return vales of AtRepl()

   PROCEDURE Main
      LOCAL cString := "ABC_ABC_ABC"

      ? AtRepl( "_", cString, "x" )           // result: ABCxABCxABC
      ? AtRepl( "_", cString, "x",, .T. )     // result: ABC_ABCxABC

      ? AtRepl( "AB", cString, "1", 2 )       // result: 1C_1C_ABC
      ? AtRepl( "AB", cString, "1", 2, .T. ) // result: ABC_1C_ABC
   RETURN
```

AtSkipStrings()

Locates the position of a substring within a character string.

Syntax

```
AtSkipStrings( <cSearch>, <cString> ) --> nPos
```

Arguments

`<cSearch>`

<*cSearch*> is the substring to search for.

`<cString>`

<*cString*> is the searched character string.

Return

The function returns a numeric value which is the position in <*cString*> where <*cSubString*> is found. The return value is zero when <*cSubString*> is not found.

Description

The function AtSkipStrings() searches the string <*cString*> from left to right for the character string <*cSearch*>. It works similar to the At() function, but skips all quoted parts of <*cString*>. A quoted part is recognized when it is enclosed in a pair of single or double quotes, or within opening and closing square brackets.

Info

See also:	At(), HB_RegEx(), IN, Left(), RAt(), Right(), StrTran(), SubStr()
Category:	Character functions, xHarbour extensions
Source:	rtl\at.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example compares results of AtSkipStrings() with At()

   PROCEDURE Main
      LOCAL cString1 := '"abc"12abc'
      LOCAL cString2 := "123;'xHarbour;compiler';200702"
      LOCAL nPos

      ? At( "abc", cString1 )             // result: 2

      ? AtSkipSTrings( "abc", cString1 )  // result: 8

      cString1 := cString2

      DO WHILE .NOT. Empty( cString1 )
         nPos := At( ";", cString1 )
         IF nPos > 0
            ? SubStr( cString1, 1, nPos-1 )
            cString1 := SubStr( cString1, nPos+1 )
         ELSE
            ? cString1
            cString1 := ""
         ENDIF
      ENDDO
```

```
   ** output
   // 123
   // 'xHarbour
   // compiler'
   // 200702


   DO WHILE .NOT. Empty( cString2 )
      nPos := AtSkipSTrings( ";", cString2 )
      IF nPos > 0
         ? SubStr( cString2, 1, nPos-1 )
         cString2 := SubStr( cString2, nPos+1 )
      ELSE
         ? cString2
         cString2 := ""
      ENDIF
   ENDDO

   ** output
   // 123
   // 'xHarbour;compiler'
   // 200702
RETURN
```

AtToken()

Returns the position of the n-th token in a string.

Syntax

```
AtToken( <cString>    , ;
        [<cDelimiter>], ;
        [<nCount>]      ) --> nPos
```

Arguments

`<cString>`

This is a character string to find a token in.

`<cDelimiter>`

This character string holds a list of characters recognized as delimiters between tokens. The default list of delimiters consist of non-printable characters having the ASCII codes 0, 9, 10, 13, 26, 32, 138 and 141 and the following punctuation characters: ,.;:!?/\<>()!HSH&%+-*

`<nCount>`

A numeric value indicating the token to retrieve the position for. It defaults to the last token.

Return

The function returns the position of the <*nCount*>th token within <*cString*> as a numeric value, or zero when no more tokens exist. A token is a sequence of characters in <*cString*> delimited with one or two characters of <*cDelimiter*>.

Info

See also:	HB_ATokens(), NumToken(), Token(), TokenLower(), TokenUpper()
Category:	CT:String manipulation, Character functions, Token functions
Source:	ct\token1.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays substrings extracted from a
// tokenized string

   PROCEDURE Main
      LOCAL cDate := DtoC( StoD( "20061231" ) )
      LOCAL cTime := Time()
      LOCAL cFile := "Test.prg"

      ? cDate                                  // result: 12/31/06
      ? SubStr( cDate, AtToken( cDate,, 3 ) )  // result: 06

      ? cTime                                  // result: 18:14:01
      ? SubStr( cTime, AtToken( cTime,, 2 ) )  // result: 14:01

      ? cFile                                  // result: Test.prg
      ? SubStr( cFile, AtToken( cFile ) )      // result: prg
   RETURN
```

BeforAtNum()

Extracts the remainder of a string before the last occurrance of a search string.

Syntax

```
BeforAtNum( <cSearch>    , ;
            <cString>    , ;
           [<nCount>]    , ;
           [<nSkipChars>] ) --> cResult
```

Arguments

<cSearch>

This is a character string to search for in *<cString>*.

<cString>

This is the character string to find *<cSearch>* in.

<nCount>

A numeric value indicating the number of occurrances of finding *<cSearch>* in *<cString>* before the function returns. It defaults to the last occurrance.

<nSkipChars>

This numeric parameter defaults to 0 so that the function begins searching with the first character of *<cString>*. Passing a value > 0 instructs the function to ignore the first *<nSkipChars>* characters in the search.

Return

The function returns the remainder of *<cString>* before the *<nCount>*th occurrance of *<cSearch>* in the string.

Description

BeforAtNum() begins searching *<cSearch>* in *<cString>* at the first plus *<nSkipChars>* characters. It terminates searching after *<nCount>* occurrances or after the last occurrance of *<cSearch>* and returns the remainder of *<cString>* to the left of the found string. The function recognizes the CSetAtMuPa() and SetAlike() settings.

Info

See also: AtNum(), AfterAtNum(), AtRepl(), NumAt(), CSetAtMuPa(), SetAtLike()
Category: CT:String manipulation, Character functions
Source: ct\atnum.c
LIB: xhb.lib
DLL: xhbdll.dll

Bin2I()

Converts a signed short binary integer (2 bytes) into a numeric value.

Syntax

```
Bin2I( <cInteger> ) --> nNumber
```

Arguments

<cInteger>

This is a character string whose first two bytes are converted to a 16 bit signed integer value of numeric data type.

Return

The function returns a numeric value in the range of -(2^15) to +(2^15) - 1.

Description

Bin2I() is a binary conversion function that converts a two byte binary number (Valtype()=="C") to a numeric value (Valtype()=="N"). The parameter *<cInteger>* is usually the return value of function I2Bin().

The range for the numeric return value is determined by a signed short integer.

Info

See also: Bin2L(), Bin2U(), Bin2W(), FRead(), I2Bin(), L2Bin(), U2Bin(), W2Bin(), Word()
Category: Binary functions, Conversion functions
Source: rtl\binnum.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates encoding of binary signed 16 bit integer
// numbers and their conversion to numeric values.

   PROCEDURE Main
      ? Bin2I( Chr(0)   +   Chr(0) )   // result:      0

      ? Bin2I( Chr(1)   +   Chr(0) )   // result:      1
      ? Bin2I( Chr(255) + Chr(255) )   // result:     -1

      ? Bin2I( Chr(0)   +   Chr(1) )   // result:    256
      ? Bin2I( Chr(0)   + Chr(255) )   // result:   -256

      ? Bin2I( Chr(255) + Chr(127) )   // result:  32767
      ? Bin2I( Chr(0)   + Chr(128) )   // result: -32768
   RETURN
```

Bin2L()

Converts a signed long binary integer (4 bytes) into a numeric value.

Syntax

```
Bin2L( <cInteger> ) --> nNumber
```

Arguments

<cInteger>

This is a character string whose first four bytes are converted to a 32 bit signed integer value of numeric data type.

Return

The function returns a numeric value in the range of -(2^31) to +(2^31) - 1.

Description

Bin2L() is a binary conversion function that converts a four byte binary number (Valtype()=="C") to a numeric value (Valtype()=="N"). The parameter *<cInteger>* is usually the return value of function L2Bin().

The range for the numeric return value is determined by a signed long integer.

Info

See also:	Bin2I(), Bin2U(), Bin2W(), FRead(), I2Bin(), L2Bin(), U2Bin(), W2Bin(), Word()
Category:	Binary functions, Conversion functions
Source:	rtl\binnum.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates encoding of binary signed 32 bit integer
// numbers and their conversion to numeric values.

   PROCEDURE Main
      ? Bin2L(   Chr(0) +   Chr(0) +   Chr(0) +   Chr(0) )  // result:          0

      ? Bin2L(   Chr(1) +   Chr(0) +   Chr(0) +   Chr(0) )  // result:          1
      ? Bin2L( Chr(255) + Chr(255) + Chr(255) + Chr(255) )  // result:         -1

      ? Bin2L(   Chr(0) +   Chr(1) +   Chr(0) +   Chr(0) )  // result:        256
      ? Bin2L(   Chr(0) + Chr(255) + Chr(255) + Chr(255) )  // result:       -256

      ? Bin2L( Chr(255) + Chr(255) + Chr(255) + Chr(127) )  // result:  2147483647
      ? Bin2L(   Chr(0) +   Chr(0) +   Chr(0) + Chr(128) )  // result: -2147483648
   RETURN
```

Bin2U()

Converts an unsigned long binary integer (4 bytes) into a numeric value.

Syntax

```
Bin2U( <cInteger> ) --> nNumber
```

Arguments

<cInteger>

This is a character string whose first four bytes are converted to a 32 bit unsigned integer value of numeric data type.

Return

The function returns a numeric value in the range of 0 to (2^32) - 1.

Description

Bin2U() is a binary conversion function that converts a four byte binary number (Valtype()=="C") to a numeric value (Valtype()=="N"). The parameter *<cInteger>* is usually the return value of function U2Bin().

The range for the numeric return value is determined by an unsigned long integer.

Info

See also:	Bin2I(), Bin2L(), Bin2W(), FRead(), I2Bin(), L2Bin(), U2Bin(), W2Bin(), Word()
Category:	Binary functions, Conversion functions
Source:	rtl\binnum.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example reads the number of records from the file header
// of a DBF file and displays the converted numeric value.

   PROCEDURE Main()
      LOCAL nHandle
      LOCAL cInteger := Space( 4 )
      LOCAL cFile    := "Customer.dbf"

      nHandle := FOpen( cFile  )

      IF nHandle > 0
         FSeek( nHandle, 4 )
         FRead( nHandle, @cInteger, 4 )
         ? "Number of records in file:", Bin2U( cInteger )
         FClose( nHandle )
      ELSE
         ? "Cannot open file:", cFile
      ENDIF

   RETURN
```

Bin2W()

Converts an unsigned short binary integer (2 bytes) into a numeric value.

Syntax

```
Bin2W( <cInteger> ) --> nNumber
```

Arguments

<cInteger>

This is a character string whose first two bytes are converted to a 16 bit unsigned integer value of numeric data type.

Return

The function returns a numeric value in the range of 0 to +(2^16) - 1.

Description

Bin2W() is a binary conversion function that converts a two byte binary number (Valtype()=="C") to a numeric value (Valtype()=="N"). The parameter *<cInteger>* is usually the return value of function W2Bin().

The range for the numeric return value is determined by an unsigned short integer.

Info

See also:	Bin2I(), Bin2L(), Bin2U(), FRead(), I2Bin(), L2Bin(), U2Bin(), W2Bin(), Word()
Category:	Binary functions, Conversion functions
Source:	rtl\binnum.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example reads the header size from the file header
// of a DBF file and displays the converted numeric value.

   PROCEDURE Main()
      LOCAL nHandle
      LOCAL cInteger := Space( 2 )
      LOCAL cFile    := "Customer.dbf"

      nHandle := FOpen( cFile  )

      IF nHandle > 0
         FSeek( nHandle, 8 )
         FRead( nHandle, @cInteger, 2 )
         ? "Size of file header:", Bin2W( cInteger )
         FClose( nHandle )
      ELSE
         ? "Cannot open file:", cFile
      ENDIF

   RETURN
```

BitToC()

Translates bits of an integer to a character string.

Syntax

```
BitToC( <nInteger>   , ;
        <cBitPattern>, ;
       [<lSpaces>]     ) --> cResult
```

Arguments

`<nInteger>`

This is a numeric integer value in the range of 0 to 65535.

`<cBitPattern>`

This is a character string of max. 16 characters used as bit pattern.

`<lSpaces>`

If set to .T. (true), blank spaces are inserted in the result string for zero bits. The default value is .F. (false).

Return

The function returns a character string containing the characters of *<cBitPattern>* which are located at the same position as a bit set in *<nInteger>*. When *<lSpaces>* is .T. (true), the return string includes blank spaces at the position of unset bits. When it is .F. (false), unset bits of *<nInteger>* are not included in the return string.

Info

See also: CtoBit(), CtoN(), HB_BitIsSet()
Category: CT:NumBits, Bitwise functions, Numbers and Bits
Source: ct\numconv.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays results of BitToC() in comparison to
// the binary representation of a number

   PROCEDURE Main

      // binary representation of 11
      ? NtoC( 11, 2, "0" )          // result: 1011

      // zero bit is ingnored
      ? BitToC( 11, "ABCD" )        // result: ACD

      // zero bit yields blank space
      ? BitToC( 11, "ABCD", .T. )   // result: A CD
   RETURN
```

Blank()

Returns empty values for the data types A, C, D, L, M and N.

Syntax

```
Blank( <xValue>, [<lSpaces>] ) --> xEmptyValue
```

Arguments

<xValue>

This is a value of data type A, C, D, L, M or N.

<lSpaces>

This parameter defaults to .F. (false), so that a null string ("") is returned for values of data type C or M. When .T. (true) is passed, the function returns a string of blank spaces with the same length as *<xValue>*.

Return

The function returns an empty value with the same data type as the first parameter.

Info

See also: Array(), CtoD(), Nul()
Category: CT:Miscellaneous, Miscellaneous functions
Source: ct\blank.prg
LIB: xhb.lib
DLL: xhbdll.dll

BlobDirectExport()

Exports the contents of a binary large object (BLOB) to a file.

Syntax

```
BlobDirectExport( <nBlobID>, <cTargetFile>, [<nMode>] ) --> lSuccess
```

Arguments

<nBlobID>

<nBlobID> is the numeric identifier of the binary large object to export. The ID can be obtained using functions BlobDirectPut() or DbFieldInfo(DBS_BLOB_POINTER, *<nFieldPos>*). An array of blob IDs can also be obtained using BlobRootGet() when the blob IDs are previously written to the root area of a blob file.

<cTargetFile>

This is a character string containing the name of the file to receive the exported data. It must be specified with a file extension. If *<cTargetFile>* contains no drive and/or directory information, the current directory is used. SET DEFAULT and SET PATH settings are ignored.

<nMode>

A numeric value specifying how to write data to the target file. #define constants must be used for this parameter.

Constants for BlobDirectExport()

Constant	Description
BLOB_EXPORT_APPEND	Appends to the target file
BLOB_EXPORT_OVERWRITE *)	Overwrites the target file

**) default value*

Return

The return value is .T. (true) if data is successfully exported, otherwise .F. (false) is returned.

Description

BlobDirectExport() copies the contents of a single binary large object (BLOB) to an external file. The object is identified by its numeric ID. A BLOB is stored either in a memo field (DBFCDX RDD) or in a stand alone DBV file which is not associated with a database file (DBFBLOB RDD). BlobDirectExport() is usually required for the latter case. If BLOBs are stored in a database with an associated memo file, function BlobExport() is more comfortable to use, since it accepts a numeric field position of a memo field, rather than a numeric BLOB identifier.

The numeric BLOB ID is obtained using DbFieldInfo() or must be taken from the array returned by BlobRootGet().

If the target file does not exist, it is created. If it exists, it is either overwritten or data is appended to the file, depending on *<nMode>*. When the file operation fails, BlobDirectExport() returns .F. (false) and function NetErr() is set to .T. (true). This can happen when the target file is currently locked by another process in concurrent file access.

Notes: BLOBs are stored in memo files having the .FPT or .DBV extension. They are maintained by the RDDs DBFCDX or DBFBLOB. DBFNTX, in contrast, does not support BLOBs.

The file Blob.ch must be #included for BlobDirectExport() to work.

Info

See also: BlobDirectGet(), BlobDirectImport(), BlobDirectPut(), BlobExport(), BlobRootGet(), DbFieldInfo()
Category: Blob functions, Database functions
Header: blob.ch
Source: rdd\dbcmd.c, rdd\dbffpt\dbffpt1.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates two possibilities of exporting BLOB data.
// The first example uses the DBFCDX RDD while the second uses the
// DBFBLOB RDD which does not maintain a database file, but only the
// BLOB file.

   #include "Blob.ch"

   REQUEST DBFBLOB
   REQUEST DBFCDX

   PROCEDURE Main
      LOCAL nBlobID, nFieldPos
      LOCAL aBlobID

      // DBF and FPT file exist
      USE PhotoArchive ALIAS Photos VIA "DBFCDX"
      LOCATE FOR FIELD->PhotoName = "Sunset in Malibu"

      IF Found()
         nFieldPos := FieldPos( "JPEG" )
         nBlobID   := DbFieldInfo( DBS_BLOB_POINTER, nFieldPos )

         IF BlobDirectExport( nBlobID, "Sunset.jpg" )
            ? "File successfully eported"
         ELSE
            ? "Unable to export BLOB data"
         ENDIF
      ENDIF

      // Only DBV file exists
      USE BlobStorage ALIAS Blob VIA "DBFBLOB"
      aBlobID := BlobRootGet()

      IF Valtype( aBlobID ) == "A"
         // Export second BLOB to file
         IF BlobDirectExport( aBlobID[2], "Picture02.jpg" )
            ? "File successfully eported"
         ELSE
            ? "Unable to export BLOB data"
         ENDIF
      ELSE
         ? "Error: Blob root is not created"
      ENDIF

      USE
   RETURN
```

BlobDirectGet()

Loads data of a binary large object (BLOB) into memory.

Syntax

```
BlobDirectGet( <nBlobID>, [<nStart>], [<nCount>] ) --> xBlobData
```

Arguments

`<nBlobID>`

<nBlobID> is the numeric identifier of the binary large object (BLOB) to load into memory. The ID can be obtained using functions BlobDirectPut() or DbFieldInfo(DBS_BLOB_POINTER, *<nFieldPos>*). An array of blob IDs can also be obtained using BlobRootGet(), when the blob IDs are previously written to the root area of a blob file.

`<nStart>`

This is a numeric value specifying the first byte of the BLOB to include in the returned string. If *<nStart>* is a positive number, data is loaded from the beginning, or the left side, of the BLOB. If *<nStart>* is a negative number, the data is loaded from the end, or the right side, of the BLOB.

Note: Both parameters, *<nStart>* and *<nCount>*, are ignored if the BLOB does not contain a character string.

`<nCount>`

This is a numeric value specifying the number of bytes to load, beginning at position *<nStart>*. If omitted, all bytes from *<nStart>* to the end of the BLOB are loaded.

Return

The function returns the BLOB data loaded into memory. The data type of a BLOB depends on the stored BLOB. It can be determined using function Valtype().

Description

BlobDirectGet() loads the contents of a single binary large object (BLOB) into memory. The object is identified by its numeric ID. A BLOB is stored either in a memo field (DBFCDX RDD) or in a stand alone DBV file which is not associated with a database file (DBFBLOB RDD). BlobDirectGet() is usually required for the latter case. If BLOBs are stored in a database with an associated memo file, function BlobGet() is more comfortable to use, since it accepts a numeric field position of a memo field, rather than a numeric BLOB identifier.

The numeric BLOB ID is obtained using DbFieldInfo() or must be taken from the array returned by BlobRootGet().

Notes: BLOBs are stored in memo files having the .FPT or .DBV extension. They are maintained by the RDDs DBFCDX or DBFBLOB. DBFNTX, in contrast, does not support BLOBs.

The file Blob.ch must be #included for BlobDirectGet() to work.

Info

See also: BlobDirectExport(), BlobDirectImport(), BlobDirectPut(), BlobExport(), BlobRootGet(), DbFieldInfo()
Category: Blob functions, Database functions
Header: blob.ch
Source: rdd\dbcmd.c, rdd\dbffpt\dbffpt1.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates two possibilities of loading BLOB data.
// The first example uses the DBFCDX RDD while the second uses the
// DBFBLOB RDD which does not maintain a database file, but only the
// BLOB file.

   #include "Blob.ch"

   REQUEST DBFBLOB
   REQUEST DBFCDX

   PROCEDURE Main
      LOCAL nBlobID, nFieldPos
      LOCAL aBlobID
      LOCAL cImage

      // DBF and FPT file exist
      USE PhotoArchive ALIAS Photos VIA "DBFCDX"
      LOCATE FOR FIELD->PhotoName = "Sunset in Malibu"

      IF Found()
         // Get numeric field position
         nFieldPos := FieldPos( "JPEG" )

         // Get numeric BLOB ID
         nBlobID := DbFieldInfo( DBS_BLOB_POINTER, nFieldPos )

         // Load BLOB
         cImage := BlobDirectGet( nBlobID )
         // < ... process BLOB data ... >
      ENDIF

      // Only DBV file exists
      USE BlobStorage ALIAS Blob VIA "DBFBLOB"
      aBlobID := BlobRootGet()

      IF Valtype( aBlobID ) == "A"
         // Load second BLOB
         cImage := BlobDirectGet( nBlobID[2] )
         // < ... process BLOB data ... >
      ELSE
         ? "Error: Blob root is not created"
      ENDIF

      USE
   RETURN
```

BlobDirectImport()

Imports a file into a binary large object (BLOB).

Syntax

```
BlobDirectImport( <nOldBlobID>, <cSourceFile> ) --> nNewBlobID
```

Arguments

`<nOldBlobID>`

<*nOldBlobID*> is the numeric identifier of the binary large object to receive the file contents of <*cSourceFile*>. The ID can be obtained using functions BlobDirectPut() or DbFieldInfo(DBS_BLOB_POINTER, <*nFieldPos*>). An array of blob IDs can also be obtained using BlobRootGet() when the blob IDs are previously written to the root area of a blob file.

All data of <*nOldBlobID*> is discarded, and the BLOB receives a new ID which is returned by the function. If zero is passed, a new BLOB is added to the BLOB file.

`<cSourceFile>`

This is a character string containing the name of the file to import BLOB data from. It must be specified with a file extension. If <*cSourceFile*> contains no drive and/or directory information, the current directory is used. SET DEFAULT and SET PATH settings are ignored.

If <*cSourceFile*> does not exist, a runtime error is raised. If the file cannot be opened due to another process having exclusive access to the file, function NetErr() is set to .T. (true).

Return

The function returns a numeric value which is the BLOB identifier of the new BLOB. The return value is zero, when the operation fails.

Description

BlobDirectImport() copies the contents of an external file into a single binary large object (BLOB). The object is identified by its numeric ID. A BLOB is stored either in a memo field (DBFCDX RDD) or in a stand alone DBV file which is not associated with a database file (DBFBLOB RDD). BlobDirectImport() is usually required for the latter case. If BLOBs are stored in a database with an associated memo file, function FieldPut() is more comfortable to use, since it accepts a numeric field position of a memo field, rather than a numeric Blob identifier.

The numeric Blob ID is obtained using DbFieldInfo() or must be taken from the array returned by BlobRootGet().

When data is imported into an existing BLOB, the BLOB is entirely discarded and a new BLOB is created. As a consequence, the existing BLOB ID <*nOldBlobID*> becomes invalid, and BlobDirectImport() returns a new BLOB ID. The imported data can only be accessed via the new BLOB ID.

Notes: BLOBs are stored in memo files having the .FPT or .DBV extension. They are maintained by the RDDs DBFCDX or DBFBLOB. DBFNTX, in contrast, does not support BLOBs.

The file Blob.ch must be #included for BlobDirectImport() to work.

Info

See also: BlobDirectGet(), BlobDirectExport(), BlobExport(), BlobGet(), BlobImport(), BlobRootGet(), DbFieldInfo()
Category: Blob functions, Database functions
Header: blob.ch
Source: rdd\dbcmd.c, rdd\dbffpt\dbffpt1.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates how to import JPG files into a DBV file.

   #include "Blob.ch"

   REQUEST DBFBLOB

   PROCEDURE Main
      LOCAL aFile
      LOCAL aFiles  := Directory( "*.jpg" )
      LOCAL aBlobID := {}

      DbCreate( "ImageArchive", {}, "DBFBLOB" )

      USE ImageArchive NEW VIA "DBFBLOB"

      FOR EACH aFile IN aFiles
         AAdd( aBlobID, BlobDirectImport( 0, aFile[1] ) )
      NEXT

      IF BlobRootLock() .AND. BlobRootPut( aBlobID )
         BlobRootUnlock()
      ELSE
         Alert( "Unable to lock BLOB root;try again later" )
      ENDIF

      USE
   RETURN
```

BlobDirectPut()

Assigns data to a binary large object (BLOB).

Syntax

```
BlobDirectPut( <nOldblobID>, <xBlobData> ) --> nNewblobID
```

Arguments

<nOldBlobID>

<*nOldBlobID*> is the numeric identifier of the binary large object to assign <*xBlobData*> to. The ID can be obtained using functions BlobDirectPut() or DbFieldInfo(DBS_BLOB_POINTER, <*nFieldPos*>). An array of blob IDs can also be obtained using BlobRootGet() when the blob IDs are previously written to the root area of a blob file.

All data of <*nOldBlobID*> is discarded, and the BLOB receives a new ID which is returned by the function. If zero is passed, a new BLOB is added to the BLOB file.

<xBlobData>

This is the value to assign to the BLOB. The supported data type of <*xBlobData*> can vary between RDDs maintaining BLOB files.

Return

The function returns a numeric value which is the BLOB identifier of the new BLOB. The return value is zero, when the operation fails.

Description

BlobDirectPut() assigns the value of <*xBlobData*> to a single binary large object (BLOB). The object is identified by its numeric ID. A BLOB is stored either in a memo field (DBFCDX RDD) or in a stand alone DBV file which is not associated with a database file (DBFBLOB RDD). BlobDirectPut() is usually required for the latter case. If BLOBs are stored in a database with an associated memo file, function FieldPut() is more comfortable to use, since it accepts a numeric field position of a memo field, rather than a numeric Blob identifier.

The numeric Blob ID is obtained using DbFieldInfo() or must be taken from the array returned by BlobRootGet().

When data is assigned to an existing BLOB, the BLOB is entirely discarded and a new BLOB is created. As a consequence, the existing BLOB ID <*nOldBlobID*> becomes invalid, and BlobDirectPut() returns a new BLOB ID. The imported data can only be accessed via the new BLOB ID.

Notes: BLOBs are stored in memo files having the .FPT or .DBV extension. They are maintained by the RDDs DBFCDX or DBFBLOB. DBFNTX, in contrast, does not support BLOBs.

The file Blob.ch must be #included for BlobDirectPut() to work.

Info

See also: BlobDirectExport(), BlobDirectGet(), BlobDirectImport(), BlobExport(), BlobRootGet(), DbFieldInfo()
Category: Blob functions, Database functions
Header: blob.ch
Source: rdd\dbcmd.c, rdd\dbffpt\dbffpt1.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates how to collect contents of files
// in a single DBV file.

   #include "Blob.ch"

   REQUEST DBFBLOB

   PROCEDURE Main
      LOCAL aFile, cText
      LOCAL aFiles  := Directory( "*.prg" )
      LOCAL aBlobID := {}

      DbCreate( "Repository", {}, "DBFBLOB" )

      USE Repository NEW VIA "DBFBLOB"

      FOR EACH aFile IN aFiles
         cText := MemoRead( aFile[1] )
         AAdd( aBlobID, BlobDirectPut( 0, cText ) )
      NEXT

      IF BlobRootLock() .AND. BlobRootPut( aBlobID )
         BlobRootUnlock()
      ELSE
         Alert( "Unable to lock BLOB root;try again later" )
      ENDIF

      USE
   RETURN
```

BlobExport()

Exports the contents of a memo field holding a binary large object (BLOB) to a file.

Syntax

```
BlobExport( <nFieldPos>, <cTargetFile>, [<nMode>] ) --> lSuccess
```

Arguments

`<nFieldPos>`

This is a numeric value specifying the ordinal position of the memo field holding the BLOB to be exported.

`<cTargetFile>`

This is a character string containing the name of the file to receive the exported data. It must be specified with a file extension. If *<cTargetFile>* contains no drive and/or directory information, the current directory is used. SET DEFAULT and SET PATH settings are ignored.

`<nMode>`

A numeric value specifying how to write data to the target file. #define constants must be used for this parameter.

Constants for BlobExport()

Constant	Description
BLOB_EXPORT_APPEND	Appends to the target file
BLOB_EXPORT_OVERWRITE *)	Overwrites the target file

**) default value*

Return

The return value is .T. (true) if data is successfully exported, otherwise .F. (false) is returned.

Description

BlobExport() copies the contents of a single binary large object (BLOB) to an external file. The object is identified by the ordinal position of the memo field holding the BLOB data. This position can be obtained using function FieldPos().

If the target file does not exist, it is created. If it exists, it is either overwritten or data is appended to the file, depending on *<nMode>*. When the file operation fails, BlobExport() returns .F. (false) and function NetErr() is set to .T. (true). This can happen when the target file is currently locked by another process in concurrent file access.

Note: the file Blob.ch must be #included for BlobExport() to work.

Info

See also:	BlobDirectGet(), BlobDirectExport(), BlobDirectImport(), BlobDirectPut(), BlobRootGet(), DbFieldInfo()
Category:	Blob functions, Database functions
Header:	blob.ch
Source:	rdd\dbcmd.c, rdd\dbffpt\dbffpt1.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates how to export BLOB data to an external file.

   #include "Blob.ch"

   REQUEST DBFCDX

   PROCEDURE Main
      LOCAL nFieldPos

      USE PhotoArchive ALIAS Photos VIA "DBFCDX"
      LOCATE FOR FIELD->PhotoName = "Sunset in Malibu"

      IF Found()
         nFieldPos := FieldPos( "JPEG" )

         IF BlobExport( nFieldPos, "Sunset.jpg" )
            ? "File successfully eported"
         ELSE
            ? "Unable to export BLOB data"
         ENDIF
      ENDIF

      USE
   RETURN
```

BlobGet()

Reads the contents of a memo field holding a binary large object (BLOB).

Syntax

```
BlobGet( <nFieldPos>, [<nStart>], [<nCount>] ) --> xBlobData
```

Arguments

`<nFieldPos>`

This is a numeric value specifying the ordinal position of the memo field holding the BLOB to read.

`<nStart>`

This is a numeric value specifying the first byte of the BLOB to include in the returned string. If *<nStart>* is a positive number, data is loaded from the beginning, or the left side, of the BLOB. If *<nStart>* is a negative number, the data is loaded from the end, or the right side, of the BLOB.

Note: Both parameters, *<nStart>* and *<nCount>*, are ignored if the BLOB does not contain a character string.

`<nCount>`

This is a numeric value specifying the number of bytes to load, beginning at position *<nStart>*. If omitted, all bytes from *<nStart>* to the end of the BLOB are loaded.

Return

The function returns the BLOB data loaded into memory. The data type of a BLOB depends on the stored BLOB. It can be determined using function Valtype(). If the indicated field at position *<nFieldPos>* is not amemo field, the return value is NIL.

Description

BlobGet() exists for compatibility reasons. It does the same as function FieldGet() but is restricted to memo fields.

Note: the file Blob.ch must be #included for BlobGet() to work.

Info

See also:	BlobDirectGet(), FieldGet()
Category:	Blob functions, Database functions
Header:	blob.ch
Source:	rdd\dbcmd.c, rdd\dbffpt\dbffpt1.c
LIB:	xhb.lib
DLL:	xhbdll.dll

BlobImport()

Imports a file into a memo field.

Syntax

```
BlobImport( <nFieldPos>, <cSourceFile> ) --> lSuccess
```

Arguments

`<nFieldPos>`

This is a numeric value specifying the ordinal position of the memo field to receive the contents of the file *<cSourceFIle>* .

`<cSourceFile>`

This is a character string containing the name of the file to import BLOB data from. It must be specified with a file extension. If *<cSourceFile>* contains no drive and/or directory information, the current directory is used. SET DEFAULT and SET PATH settings are ignored.

If *<cSourceFile>* does not exist, a runtime error is raised. If the file cannot be opened due to another process having exclusive access to the file, function NetErr() is set to .T. (true).

Return

The return value is .T. (true) if data is successfully imported, otherwise .F. (false) is returned.

Description

BlobImport() copies the contents of an external file into a memo field holding binary large objects (BLOB). The memo field is identified by its ordinal position, which can be obtained using function FieldPos().

Note: the file Blob.ch must be #included for BlobImport() to work.

Info

See also:	BlobDirectGet(), BlobDirectExport(), BlobExport(), BlobGet(), BlobRootGet(), FieldPos()
Category:	Blob functions, Database functions
Header:	blob.ch
Source:	rdd\dbcmd.c, rdd\dbffpt\dbffpt1.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// In this example, an image file is moved into an image
// archive database

   #include "Blob.ch"

   REQUEST DBFCDX

   PROCEDURE Main
      LOCAL nFieldPos
      LOCAL cFileName := "Sunset.jpg"

      USE PhotoArchive ALIAS Photos VIA "DBFCDX"
      APPEND BLANK

      IF .NOT. NetErr()
```

```
      nFieldPos := FieldPos( "JPEG" )
      REPLACE Photos->PhotoName WITH "Sunset in Malibu"
      REPLACE Photos->FileName  WITH cFileName

      IF BlobImport( nFieldPos, cFileName )
         FErase( cFileName )
      ENDIF
   ENDIF

   USE
RETURN
```

BlobRootDelete()

Deleted the root area of a BLOB file.

Syntax

```
BlobRootDelete() --> lSuccess
```

Return

The return value is .T. (true) if the root area is successfully deleted, otherwise .F. (false) is returned.

Description

BlobRootDelete() removes the data stored in the root area of a BLOB file. This data is written by BlobRootPut().

Note: the file Blob.ch must be #included for BlobRootDelete() to work.

Info

See also:	BlobRootGet(), BlobRootPut(), BlobRootLock()
Category:	Blob functions, Database functions
Header:	blob.ch
Source:	rdd\dbcmd.c, rdd\dbffpt\dbffpt1.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates the coding pattern that must be used to
// remove the root area of a BLOB file.

   #include "Blob.ch"

   REQUEST DBFCDX

   PROCEDURE Main
      USE PhotoArchive ALIAS Photos VIA "DBFCDX"

      IF BlobRootLock()
         BlobRootDelete()
         BlobRootUnlock()
      ELSE
         Alert( "Unable to lock the root area" )
      ENDIF

      USE
   RETURN
```

BlobRootGet()

Retrieves data from the root area of a BLOB file.

Syntax

```
BlobRootGet() --> xBlobData
```

Return

The function returns the data stored in the root area of a BLOB file. The data type of this data can be determined using Valtype().

Description

BlobRootGet() is used to read the data stored in the root area of a BLOB file. This data must be previously written with BlobRootPut().

When the BLOB file is open in SHARED mode, function BlobRootLock() must be called to avoid conflicts in concurrent file access. This is the only function that places a lock on the root area of a BLOB file.

Note: the file Blob.ch must be #included for BlobRootGet() to work.

Info

See also:	BlobRootDelete(), BlobRootLock(), BlobRootPut()
Category:	Blob functions, Database functions
Header:	blob.ch
Source:	rdd\dbcmd.c, rdd\dbffpt\dbffpt1.c
LIB:	xhb.lib
DLL:	xhbdll.dll

BlobRootLock()

Places a lock on the root area of a BLOB file.

Syntax

```
BlobRootLock() --> lSuccess
```

Return

The return value is .T. (true) if the root area is successfully locked, otherwise .F. (false) is returned.

Description

BlobRootLock() is the only function that can place a lock on the root area of a BLOB file. A lock is required to avoid concurrency conflicts when a BLOB file is open in SHARED mode. It must be released later with BlobRootUnlock().

Note: the file Blob.ch must be #included for BlobRootLock() to work.

Info

See also:	BlobRootGet(), BlobRootPut(), BlobRootUnlock()
Category:	Blob functions, Database functions
Header:	blob.ch
Source:	rdd\dbcmd.c, rdd\dbffpt\dbffpt1.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates the coding pattern that must be used to
// read the root area of a shared BLOB file.

   #include "Blob.ch"

   REQUEST DBFCDX

   PROCEDURE Main
      LOCAL xBlobRoot

      USE PhotoArchive ALIAS Photos VIA "DBFCDX" SHARED

      IF BlobRootLock()
         xBlobRoot := BlobRootGet()
         BlobRootUnlock()
         ? Valtype( xBlobRoot )
      ELSE
         Alert( "Unable to lock the root area" )
      ENDIF

      USE
   RETURN
```

BlobRootPut()

Stores data in the root area of a BLOB file.

Syntax

```
BlobRootPut( <xBlobData> ) --> lSuccess
```

Arguments

<xBlobData>

This is the value to assign to the BLOB. The supported data type of *<xBlobData>* can vary between RDDs maintaining BLOB files.

Return

The return value is .T. (true) if the data is successfully written to the root area of a BLOB file, otherwise .F. (false) is returned.

Description

BlobRootPut() is used to store data in the root area of a BLOB file. The root area is a unique storage place within a BLOB file that is only accessible with the functions BlobRootGet() and BlobRootPut().

When the BLOB file is open in SHARED mode, function BlobRootLock() must be called to avoid conflicts in concurrent file access. This is the only function that places a lock on the root area of a BLOB file.

Note: the file Blob.ch must be #included for BlobRootPut() to work.

Info

See also:	BlobRootDelete(), BlobRootGet(), BlobRootLock()
Category:	Blob functions, Database functions
Header:	blob.ch
Source:	rdd\dbcmd.c, rdd\dbffpt\dbffpt1.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// This example creates a stand alone DBV file and stores the BLOB IDs
// in the root area of the BLOB file.

   #include "Blob.ch"

   REQUEST DBFBLOB

   PROCEDURE Main
      LOCAL aFile
      LOCAL aFiles  := Directory( "*.jpg" )
      LOCAL aBlobID := {}

      DbCreate( "ImageArchive", {}, "DBFBLOB" )

      USE ImageArchive NEW VIA "DBFBLOB"

      FOR EACH aFile IN aFiles
         AAdd( aBlobID, BlobDirectImport( 0, aFile[1] ) )
      NEXT

      IF BlobRootLock() .AND. BlobRootPut( aBlobID )
```

```
      BlobRootUnlock()
   ELSE
      Alert( "Unable to lock BLOB root;try again later" )
   ENDIF

   USE
RETURN
```

BlobRootUnlock()

Releases the lock on the root area of a BLOB file.

Syntax

```
BlobRootUnlock() --> NIL
```

Return

The return value is always NIL.

Description

BlobRootUnlock() releases the a lock previously set with function BlobRootLock(). A lock is required when a BLOB file is open in SHARED mode and the root area of a BLOB file is accessed.

Note: the file Blob.ch must be #included for BlobRootUnlock() to work.

Info

See also:	BlobRootGet(), BlobRootLock(), BlobRootPut()
Category:	Blob functions, Database functions
Header:	blob.ch
Source:	rdd\dbcmd.c, rdd\dbffpt\dbffpt1.c
LIB:	xhb.lib
DLL:	xhbdll.dll

BoF()

Determines if the record pointer reached the begin-of-file boundary.

Syntax

```
BoF() --> lBeginOfFile
```

Return

The function returns .T. (true) when an attempt is made to move the record pointer in a work area backwards beyond the first logical record, or when the database in a work area has no records. In all other situations, Bof() returns .F. (false).

Description

The database function Bof() is required to determine if the record pointer has reached the begin-of-file during a backwards oriented database navigation. This can only be achieved with passing a negative value to SKIP or DbSkip(). Once the begin-of-file is reached, the record pointer remains at the first logical record and Bof() returns .T. (true). Bof() continues to return .T. (true) until the record pointer is moved forwards again.

BoF() operates on logical records unless no index is open in the work area and no filter or scope is set. With no filter, index or scope, Bof() returns .T. (true) when attempting to move the record pointer to a record number < *1*.

The function operates in the current work area. To query the begin-of-file in a different work area, use an aliased expression.

Info

See also:	DbGoto(), DbSkip(), Eof(), GO, INDEX, SET DELETED, SET FILTER, SET SCOPE, SKIP
Category:	Database functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example illustrates the Bof() state of a work area during
// physical and logical record pointer movement.

   #include "Ord.ch"

   PROCEDURE Main
      LOCAL nCount := 0

      USE Customer
      INDEX ON Upper(LastName+FirstName) TO Cust01

      ? BoF(), LastName                // result: .F. Alberts

      // physical movement of record pointer
      GOTO 10
      ? BoF(), nCount, Recno()         // result: .F.   0   10

      // logical movement of record pointer
      DO WHILE .NOT. BoF()
         SKIP -1
         nCount ++
```

```
   ENDDO

   // Note: from record #10 we skipped 5 records back
   // to hit begin-of-file, and ended up at record #20
   // That's because the database is indexed.
   ? BoF(), nCount, Recno()         // result: .T.   5   20

   ? BoF(), LastName                // result: .T. Alberts
   SKIP 0
   ? BoF(), LastName                // result: .T. Alberts

   SKIP 1
   ? BoF(), LastName                // result: .F. Becker

   SET SCOPETOP TO "G"              // restrict logical navigation

   GO TOP
   ? BoF(), LastName                // result: .F. Grastner

   SKIP -1                          // BoF reached due to scope
   ? BoF(), LastName                // result: .T. Grastner

   CLOSE Customer
RETURN
```

BoM()

Returns the date of the first day of a month.

Syntax

```
BoM( [<dDate>] ) --> dFirstDayOfMonth
```

Arguments

<dDate>

Any Date value, except for an empty date, can be passed. The default is the return value of Date().

Return

The function returns the first day of the month that includes *<dDate>* as a Date value, or an empty date on error.

Info

See also:	BoQ(), BoY(), EoM(), EoQ(), EoY(), Month()
Category:	CT:DateTime, Date and time
Source:	ct\datetime.c
LIB:	xhb.lib
DLL:	xhbdll.dll

BoQ()

Returns the date of the first day of a quarter.

Syntax

```
BoQ( [<dDate>] ) --> dFirstDayOfQuarter
```

Arguments

<dDate>

Any Date value, except for an empty date, can be passed. The default is the return value of Date().

Return

The function returns the first day of the quarter that includes *<dDate>* as a Date value, or an empty date on error.

Info

See also:	BoM(), BoY(), EoM(), EoQ(), EoY(), Quarter()
Category:	CT:DateTime, Date and time
Source:	ct\datetime.c
LIB:	xhb.lib
DLL:	xhbdll.dll

BoY()

Returns the date of the first day of a year.

Syntax

```
BoY( [<dDate>] ) --> dFirstDayOfYear
```

Arguments

<dDate>

Any Date value, except for an empty date, can be passed. The default is the return value of Date().

Return

Returns the 1st of January of the year specified with *<dDate>*.

Info

See also:	BoM(), BoQ(), EoM(), EoQ(), EoY(), Year()
Category:	CT:DateTime, Date and time
Source:	ct\datetime.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Break()

Exits from a BEGIN SEQUENCE block.

Syntax

```
Break( <Expression> ) --> NIL
```

Arguments

<Expression>

This is an arbitrary expression whose value is passed to the RECOVER USING statement of the last BEGIN SEQUENCE .. ENDSEQUENCE block. Use the value NIL for <*Expression*> to pass no expression value.

Return

The return value is always NIL.

Description

The Break() function branches program flow to the RECOVER statement of the last BEGIN SEQUENCE block and passes the value of <*Expression*> to the variable defined with the USING <*errorVar*> option, if defined. If BEGIN SEQUENCE is used with no RECOVER statement, program flow continues with the next executable statement following ENDSEQUENCE.

Break() is mainly used in conjunction with error handling, when a user defined code block replaces the standard ErrorBlock().

Info

See also:	BEGIN SEQUENCE, ErrorBlock(), ErrorNew(), Throw(), TRY...CATCH
Category:	Error functions
Source:	vm\break.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows a typical usage scenario for the Break() function.
// It is called from within a user defined error code block. The
// error code block receives an error object created by the xHarbour
// runtime system. The error object is passed to Break() and ends up
// in the RECOVER USING variable. The error condition can then be
// inspected.

   PROCEDURE Main
      OpenDatabase()

      ? "Files are open"
      ? "Program can start"
   RETURN


   PROCEDURE OpenDatabase
      LOCAL oError
      LOCAL bError := ErrorBlock( {|e| Break(e) } )

      BEGIN SEQUENCE
         USE Customer NEW SHARED
         USE Invoice  NEW SHARED
```

```
      ErrorBlock( bError )
   RECOVER USING oError
      ErrorBlock( bError )

      ? "Unrecoverable error:"

      ? oError:description
      ? oError:operation
      ? oError:subsystem
      QUIT
   ENDSEQUENCE
RETURN
```

Browse()

Browse a database file

Syntax

```
Browse( [<nTop>]    , ;
        [<nLeft>]   , ;
        [<nBottom>], ;
        [<nRight>]    ) --> lOk
```

Arguments

<nTop> and <nLeft>

Numeric values indicating the screen coordinates for the upper left corner of the Browse() window. The default value for *<nTop>* is 1 and for *<nLeft>* is zero.

<nBottom>

Numeric value indicating the bottom row screen coordinate. It defaults to MaxRow().

<nRight>

Numeric value indicating the right column screen coordinate. It defaults to MaxCol().

Return

The return value is .F. (false) if there is no database open in the work area, otherwise .T. (true) is returned.

Description

The Browse() function displays a simple database browser in a console window. Data is taken from the database open in the current work area, unless Browse() is used in an aliased expression.

The function provides for basic database navigation and editing using the following keys:

Table navigation with Browse()

Key	Description
Up	Move up one row (previous record)
Down	Move down one row (next record)
PgUp	Move to the previous screen
PgDn	Move to the next screen
Left	Move one column to the left (previous field)
Right	Move one column to the right (next field)
Home	Move to the leftmost visible column
End	Move to the rightmost visible column
Ctrl+Home	Move to the leftmost column
Ctrl+End	Move to the rightmost column
Ctrl+Left	Pan one column to the left
Ctrl+Right	Pan one column to the right
Ctrl+PgUp	Move to the top of the file
Ctrl+PgDn	Move to the end of the file
Enter	Edit current cell
Del	Toggles the deleted flag of current record.
Esc	Terminate Browse()

A status line is displayed on top of the Browse() window presenting the user the following information:

Browse() status information

Display	Description
Record ###/###	Current record number / Total number of records.
<none>	There are no records, the database is empty.
<new>	A new record is about to be added.
<Deleted>	Current record is deleted.
<bof>	Top-of-file is reached.

A new record is automatically added when the user browses past the end of the database and edits a field on the new record.

Info

See also: DbEdit(), TBrowse(), TBrowseDB()
Category: Database functions, UI functions
Source: rtl\browse.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example shows how to browse an indexed database.

   PROCEDURE Main

      USE Customer
      INDEX ON Upper(Lastname+Firstname) TO Cust01

      Browse()

      CLOSE Customer
   RETURN
```

CallDll()

Executes a function located in a dynamically loaded external library.

Syntax

```
CallDll( <pFunction> [,<xParams,...>] ) --> xResult
```

Arguments

`<pFunction>`

This is a pointer holding the memory address of the function to execute. It is returned from function GetProcAddress().

`<xParams>`

The values of all following parameters specified in a comma separated list are passed on to the DLL function.

Return

The function returns the result of the called DLL function as a numeric value.

Description

CallDll() exeutes a function located in a DLL that is not created by the xHarbour compiler. It works almost like function DllCall(), but accepts a function pointer obtained by GetProcAddress(), rather than the name of the DLL file and function. The calling convention is DC_CALL_STD. Refer to function DllCall() for more information on calling external DLL functions.

Info

See also:	DllCall(), DllExecuteCall(), DllPrepareCall(), FreeLibrary(), GetProcAddress(), GetLastError(), LoadLibrary()
Category:	DLL functions, xHarbour extensions
Source:	rtl\dllcall.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example implements a simple command line utility
// that opens a file using the associated file viewer.

   PROCEDURE Main( cFile )
      LOCAL nDll, pFunc

      IF cFile == NIL .OR. .NOT. File( cFile )
         CLS
         ? "File name must be specified"
         QUIT
      ENDIF

      nDll  := DllLoad( "Shell32.dll" )
      pFunc := GetProcAddress( nDll, "ShellExecute" )

      ? CallDll( pFunc, 0, "open", cFile, NIL, NIL, 1 )

      DllUnload( nDll )
   RETURN
```

CDoW()

Returns the name of a week day from a date.

Syntax

```
CDoW( <dDate> ) --> cDayName
```

Arguments

<dDate>

This is an expression returning a value of data type Date.

Return

The function returns a character string holding the weekday name of *<dDate>*. When *<dDate>* is an empty date, the function returns a null string ("").

Description

The CDoW() function converts a date value into the name of the corresponding week day. Use CMonth() to obtain the month name as character value. Both functions are used to format date values in a textual way.

Note: xHarbour's national language support allows for returning the name of a weekday in various languages. Refer to HB_LangSelect() for selecting a language module.

Info

See also:	CMonth(), CtoD(), Date(), Day(), DoW(), HB_LangSelect(), Month(), Year()
Category:	Conversion functions, Date and time
Source:	rtl\datec.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates the conversion of a date value into text.

   PROCEDURE Main
      LOCAL dDate := Date()
      LOCAL cText := "Today is "

      cText += CDoW( dDate ) + ", "
      cText += LTrim( Str( Day( dDate ) ) ) + ". "
      cText += CMonth( dDate ) + " "
      cText += LTrim( Str( Year( dDate ) ) )

      ? cText

      // result: Today is Tuesday, 24. January 2006
   RETURN
```

Ceiling()

Rounds a decimal number to the next higher integer.

Syntax

```
Ceiling( <nValue> ) --> nInteger
```

Arguments

`<nValue>`

Any numeric value can be passed.

Return

The return value is the next higher integer value of *<nValue>*. If *<nValue>* is an integer already, it is returned unchanged.

Info

See also:	Floor(), Round()
Category:	CT:Math, Mathematical functions
Source:	ct\ctmath2.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows results of function Ceiling().

   PROCEDURE Main

      ? Ceiling( -2.0 )       // result: -2

      ? Ceiling( -2.00001 )   // result: -2

      ? Ceiling(  2.0 )       // result: 2

      ? Ceiling(  2.00001 )   // result: 3

   RETURN
```

Celsius()

Converts degrees Fahrenheit to Celsius.

Syntax

```
Celsius( <nFahrenheit> ) --> nCelsius
```

Arguments

`<nFahrenheit>`

This is a numeric value specifying degrees Fahrenheit.

Return

The function returns degrees Celsius as a numeric value.

Info

See also:	Fahrenheit()
Category:	CT:NumBits, Numbers and Bits
Source:	ct\num1.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Center()

Returns a string for centered display.

Syntax

```
Center( <cString>   , ;
       [<nLength>]  , ;
       [<cPadChar>] , ;
       [<lBothSides>] ) --> cCenteredString
```

Arguments

`<cString>`

This is a character string to fill with padding characters.

`<nLength>`

This optional numeric value specifies the number of characters that fit into one line for display. The default value is MaxCol()+1.

`<cPadChar>`

A single character used to pad *<cString>* with. It defaults to a space character (Chr(32)).

`<lBothSides>`

This parameter defaults to .F. (false) so that *<cString>* is only filled on the left side up to the length *<nLength>*. When .T. (true) is passed, the string is padded on the left and right side.

Return

The function returns a string padded with *<cPadChar>* that displays centered.

Description

Center() fills a string with *<cFillCharacter>* so it is centered within a space of *<nLength>* characters. By default the string is only filled to the left. If *<nLength>* is not specified the string will be centered between the current cursor position and MaxCol()+1.

Info

See also:	PadLeft(), PadRight()
Category:	CT:String manipulation, Character functions
Source:	ct\ctmisc.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays a series of character strings centered
// on a 40 column wide area of the screen and fills the remainder
// with dots.

   PROCEDURE Main
      LOCAL aLine := { "The", "xHarbour", "compiler", "is", "more!" }
      LOCAL cLine

      CLS

      FOR EACH cLine IN aLine
        ? Center( cLine, 40, ".", .T. )
      NEXT
```

```
   ** result:
   // ..................The...................
   // ................xHarbour................
   // ................compiler................
   // ...................is...................
   // .................more!..................
RETURN
```

CFTSAdd()

Adds a text string entry to a Full Text Search index file.

Syntax

```
CFTSAdd( <nFileHandle>, <bText> ) --> nIndexEntry
```

Arguments

`<nFileHandle>`

This is the numeric file handle of the Full Text Search index file to add a new entry to. The file handle is returned from CFTSOpen() or CFTSCrea().

`<bText>`

The second parameter is a code block which returns the character string to add to the Full Text Search index file.

Return

The function returns a numeric value. Positive values identify the ordinal number of the new index entry, while negative numbers are error codes.

Description

CFTSAdd() is a synonym for function HS_Add(). Refer to HS_Add() for more information.

Info

See also:	HS_Delete(), HS_Replace()
Category:	CFTS functions
Source:	rdd\hsx\cftsfunc.c
LIB:	lib\xhb.lib
DLL:	dll\xhbdll.dll

CFTSClose()

Closes a Full Text Search index file.

Syntax

```
CFTSClose( <nFileHandle> ) --> nErrorCode
```

Arguments

<nFileHandle>

This is the numeric file handle of the Full Text Search index file to add a new entry to. The file handle is returned from CFTSOpen() or CFTSCrea().

Return

The function returns a numeric value. Positive 1 indicates that the file is successfully closed, while negative numbers identify an error condition.

Description

CFTSClose() is a synonym for function HS_Close(). Refer to HS_Close() for more information.

Info

See also: HS_Add(), HS_Index(), HS_Next(), HS_Open(), HS_Set()
Category: CFTS functions
Source: rdd\hsx\cftsfunc.c
LIB: lib\xhb.lib
DLL: dll\xhbdll.dll

CFTSCrea()

Creates a new Full Text Search index file.

Syntax

```
CFTSCrea( <cFileName>       , ;
          <nBufferSize>     , ;
          <nKeySize>        , ;
          <lCaseInsensitive>, ;
          <nFilterSet>       ) --> nFileHandle
```

Arguments

`<cFileName>`

This is a character string holding the name of the Full Text Search index file to create. It must include path and file extension. If the path is omitted from *<cFileName>*, the file is created in the current directory. If no file extension is given, the extension .HSX is used.

`<nBufferSize>`

This is a numeric value specifying the memory buffer size in kB to be used by CFTS_*() functions for this file (the value 10 means 10240 bytes).

`<nKeySize>`

An numeric value of 1, 2 or 3 must be passed. It defines the size of a single index entry in the Full Text Search index file. The larger *<nKeySize>* is, the more unique index entries can be stored, reducing the possibility of false positives when searching a Full Text Search index file. The size of an index entry is 16 bytes (1), 32 bytes (2) or 64 bytes (3).

`<lCaseInsensitive>`

This is a logical value. When .T. (true) is passed, the index is case insensitive, otherwise it is case sensitive.

`<nFilterSet>`

This is a numeric value of 1 or 2 defining the filter set to use with the Full Text Search index file. When *<nFilterSet>* is 1, all non-printable characters, such as Tab, or carriage return/line feed, are treated as blank spaces and the high-order bit is ignored for characters (7-bit character set).

Specifying 2 recognizes all characters by their ASCII value.

Return

The function returns a positive numeric value when the Full Text Search index file is successfully created. This is the handle to the new file, which must be used with other CFTS_*() functions. A negative value indicates an error.

Description

CFTSCrea() is a synonym for function HS_Create(). Refer to HS_Create() for more information.

Info

See also:	HS_Add(), HS_Index(), HS_Next(), HS_Open(), HS_Set()
Category:	CFTS functions
Source:	rdd\hsx\cftsfunc.c
LIB:	lib\xhb.lib
DLL:	dll\xhbdll.dll

CFTSDelete()

Marks an index entry as deleted in a Full Text Search index file.

Syntax

```
CFTSDelete( <nFileHandle>, <nIndexEntry> ) --> nErrorCode
```

Arguments

<nFileHandle>

This is the numeric file handle of the Full Text Search index file to mark an entry as deleted. The file handle is returned from CFTSOpen() or CFTSCrea().

<nIndexEntry>

The ordinal position of the index entry to delete must be specified as a numeric value.

Return

The function returns 1 when the index entry is successfully removed from the file, or a negative numeber indicating an error condition:

Description

CFTSDelete() is a synonym for function HS_Delete(). Refer to HS_Delete() for more information.

Info

See also:	DbDelete(), HS_IfDel(), HS_Replace(), HS_Undelete()
Category:	CFTS functions
Source:	rdd\hsx\cftsfunc.c
LIB:	lib\xhb.lib
DLL:	dll\xhbdll.dll

CFTSIfDel()

Checks if a Full Text Search index entry is marked as deleted.

Syntax

```
CFTSIfDel( <nFileHandle>, <nIndexEntry> ) --> nErrorCode
```

Arguments

`<nFileHandle>`

This is the numeric file handle of the Full Text Search index file to check. The file handle is returned from CFTSOpen() or CFTSCrea().

`<nIndexEntry>`

This is a numeric value specifying the ordinal position of the index entry to check.

Return

The function returns 1 when the index entry *<nIndexEntry>* is marked as deleted, zero when the deletion flag is not set, or a negative numeber indicating an error condition.

Description

CFTSIfDel() is a synonym for function HS_IfDel(). Refer to HS_IfDel() for more information.

Info

See also:	Deleted(), HS_Delete(), HS_Undelete()
Category:	CFTS functions
Source:	rdd\hsx\cftsfunc.c
LIB:	lib\xhb.lib
DLL:	dll\xhbdll.dll

CFTSNext()

Searches a Full Text Search index file for a matching index entry.

Syntax

```
CFTSNext( <nFileHandle> ) --> nRecno
```

Arguments

<nFileHandle>

This is the numeric file handle of the Full Text Search index file to use for a search. The file handle is returned from CFTSOpen() or CFTSCrea().

Return

The function returns a numeric value. When it is greater than zero, it represents the ordinal position of the index entry found in the file. The value zero is returned when no match is found, and a negative numeber indicates an error condition.

Description

CFTSNext() is a synonym for function HS_Next(). Refer to HS_Next() for more information.

Info

See also:	HS_Set(), HS_Verify()
Category:	CFTS functions
Source:	rdd\hsx\cftsfunc.c
LIB:	lib\xhb.lib
DLL:	dll\xhbdll.dll

CFTSOpen()

Opens a Full Text Search index file.

Syntax

```
CFTSOpen( <cFileName>   , ;
          <nBufferSize>, ;
          <nOpenMode>    ) --> nFileHandle
```

Arguments

<cFileName>

This is a character string holding the name of the Full Text Search index file to open. It must include path and file extension. If the path is omitted from *<cFileName>*, the file is searched in the current directory. If no file extension is given, the extension .HSX is used.

<nBufferSize>

This is a numeric value specifying the memory buffer size in kB to be used by other CFTS_*() functions for this file (the value 10 means 10240 bytes).

<nOpenMode>

This numeric parameter specifies how the Full Text Search index file is opened. Possible values for *<nOpenMode>* are:

Open modes for Full Text Search index files

Value	Description
0	READ-WRITE + SHARED
1	READ-WRITE + EXCLUSIVE
2	READ-ONLY + SHARED
3	READ-ONLY + EXCLUSIVE4

Return

The function returns a numeric value greater or equal to zero when the Full Text Search index file can be opened. This value is a handle to the Full Text Search index file and must be used with other CFTS_*() functions.

A negative return value indicates an error condition.

Description

CFTSOpen() is a synonym for function HS_Open(). Refer to HS_Open() for more information.

Info

See also:	HS_Add(), HS_Close(), HS_Create(), HS_Next(), HS_Set()
Category:	CFTS functions
Source:	rdd\hsx\cftsfunc.c
LIB:	lib\xhb.lib
DLL:	dll\xhbdll.dll

CFTSRecn()

Returns the number of index entries in a Full Text Search index file.

Syntax

```
CFTSRecn( <nFileHandle> ) --> nKeyCount
```

Arguments

<nFileHandle>

This is the numeric file handle of the Full Text Search index file to count the index entries in. The file handle is returned from CFTSOpen() or CFTSCrea().

Return

The function returns a numeric value greater or equal to zero representing the number of index entries contained in the HiPer-SEEK index. A negative value indicates an error condition:

Description

CFTSRecn() is a synonym for function HS_KeyCount(). Refer to HS_KeyCount() for more information.

Info

See also:	Hs_Filter(), OrdKeyCount()
Category:	CFTS functions
Source:	rdd\hsx\cftsfunc.c
LIB:	lib\xhb.lib
DLL:	dll\xhbdll.dll

CFTSReplac()

Changes a Full Text Search index entry.

Syntax

```
CFTSReplac( <nFileHandle>, ;
            <cNewIndex>   , ;
            <nIndexEntry>  ) --> nErrorCode
```

Arguments

<nFileHandle>

This is the numeric file handle of the Full Text Search index file to replace an index entry in. The file handle is returned from CFTSOpen() or CFTSCrea().

<cNewIndex>

This is a character string holding the new value for the index entry specified with *<nIndexEntry>*.

<nIndexEntry>

The ordinal position of the index entry to change must be specified as a numeric value.

Return

The function returns numeric 1 whan the index entry is successfully changed, or a negative number as error code.

Description

CFTSReplac() is a synonym for function HS_Replace(). Refer to HS_Replace() for more information.

Info

See also:	HS_Add(), HS_Delete(), REPLACE
Category:	CFTS functions
Source:	rdd\hsx\cftsfunc.c
LIB:	lib\xhb.lib
DLL:	dll\xhbdll.dll

CFTSSet()

Defines a search string for subsequent CFTSNext() calls.

Syntax

```
CFTSSet( <nFileHandle>, <cSearch> ) --> nIndexEntry
```

Arguments

<nFileHandle>

This is the numeric file handle of the Full Text Search index file to find *<cSearch>* in. The file handle is returned from CFTSOpen() or CFTSCrea().

<cSearch>

This is a character string holding the search string for subsequent HS_Next() calls.

Return

The function returns numeric 1 whan the search string is successfully defined, or a negative number as error code.

Description

CFTSSet() is a synonym for function HS_Set(). Refer to HS_Set() for more information.

Info

See also: HS_Next()
Category: CFTS functions
Source: rdd\hsx\cftsfunc.c
LIB: lib\xhb.lib
DLL: dll\xhbdll.dll

CFTSUndel()

Removes the deletion mark from an index entry in a Full Text Search index file.

Syntax

```
CFTSUndel( <nFileHandle>, <nIndexEntry> ) --> nErrorCode
```

Arguments

<nFileHandle>

This is the numeric file handle of the Full Text Search index file containing index entries marked as deleted. The file handle is returned from CFTSOpen() or CFTSCrea().

<nIndexEntry>

The ordinal position of the index entry to mark as undeleted must be specified as a numeric value.

Return

The function returns 1 when the deletion mark is successfully removed from the index entry, or a negative numeber indicating an error condition.

Description

CFTSUndel() is a synonym for function HS_Undelete(). Refer to HS_Undelete() for more information.

Info

See also: DbRecall(), HS_Delete(), HS_IfDel(), HS_Replace()
Category: CFTS functions
Source: rdd\hsx\cftsfunc.c
LIB: lib\xhb.lib
DLL: dll\xhbdll.dll

CFTSVeri()

Verifies a CFTSNext() match against the index key.

Syntax

```
CFTSVeri( <xIndexKey>, <cSearch> ) --> lMatched
```

Arguments

<xIndexKey>

This is either a code block holding the index key, or a character string holding the index key expression. A code block is recommended, since it does not need to be macro compiled.

<cSearch>

The searched value must be passed as a character string.

Return

The function returns .T. (true) when *<cSearch>* is contained anywhere in the result of the code block, or index value, otherwise .F. (false) is returned.

Description

CFTSVeri() is a synonym for function HS_Verify(). Refer to HS_Verify() for more information.

Info

See also:	HS_Next()
Category:	CFTS functions
Source:	rdd\hsx\cftsfunc.c
LIB:	lib\xhb.lib
DLL:	dll\xhbdll.dll

CFTSVers()

Returns version information for HiPer-SEEK functions.

Syntax

```
CFTSVers() -->  cVersionInfo
```

Return

The function returns a character string holding version information of HiPer-SEEK functions.

Description

CFTSVers() is a synonym for function HS_Version(). Refer to HS_Version() for more information.

Info

See also:	HS_Create(), HS_Index(), Version()
Category:	CFTS functions
Source:	rdd\hsx\cftsfunc.c
LIB:	lib\xhb.lib
DLL:	dll\xhbdll.dll

CharAdd()

Creates a string from the sum of ASCII codes of two strings.

Syntax

```
CharAdd( <cString1> , <cString2> ) --> cResult
```

Arguments

`<cString1> and <cString2>`

These are two character strings whose ASCII codes are added in the result string.

Return

The function returns a string by adding the ASCII codes of the individual characters of both input strings.

Description

The function operates until all characters of *<cString1>* are processed. It distinguishes the following situations:

1) **Len(*<cString1>*) == Len(*<cString2>*)**

   When both input strings are of the same length, the ASCII code of each character in *<cString2>* is added to the ASCII code of the corresponding character in *<cString1>*.

2) **Len(*<cString1>*) > Len(*<cString2>*)**

   When the last character of *<cString2>* is reached, the function starts over with the first character of *<cString2>*, until the last character of *<cString1>* is processed.

3) **Len(*<cString1>*) < *Len(<cString2>*)**

   The function returns when the last character of *<cString1>* is processed.

Info

See also:	AddASCII(), CharAND(), CharOR(), CharSub(), CharXOR()
Category:	CT:String manipulation, Character functions
Source:	ct\charop.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The function displays results of CharAdd()

   PROCEDURE Main
      LOCAL cString := "ABC"

      ? CharAdd( cString, Chr(3) )      // result: DEF

      ? CharAdd( cString, Space(3) )    // result: abc
   RETURN
```

CharAND()

Binary ANDs the ASCII codes of characters in two strings.

Syntax

```
CharAND( <cString1>, <cString2> ) --> cResult
```

Arguments

`<cString1> and <cString2>`

These are two character strings whose ASCII codes are ANDed in the result string.

Return

The function returns a string by ANDing the ASCII codes of the individual characters of both input strings.

Description

The function operates until all characters of *<cString1>* are processed. It distinguishes the following situations:

1) **Len(*<cString1>*) == Len(*<cString2>*)**

   When both input strings are of the same length, the ASCII code of each character in *<cString2>* is ANDed with the ASCII code of the corresponding character in *<cString1>*.

2) **Len(*<cString1>*) > Len(*<cString2>*)**

   When the last character of *<cString2>* is reached, the function starts over with the first character of *<cString2>*, until the last character of *<cString1>* is processed.

3) **Len(*<cString1>*) < *Len(* *<cString2>*)**

   The function returns when the last character of *<cString1>* is processed.

Note: CharAnd() exists for compatibility reasons. It is superseeded by xHarbour's binary AND operator.

Info

See also:	& (bitwise AND), CharNOT(), CharOR(), CharXOR()
Category:	CT:String manipulation, Bitwise functions, Character functions
Source:	ct\charop.c
LIB:	xhb.lib
DLL:	xhbdll.dll

CharEven()

Extracts characters at even positions from a string.

Syntax

```
CharEven( <cString> ) --> cResult
```

Arguments

`<cString>`

This is the character string to process.

Return

The function returns a string containing all characters at even positions in the input string *<cString>*. This is useful, for example, to extract the video bytes from a SaveScreen() string.

Info

See also:	CharMix(), CharOdd(), ScreenMix(), ScreenStr(), StrScreen()
Category:	CT:String manipulation, Character functions, Screen functions
Source:	ct\charevod.c
LIB:	xhb.lib
DLL:	xhbdll.dll

CharHist()

Creates a histogram of characters in a character string

Syntax

```
CharHist( [<cString>] ) --> aHistogram
```

Arguments

<cString>

This is the character string to process. It defaults to a null string ("").

Return

The function returns a one dimensional array of 256 elements. Each element hold a numeric value which indicates how often a character is found in *<cString>*. The numeric ASCII code of a character plus 1 points to the array element holding the result for a single character.

Info

See also: CharList(), CharNoList(), CharSList(), CharSort()
Category: CT:String manipulation, Character functions, xHarbour extensions
Source: ct\misc1.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example creates a histogram of a character string
// and displays which character appears how often.

   PROCEDURE Main
      LOCAL cString := "The excellent xHarbour compiler"
      LOCAL aHist, cChar, nCount

      aHist := CharHist( cString )

      // remove duplicate characters and sort
      cString := CharSList( cString )

      // display the histogram
      FOR EACH cChar IN cString
         // the ASCII code of a character points to the histogram array
         nCount := aHist[ Asc(cChar)+1 ]

         IF nCount == 1
            ? '"' + cChar + '" = 1x'

         ELSEIF nCount > 0
            ? '"' + cChar + '" = ' + LTrim(Str(nCount)) +"x"

         ENDIF
      NEXT

      /* Result:
         " " = 3x
         "H" = 1x
         "T" = 1x
         "a" = 1x
```

```
            "b" = 1x
            "c" = 2x
            "e" = 5x
            "h" = 1x
            "i" = 1x
            "l" = 3x
            "m" = 1x
            "n" = 1x
            "o" = 2x
            "p" = 1x
            "r" = 3x
            "t" = 1x
            "u" = 1x
            "x" = 2x
        */
   RETURN
```

CharList()

Removes duplicate characters from a string.

Syntax

```
CharList( <cString> ) --> cResult
```

Arguments

<cString>

This is the character string to process.

Return

The function removes all duplicate characters from the input string and returns the result.

Info

See also: CharHist(), CharNoList(), CharOne(), CharSList(), CharSort()
Category: CT:String manipulation, Character functions
Source: ct\charlist.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example shows results of CharList()

   PROCEDURE Main
      LOCAL cString1 := "xHarbour compiler"
      LOCAL cString2 := "aabbbcccc"

      ? CharList( cString1 )   // result: xHarbou cmpile

      ? CharList( cString2 )   // result: abc
   RETURN
```

CharMirr()

Reverses the order of characters in a string.

Syntax

```
CharMirr( <cString>, [<lNoSpaces>] ) --> cResult
```

Arguments

`<cString>`

This is the character string to process.

`<lNoSpaces>`

This parameter defaults to .F. (false) so that trailing spaces of the input string are included at the beginning of the result string. Passing .T. (true) causes the function to ignore trailing spaces of *<cString>*, and leave them at the end of the result string.

Return

The function reverses the order of the characters in *<cString>* and returns the result.

Info

See also: CharList()
Category: CT:String manipulation, Character functions
Source: ct\charmirr.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays results of CharMirr()

   PROCEDURE Main
      LOCAL cString1 := "abcdef"
      LOCAL cString2 := "xHarbour  "

      ? ">"+CharMirr( cString1 )+"<"       // result: >fedcba<

      ? ">"+CharMirr( cString2 )+"<"       // result: >  ruobraHx<
      ? ">"+CharMirr( cString2, .T. )+"<"  // result: >ruobraHx  <
   RETURN
```

CharMix()

Merges the characters of two strings.

Syntax

```
CharMix( <cString1>, <cString2> ) --> cResult
```

Arguments

`<cString1> and <cString2>`

These are two character strings whose characters are merged in the result string.

Return

The function returns a string containing the characters of *<cString1>* at odd positions and those of *<cString1>* at even positions. This is useful, for example, to build a SaveScreen() string from a text string and video bytes.

Description

The function operates until all characters of *<cString1>* are processed. It distinguishes the following situations:

1) **Len(*<cString1>*) == Len(*<cString2>*)**

   When both input strings are of the same length, each character in *<cString1>* is followed by the corresponding character in *<cString2>*.

2) **Len(*<cString1>*) > Len(*<cString2>*)**

   When the last character of *<cString2>* is reached, the function starts over with the first character of *<cString2>*, until the last character of *<cString1>* is processed.

3) **Len(*<cString1>*) < *Len(* *<cString2>*)**

   The function returns when the last character of *<cString1>* is processed.

Info

See also: CharEven(), CharOdd(), Expand(), ScreenMix()
Category: CT:String manipulation, Character functions
Source: ct\charmix.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example manipulates a savescreen string for changing
// the color periodically until a key is pressed

   PROCEDURE Main
      LOCAL cScreen, cText, cVideo

      CLS
      @ 2,2 SAY "Changing video bytes" COLOR "W+/B"

      cScreen := SaveScreen( 2, 2, 2, 22 )
      cText   := CharOdd( cScreen )
      cVideo  := Chareven( cScreen )

      DO WHILE Inkey(.5) == 0
         cVideo := CharXOR( cVideo, chr(8) )
```

```
        cScreen := CharMix( cText, cVideo )
        RestScreen( 2, 2, 2, 22, cScreen )
     ENDDO
  RETURN
```

CharNoList()

Returns a string containing all characters not included in a string.

Syntax

```
CharNoList( [<cString>] ) --> cNotInString
```

Arguments

<cString>

This is an optional character string to process. It defaults to a null string ("").

Return

The function returns a character string holding all characters from Chr(0) to Chr(255) not included in the input string. If the function is called without parameter, the returned string contains all 256 ASCII characters.

Info

See also:	CharList(), CharOne(), CharSList(), CharSort()
Category:	CT:String manipulation, Character functions
Source:	ct\charlist.c
LIB:	xhb.lib
DLL:	xhbdll.dll

CharNOT()

Binary NOTs the ASCII codes of characters in two strings.

Syntax

```
CharNOT( <cString> ) --> cResult
```

Arguments

`<cString1> and <cString2>`

These are two character strings whose ASCII codes are NOTed in the result string.

Return

The function returns a string by NOTing the ASCII codes of the individual characters of both input strings.

Description

The function operates until all characters of *<cString1>* are processed. It distinguishes the following situations:

1) **Len(*<cString1>*) == Len(*<cString2>*)**

   When both input strings are of the same length, the ASCII code of each character in *<cString2>* is ANDed with the ASCII code of the corresponding character in *<cString1>*.

2) **Len(*<cString1>*) > Len(*<cString2>*)**

   When the last character of *<cString2>* is reached, the function starts over with the first character of *<cString2>*, until the last character of *<cString1>* is processed.

3) **Len(*<cString1>*) < *Len(<cString2>*)**

   The function returns when the last character of *<cString1>* is processed.

Info

See also:	CharAND(), CharOR(), CharXOR(), Complement()
Category:	CT:String manipulation, Bitwise functions, Character functions
Source:	ct\charop.c
LIB:	xhb.lib
DLL:	xhbdll.dll

CharOdd()

Extracts characters at odd positions from a string.

Syntax

```
CharOdd( <cString> ) --> cResult
```

Arguments

<cString>

This is the character string to process.

Return

The function returns a string containing all characters at odd positions in the input string *<cString>*. This is useful, for example, to extract the visible, or textual, characters from a SaveScreen() string.

Info

See also:	CharEven(), CharMix(), ScreenMix(), ScreenStr(), StrScreen()
Category:	CT:String manipulation, Character functions, Screen functions
Source:	ct\charevod.c
LIB:	xhb.lib
DLL:	xhbdll.dll

CharOne()

Removes duplicate adjacent characters from a string.

Syntax

```
CharOne( [<cDelete>], <cString> ) --> cResult
```

Arguments

<cDelete>

Passing a character string for *<cDelete>* restricts the search to the characters contained in it. The default removes all duplicate adjacent characters from *<cString>*.

<cString>

This is the character string to process.

Return

The function searches for duplicate adjacent characters in *<cString>* and removes them in the result string.

Note: the function differs from CharList() which removes **all** duplicates. CharOne() removes only **adjacent** duplicates.

Info

See also:	CharList(), CharOnly(), CharRem(), WordOne()
Category:	CT:String manipulation, Character functions
Source:	ct\charone.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example outlines the difference between CharOne()
// and CharList()

   PROCEDURE Main
      LOCAL cString := "11ab12bb33cc"

      ? CharOne( "abc123", cString )  // result: 1ab12b3c

      ? CharList( cString )           // result: 1ab23c

   RETURN
```

CharOnly()

Removes all characters but the specified ones from a string

Syntax

```
CharOnly( <cRemaining>, <cString> ) --> cResult
```

Arguments

<cRemaining>

This character string defines the characters that are to remain in the result string.

<cString>

This is the character string to process.

Return

The function removes all characters from *<cString>* except those contained in *<cRemaining>* and returns the result.

Info

See also:	CharOne(), CharRem(), WordOnly()
Category:	CT:String manipulation, Character functions
Source:	ct\charonly.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example extracts digits from formatted phone numbers

   PROCEDURE Main
      LOCAL cDigits := "0123456789"

      ? CharOnly( cDigits, "(555) 45 67 788" )     // result: 5554567788

      ? CharOnly( "0123456789", "555 / 345 9876" ) // result: 5553459876
   RETURN
```

CharOR()

Binary ORs the ASCII codes of characters in two strings.

Syntax

```
CharOR( <cString1>, <cString2> ) --> cResult
```

Arguments

`<cString1> and <cString2>`

These are two character strings whose ASCII codes are ORed in the result string.

Return

The function returns a string by ORing the ASCII codes of the individual characters of both input strings.

Description

The function operates until all characters of *<cString1>* are processed. It distinguishes the following situations:

1) **Len(*<cString1>*) == Len(*<cString2>*)**

   When both input strings are of the same length, the ASCII code of each character in *<cString2>* is XORed with the ASCII code of the corresponding character in *<cString1>*.

2) **Len(*<cString1>*) > Len(*<cString2>*)**

   When the last character of *<cString2>* is reached, the function starts over with the first character of *<cString2>*, until the last character of *<cString1>* is processed.

3) **Len(*<cString1>*) < *Len(* *<cString2>*)**

   The function returns when the last character of *<cString1>* is processed.

Note: CharOR() exists for compatibility reasons. It is superseeded by xHarbour's binary OR operator.

Info

See also: | (bitwise or), CharAND(), CharNOT(), CharXOR()
Category: CT:String manipulation, Bitwise functions, Character functions
Source: ct\charop.c
LIB: xhb.lib
DLL: xhbdll.dll

CharPack()

Compresses a string.

Syntax

```
CharPack( <cString> ) --> cCompressed
```

Arguments

<cString>

This is the character string to compress.

Return

The function returns a compressed string. It can be uncompressed with CharUnpack().

Info

See also: CharUnpack()
Category: CT:String manipulation, Character functions
Source: ct\pack.c
LIB: xhb.lib
DLL: xhbdll.dll

CharRela()

Tests if two substrings in two strings have the same position.

Syntax

```
CharRela( <cSearch1>, ;
          <cString1>, ;
          <cSearch2>, ;
          <cString2>  ) --> nPosition
```

Arguments

<cSearch1>

This is a character string to search in *<cString1>*.

<cString1>

This is the character string to find *<cSearch1>* in.

<cSearch2>

This is a character string to search in *<cString2>*.

<cString2>

This is the character string to find *<cSearch2>* in.

Return

The function searches *<cSearch1>* in *<cString1>*, and *<cSearch2>* in *<cString2>*. If both search strings are found at the same position in both strings, the function returns it as a numeric value. The return value is zero when no match is found.

Info

See also: CharRelRep(), CharSort()
Category: CT:String manipulation, Character functions
Source: ct\relation.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example compares the position of letters in a string
// with the position of digits in a second string.

   PROCEDURE Main
      LOCAL cString1 := "abbaacc"
      LOCAL cString2 := "0011011"

      ? CharRela( "a", cString1, "1", cString2 ) // result: 4
      ? CharRela( "b", cString1, "1", cString2 ) // result: 3
      ? CharRela( "c", cString1, "1", cString2 ) // result: 6
      ? CharRela( "c", cString1, "0", cString2 ) // result: 0
   RETURN
```

CharRelRep()

Replaces characters if two substrings in two strings have the same position.

Syntax

```
CharRelRep( <cSearch1>, ;
            <cString1>, ;
            <cSearch2>, ;
            <cString2>, ;
            <cReplace>  ) --> cResult
```

Arguments

<cSearch1>

This is a character string to search in *<cString1>*.

<cString1>

This is the character string to find *<cSearch1>* in.

<cSearch2>

This is a character string to search in *<cString2>*.

<cString2>

This is the character string to find *<cSearch2>* in.

<cReplace>

This is the character string which replaces characters in *<cString2>* at all matching positions.

Return

The function searches *<cSearch1>* in *<cString1>*, and *<cSearch2>* in *<cString2>*. If both search strings are found at the same position in both strings, the function replaces characters in *<cString2>* until no more match is found. The return value *<cString2>* when no match is found at all.

Info

See also:	CharRela()
Category:	CT:String manipulation, Character functions
Source:	ct\relation.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example replaces digits with "X" based on the position of letters
// in a comparison string.

   PROCEDURE Main
      LOCAL cStr1 := "abbaacc"
      LOCAL cStr2 := "0011011"

      ? CharRelRep( "a", cStr1, "1", cStr2, "X" ) // result: 001X011
      ? CharRelRep( "b", cStr1, "1", cStr2, "X" ) // result: 00X1011
      ? CharRelRep( "c", cStr1, "1", cStr2, "X" ) // result: 00110XX
      ? CharRelRep( "c", cStr1, "0", cStr2, "X" ) // result: 0011011
   RETURN
```

CharRem()

Deletes specified characters from a string.

Syntax

```
CharRem( <cDelete>, <cString> ) --> cResult
```

Arguments

<cDelete>

This is a character string holding the list of characters to delete from *<cString>*.

<cString>

This is the character string to process.

Return

The function removes all characters contained in *<cDelete>* from the input string *<cString>* and returns the result.

Info

See also:	CharOne(), CharOnly()
Category:	CT:String manipulation, Character functions
Source:	ct\charonly.c
LIB:	xhb.lib
DLL:	xhbdll.dll

CharRepl()

Searches a list of characters and replaces them with a corresponding list.

Syntax

```
CharRepl( <cSearch> , ;
          <cString> , ;
          <cReplace>, ;
         [<lOnePass>] ) --> cResult
```

Arguments

`<cSearch>`

This is a character string holding the list of characters to search for in *<cString>*.

`<cString>`

This is the character string to process.

`<cReplace>`

This is a character string holding the replacemnt list of characters. It must be of the same length as *<cSearch>*. If characters of *<cSearch>* are found in *<cString>*, they are replaced with the corresponding character of *<cReplace>*

`<lOnePass>`

This parameter defaults to .F. (false) so that the search and replace operation is repeated until no more match is found. When *<lOnePass>* is .T. (true), the function processed *<cString>* only once.

Return

The function returns a string where all characters contained in *<cSearch>* are replaced with the corresponding characters of *<cReplace>*. The string *<cString>* is returned unchanged when no character of *<cSearch>* is found.

Info

See also:	PosRepl(), RangeRepl(), WordRepl(), WordToChar()
Category:	CT:String manipulation, Character functions
Source:	ct\charrepl.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows results of CharRepl()

   PROCEDURE Main
      LOCAL cString := "xHarbour compiler"

      ? CharRepl( "aeiou", cString, "AEIOU" ) // result: xHArbOUr cOmpIlEr

      ? CharRepl( "xmr"  , cString, "123" )   // result: 1Ha3bou3 co2pile3
   RETURN
```

CharRLL()

Rotates bits in a character string to the left.

Syntax

```
CharRLL( <cString>, <nShift> ) --> cResult
```

Arguments

<cString>

This is the character string to process.

<nShift>

This is a numeric integer value indicating how many places the bits of each character of *<cString>* are rotated to the left.

Return

The function returns a string by rotating the bits of the individual characters of *<cString>* for *<nShift>* places to the left. The highest bit replaces the lowest bit for each rotation.

Info

See also: CharAND(), CharNOT(), CharOR(), CharRLR(), CharSHL(), CharSHR(), CharXOR()
Category: CT:String manipulation, Bitwise functions, Character functions, xHarbour extensions
Source: ct\charop.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates the effect of rotating bits in
// a two-character string, by displaying the ASCII codes and
// their binary representation before and after the left rotation.

   PROCEDURE Main
      LOCAL cString := "AB"
      LOCAL cShift
      LOCAL nAsc1, nAsc2

      ? nAsc1 := Asc( cString[1] )      // result: 65
      ? nAsc2 := Asc( cString[2] )      // result: 66

      ? NtoC( nAsc1, 2, 8, "0" )        // result: 01000001
      ? NtoC( nAsc2, 2, 8, "0" )        // result: 01000010

      cShift := CharRLL( cString, 2 )

      ? nAsc1 := Asc( cShift[1] )       // result:  5
      ? nAsc2 := Asc( cShift[2] )       // result:  9
      ? NtoC( nAsc1, 2, 8, "0" )        // result: 00000101
      ? NtoC( nAsc2, 2, 8, "0" )        // result: 00001001
   RETURN
```

CharRLR()

Rotates bits in a character string to the right.

Syntax

```
CharRLR( <cString>, <nShift> ) --> cResult
```

Arguments

<cString>

This is the character string to process.

<nShift>

This is a numeric integer value indicating how many places the bits of each character of *<cString>* are rotated to the right.

Return

The function returns a string by rotating the bits of the individual characters of *<cString>* for *<nShift>* places to the right. The lowest bit replaces the highest bit for each rotation.

Info

See also:	CharAND(), CharNOT(), CharOR(), CharRLL(), CharSHL(), CharSHR(), CharXOR()
Category:	CT:String manipulation, Bitwise functions, Character functions, xHarbour extensions
Source:	ct\charop.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates the effect of rotating bits in
// a two-character string, by displaying the ASCII codes and
// their binary representation before and after the right rotation.

   PROCEDURE Main
      LOCAL cString := "AB"
      LOCAL cShift
      LOCAL nAsc1, nAsc2

      ? nAsc1 := Asc( cString[1] )      // result: 65
      ? nAsc2 := Asc( cString[2] )      // result: 66

      ? NtoC( nAsc1, 2, 8, "0" )        // result: 01000001
      ? NtoC( nAsc2, 2, 8, "0" )        // result: 01000010

      cShift := CharRLR( cString, 2 )

      ? nAsc1 := Asc( cShift[1] )       // result:  80
      ? nAsc2 := Asc( cShift[2] )       // result: 144
      ? NtoC( nAsc1, 2, 8, "0" )        // result: 01010000
      ? NtoC( nAsc2, 2, 8, "0" )        // result: 10100000
   RETURN
```

CharSHL()

Shifts bits in a character string to the left.

Syntax

```
CharSHL( <cString>, <nShift> ) --> cResult
```

Arguments

<cString>

This is the character string to process.

<nShift>

This is a numeric integer value indicating how many places the bits of each character of *<cString>* are shifted to the left.

Return

The function returns a string by shifting the bits of the individual characters of *<cString>* for *<nShift>* places to the left. The highest bit is discarded and the lowest bit is set to zero.

Info

See also: <<, CharAND(), CharNOT(), CharOR(), CharRLL(), CharRLR(), CharSHR(), CharXOR(), HB_BitShift()
Category: CT:String manipulation, Bitwise functions, Character functions, xHarbour extensions
Source: ct\charop.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates the effect of shifting bits in
// a two-character string, by displaying the ASCII codes and
// their binary representation before and after the left shift
// operation. Note that the most significant bit is lost.

   PROCEDURE Main
      LOCAL cString := "AB"
      LOCAL cShift
      LOCAL nAsc1, nAsc2

      ? nAsc1 := Asc( cString[1] )      // result: 65
      ? nAsc2 := Asc( cString[2] )      // result: 66

      ? NtoC( nAsc1, 2, 8, "0" )        // result: 01000001
      ? NtoC( nAsc2, 2, 8, "0" )        // result: 01000010

      cShift := CharSHL( cString, 2 )

      ? nAsc1 := Asc( cShift[1] )       // result:  4
      ? nAsc2 := Asc( cShift[2] )       // result:  8
      ? NtoC( nAsc1, 2, 8, "0" )        // result: 00000100
      ? NtoC( nAsc2, 2, 8, "0" )        // result: 00001000
   RETURN
```

CharSHR()

Shifts bits in a character string to the right.

Syntax

```
CharSHR( <cString>, <nShift> ) --> cResult
```

Arguments

<cString>

This is the character string to process.

<nShift>

This is a numeric integer value indicating how many places the bits of each character of *<cString>* are shifted to the right.

Return

The function returns a string by shifting the bits of the individual characters of *<cString>* for *<nShift>* places to the right. The lowest bit is discarded and the highest bit is set to zero.

Info

See also: >>, CharAND(), CharNOT(), CharOR(), CharRLL(), CharRLR(), CharSHL(), CharXOR(), HB_BitShift()
Category: CT:String manipulation, Bitwise functions, Character functions, xHarbour extensions
Source: ct\charop.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates the effect of shifting bits in
// a two-character string, by displaying the ASCII codes and
// their binary representation before and after the right shift
// operation. Note that the least significant bit is lost.

   PROCEDURE Main
      LOCAL cString := "AB"
      LOCAL cShift
      LOCAL nAsc1, nAsc2

      ? nAsc1 := Asc( cString[1] )       // result: 65
      ? nAsc2 := Asc( cString[2] )       // result: 66

      ? NtoC( nAsc1, 2, 8, "0" )         // result: 01000001
      ? NtoC( nAsc2, 2, 8, "0" )         // result: 01000010

      cShift := CharSHR( cString, 2 )

      ? nAsc1 := Asc( cShift[1] )        // result: 16
      ? nAsc2 := Asc( cShift[2] )        // result: 16
      ? NtoC( nAsc1, 2, 8, "0" )         // result: 00010000
      ? NtoC( nAsc2, 2, 8, "0" )         // result: 00010000
   RETURN
```

CharSList()

Removes duplicate characters from a string and sorts the result.

Syntax

```
CharSList( <cString> ) --> cSortedList
```

Arguments

<cString>

This is the character string to process.

Return

The function is a combination of CharList() and CharSort(), i.e. it removes all duplicate characters from the input string and returns the sorted result.

Info

See also: CharHist(), CharList(), CharNoList(), CharSort()
Category: CT:String manipulation, Character functions, xHarbour extensions
Source: ct\misc1.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example comperes results of CharSList() and CharList()

   PROCEDURE Main
      LOCAL cString1 := "xHarbour compiler"
      LOCAL cString2 := "bbbccccaa"

      ? CharList( cString1 )    // result: xHarbou cmpile

      ? CharSList( cString1 )   // result: Habceilmoprux

      ? CharList( cString2 )    // result: bca

      ? CharSList( cString2 )   // result: abc

   RETURN
```

CharSort()

Sorts character (sequences) within a string.

Syntax

```
CharSort( <cString>     , ;
         [<nSeqLen>]    , ;
         [<nCompareLen>], ;
         [<nSkipChars>] , ;
         [<nSkipSeq>]   , ;
         [<nSortLen>]   , ;
         [<lDescending>]  ) --> cResult
```

Arguments

<cString>

This is the character string to process.

<nSeqLen>

This is a numeric value defining the number of characters per character sequence to be sorted. The default value is 1 (single character).

<nCompare>

This is a numeric value defining the number of characters per character sequence that are recognized in a sort. The default is *<nSeqLen>*, so that all characters of a sequence are recognized.

<nSkipChars>

This numeric parameter defaults to 0 so that the function begins sorting with the first character of *<cString>*. Passing a value > 0 instructs the function to ignore the first *<nSkipChars>* characters in the operation.

<nSkipSeq>

This numeric parameter defaults to 0 so that the functions sorts character sqeuences with the first character of a sequence. Passing a value > 0 instructs the function to ignore the first *<nSkipSeq>* characters in a character sequence.

<nSortLen>

This numeric parameter defaults to Len(*<STring>* so that the functions sorts all character sqeuences contained in *<cString>*. Passing a value > 0 instructs the function to stop sorting at position *<nSkipChars>* + *<nSortLen>*.

<lDescending>

By default, the function sorts in asscending order. When .T. (true) is passed for *<lDescending>*, the result string is sorted in descending order.

Return

The function returns the sorted string or a null string ("") when an error occurs.

Info

See also: CharRela()
Category: CT:String manipulation, Character functions
Source: ct\charsort.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates various sort results of CharSort()

   PROCEDURE Main
      LOCAL cStr := "4ef2ab1cd3gh"

      // sort single characters
      ? CharSort( cStr )             // result: 1234abcdefgh

      // sort character pairs
      ? CharSort( cStr, 2 )          // result: 1c4eabd3f2gh

      // sort character triplets (by digit)
      ? CharSort( cStr, 3 )          // result: 1cd2ab3gh4ef

      // sort character triplets (by alphabet)
      ? CharSort( cStr, 3, 2, 0, 1 ) // result: 2ab1cd4ef3gh
   RETURN
```

CharSpread()

Formats a character string for block paragraphs.

Syntax

```
CharSpread( <cString>  , ;
            <nLineLen> , ;
           [<xInsChar>]  ) --> cResult
```

Arguments

<cString>

This is a character string to format for a block paragraph.

<nLineLen>

A numeric value specifying the length of a line in the block paragraph (fixed font).

<xInsChar>

This parameter defaults to 32, i.e. Chr(32) is inserted into *<cString>* for formatting. *<xInsChar>* is either a numeric ASCII code or a single character of Valtype()="C".

Return

The function returns a string formatted to the length of *<nLineLen>*. The string is expanded by inserting *<xInsChar>* between words. If *<xInsChar>* is not contained in *<cString>*, the function returns a unchanged string.

Info

See also:	Center(), CharOne(), Expand()
Category:	CT:String manipulation, Character functions
Source:	ct\spread.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
PROCEDURE Main
   LOCAL cString := "The xHarbour compiler"

   ? CharSpread( cString, 35 )

   // result: "The        xHarbour        compiler"
RETURN
```

CharSub()

Creates a string by subtracting ASCII codes of two strings.

Syntax

```
CharSub( <cString1>, <cString2> ) --> cResult
```

Arguments

`<cString1> and <cString2>`

These are two-character strings whose ASCII codes are subtracted in the result string.

Return

The function returns a string by subtracting the ASCII codes of the individual characters of *<cStringg2>* from those of *<cString1>*.

Description

The function operates until all characters of *<cString1>* are processed. It distinguishes the following situations:

1) **Len(*<cString1>*) == Len(*<cString2>*)**

   When both input strings are of the same length, the ASCII code of each character in *<cString2>* is subtracted from the ASCII code of the corresponding character in *<cString1>*.

2) **Len(*<cString1>*) > Len(*<cString2>*)**

   When the last character of *<cString2>* is reached, the function starts over with the first character of *<cString2>*, until the last character of *<cString1>* is processed.

3) **Len(*<cString1>*) < *Len(* *<cString2>*)**

   The function returns when the last character of *<cString1>* is processed.

Info

See also: AddASCII(), CharAdd(), CharAND(), CharOR(), CharXOR()
Category: CT:String manipulation, Character functions, xHarbour extensions
Source: ct\charop.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The function displays results of CharSub()

   PROCEDURE Main
      LOCAL cString := "abc"

      ? CharSub( cString, Chr(9)  )    // result: XYZ

      ? CharSub( cString, Space(3) )   // result: ABC
   RETURN
```

CharSwap()

Exchanges adjacent characters in a string.

Syntax

```
CharSwap( <cString> ) --> cResult
```

Arguments

<cString>

This is the character string to process.

Return

The function exchanges characters at odd positions with characters at even positions and returns the resulting string.

Info

See also: CharSort(), WordSwap()
Category: CT:String manipulation, Character functions
Source: ct\charswap.c
LIB: xhb.lib
DLL: xhbdll.dll

CharUnpack()

Uncompresses a CharPack() compressed string.

Syntax

```
CharUnpack( <cCompressed> ) --> cUncompressed
```

Arguments

<cCompressed>

This is a character string as returned from function CharPack().

Return

The function returns the original, uncompressed string previously passed to Charpack().

Info

See also:	CharPack()
Category:	CT:String manipulation, Character functions
Source:	ct\pack.c
LIB:	xhb.lib
DLL:	xhbdll.dll

CharWin()

Replaces characters in a specified screen area.

Syntax

```
CharWin( [<nTop>]    , ;
         [<nLeft>]   , ;
         [<nBottom>] , ;
         [<nRight>]  , ;
         [<xNewChar>], ;
         [<xOldChar>]  ) --> cNull
```

Arguments

`<nTop> and <nLeft>`

Numeric values indicating the screen coordinates for the upper left corner of the screen area. The default value for both parameters is zero.

`<nBottom>`

Numeric value indicating the bottom row screen coordinate. It defaults to MaxRow().

`<nRight>`

Numeric value indicating the right column screen coordinate. It defaults to MaxCol().

`<xNewChar>`

This is a single character or its numeric ASCII code. It replaces *<xOldChar>* and defaults to the return value of GetClearB().

`<xOldChar>`

This is a single character or its numeric ASCII code to be replaced on the screen. If omitted, all characters displayed on the screen are replaced.

Return

The return value is always a null string ("").

Info

See also: ColorWin(), SetClearB()
Category: CT:Video, Screen functions
Source: ct\screen3.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays a box of "A" and replaces it with "B"

   PROCEDURE Main
      CLS
      DispBox( 10, 10, 20, 20, "A" )

      WAIT
      CharWin( 10, 10, 20, 20, "B", "A" )

   RETURN
```

CharXOR()

Binary XORs the ASCII codes of characters in two strings.

Syntax

```
CharXOR( <cString1>, <cString2> ) --> cResult
```

Arguments

`<cString1> and <cString2>`

These are two character strings whose ASCII codes are XORed in the result string.

Return

The function returns a string by XORing the ASCII codes of the individual characters of both input strings.

Description

The function operates until all characters of *<cString1>* are processed. It distinguishes the following situations:

1) **Len(*<cString1>*) == Len(*<cString2>*)!EF**

   When both input strings are of the same length, the ASCII code of each character in *<cString2>* is XORed with the ASCII code of the corresponding character in *<cString1>*.

2) **Len(*<cString1>*) > Len(*<cString2>*)!EF**

   When the last character of *<cString2>* is reached, the function starts over with the first character of *<cString2>*, until the last character of *<cString1>* is processed.

3) **Len(*<cString1>*) < *Len(* *<cString2>*)!EF**

   The function returns when the last character of *<cString1>* is processed.

Note: CharXOR() exists for compatibility reasons. It is superseeded by xHarbour's binary XOR operator.

Info

See also:	^^, CharAND(), CharNOT(), CharOR(), Crypt()
Category:	CT:String manipulation, Bitwise functions, Character functions
Source:	ct\charop.c
LIB:	xhb.lib
DLL:	xhbdll.dll

CheckSum()

Calculates the checksum for a character string.

Syntax

```
CheckSum( <cString> ) --> nCheckSum
```

Arguments

<cString>

This is the character string to calculate a chacksum for.

Return

The function returns the checksum for the input string as a numeric value.

Info

See also:	ASCIISum()
Category:	CT:String manipulation, Character functions, Checksum functions
Source:	ct\ctchksum.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Chr()

Converts a numeric ASCII code to a character.

Syntax

```
Chr( <nAsciiCode> )  --> cCharacter
```

Arguments

```
<nAsciiCode>
```

This is an expression yielding a numeric value in the range of 0 to 255.

Return

The function returns a single character that has the numeric ASCII code <*nAsciiCode*>.

Description

The function converts a numeric value to a character of the corresponding ASCII code. ASCII codes are integer values in the range of 0 to 255. When a value outside this range is passed, it is corrected according to the following rules:

1) If the value <*nAsciiCode*> is larger than 255, the ASCII code is calculated as <*nAsciiCode*> modulus 256.

2) If <*nAsciiCode*> is negative, the ASCII code is calculated as 256 + <*nAsciiCode*> modulus 256.

The Chr() function is used to compose character strings consisting of non-printable characters. Such strings can contain control characters for a device, like printer or keyboard, for example.

Info

See also:	Asc(), Inkey(), KEYBOARD
Category:	Numeric functions, Conversion functions
Source:	rtl\chrasc.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example converts various numeric values to characters
// and creates a text string including non-printable characters

   PROCEDURE Main
      LOCAL cText

      ? Chr(65)                        // result: A
      ? Chr(322)                       // result: B

      ? Asc( Chr(322) )                // result: 66
      ? 322 % 256                      // result: 66

      n := -142
      ? Chr( n )                       // result: r

      ? Asc( Chr(n) )                  // result: 114
      ? 256+n % 256                    // result: 114

      cText := "This is a"
      cText += Chr(13) + Chr(10)
      cText += "two line text"
```

```
   ? cText
RETURN
```

ClearBit()

Sets one or more bits of a numeric integer value to 0.

Syntax

```
ClearBit( <nInteger>|<cHex>,[<nBitPos,...>]) --> nNewValue
```

Arguments

`<nInteger>`

A numeric 32-bit integer value can be specified as first parameter.

`<cHex>`

Alternatively, the integer can be passed as a hex-encoded character string (see NumToHex()).

`<nBitPos>`

The positions of the bits to clear are defined as a comma separated list of numeric integer values. The range for *<nBitPos>* is 1 to 32.

Return

The function returns a numeric integer value with the bits at the specified positions set to 0.

Info

See also:	IsBit(), NumAND(), NumNOT(), NumOR(), NumXOR(), SetBit()
Category:	CT:NumBits, Numbers and Bits
Source:	ct\bit2.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays results of ClearBit() along with the
// binary representation of numbers.

   PROCEDURE Main

      ? NtoC( 255, 2, 8, "0" )          // result: 11111111

      ? n := ClearBit( 255, 1, 7 )      // result:   190.00
      ? NtoC( n, 2, 8, "0" )            // result: 10111110

      ? n := ClearBit( 255, 2, 4, 6 )   // result:   213.00
      ? NtoC( n, 2, 8, "0" )            // result: 11010101

   RETURN
```

ClearEol()

Clears a row on the screen beginning at a specified position.

Syntax

```
ClearEol( [<nRow>]  , ;
          [<nCol>]  , ;
          [<xColor>], ;
          [<xChar>]   ) --> cNull
```

Arguments

<nRow>

A numeric value indicating the screen row to clear. It defaults to Row().

<nCol>

A numeric value indicating the screen column to begin to clear. It defaults to Col().

<xColor>

This is either a single color value (see SetColor()), or its numeric color attribute (see ColorToN()). It defines the color for clearing the row and defaults to the return value of GetClearA().

<xChar>

This is a single character or its numeric ASCII code. It is displayed in the screen row beginning at the specified column to the rightmost column. The default value is determined by GetClearB().

Return

The return value is always a null string ("").

Note: the function does not change the current screen cursor position.

Info

See also: ClEol(), SetClearA(), SetClearB()
Category: CT:Video, Screen functions
Source: ct\screen3.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays a dashed yellow line across the screen in
// every fifth row.

   PROCEDURE Main
      LOCAL i
      CLS

      FOR i:=0 TO MaxRow()
         ? Str(i,2),": "

         IF i % 5 == 0
            ?? ClearEol( , , "GR+/N", "-" )
         ENDIF
      NEXT
   RETURN
```

ClearSlow()

Clears a screen area incrementally with a delayed imploding effect.

Syntax

```
ClearSlow( <nDelay>  , ;
          [<nTop>]   , ;
          [<nLeft>]  , ;
          [<nBottom>], ;
          [<nRight>] , ;
          [<xChar>]    ) --> cNull
```

Arguments

<nDelay>

This is a numeric value specifying the number of milliseconds the function waits for the next incremental clearing step.

<nTop> and <nLeft>

Numeric values indicating the screen coordinates for the upper left corner of the screen area to clear. The default value for both parameters is zero.

<nBottom>

Numeric value indicating the bottom row screen coordinate. It defaults to MaxRow().

<nRight>

Numeric value indicating the right column screen coordinate. It defaults to MaxCol().

<xChar>

This is a single character or its numeric ASCII code. It is used for clearing the screen and defaults to the return value of GetClearB().

Return

The return value is always a null string ("").

Info

See also:	@...CLEAR, SetClearB()
Category:	CT:Video, Screen functions
Source:	ct\screen3.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays a box and shows the imploding effect
// of ClearSlow().

   #include "Box.ch"

   PROCEDURE Main
      CLS

      DispBox( 5, 5, 20, 75, B_SINGLE+ " ", "W+/B" )
      WAIT

      ClearSlow( 20, 5, 5, 20, 75, " " )
   RETURN
```

ClearWin()

Clears all or parts of the screen.

Syntax

```
ClearWin( [<nTop>]    , ;
          [<nLeft>]   , ;
          [<nBottom>], ;
          [<nRight>] , ;
          [<xColor>] , ;
          [<xChar>]    ) --> cNull
```

Arguments

<nTop> and <nLeft>

Numeric values indicating the screen coordinates for the upper left corner of the screen area to clear. The default value for both parameters is determined by Row() and Col().

<nBottom>

Numeric value indicating the bottom row screen coordinate. It defaults to MaxRow().

<nRight>

Numeric value indicating the right column screen coordinate. It defaults to MaxCol().

<xColor>

This is either a single color value (see SetColor()), or its numeric color attribute (see ColorToN()). It defines the color for clearing the screen and defaults to the return value of GetClearA().

<xChar>

This is a single character or its numeric ASCII code. It is used for clearing the screen and defaults to the return value of GetClearB().

Return

The return value is always a null string ("").

Note: ClearWin() leaves the cursor position unchanged.

Info

See also: @...CLEAR, ClearSlow(), ClWin(), SetClearA(), SetClearB()
Category: CT:Video, Screen functions
Source: ct\setclear.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
PROCEDURE Main
   SET COLOR TO N/B
   CLS

   SetPos( 2, 20 )
   ClWin()
RETURN
```

ClEol()

Clears characters and colors in a row on the screen.

Syntax

```
ClEol( [<nRow>], [<nCol>] ) --> cNull
```

Arguments

<nRow>

This is a numeric value specifying the row on the screen to clear. It defaults to Row().

<nCol>

This is a numeric value specifying the column on the screen to begin clearing up to the rightmost column. It defaults to Col().

Return

The return value is always a null string ("").

Note: ClEol() leaves the cursor position unchanged.

Info

See also:	ClWin(), ClearEol(), ClearWin()
Category:	CT:Video, Screen functions
Source:	ct\screen3.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

ClWin()

Clears characters and colors on the screen.

Syntax

```
ClWin( [<nTop>]    , ;
       [<nLeft>]   , ;
       [<nBottom>], ;
       [<nRight>]    ) --> cNull
```

Arguments

`<nTop> and <nLeft>`

Numeric values indicating the screen coordinates for the upper left corner of the screen area to clear. The default value for both parameters is determined by Row() and Col().

`<nBottom> and <nRight>`

Numeric values indicating the screen coordinates for the lower right corner of the screen area to clear. The default value for both parameters is determined by MaxRow() and MaxCol().

Return

The return value is always a null string ("").

Note: ClWin() leaves the cursor position unchanged.

Info

See also:	ClEol(), ClearEol(), ClearWin()
Category:	CT:Video, Screen functions
Source:	ct\screen3.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

CMonth()

Returns the name of a month from a date.

Syntax

```
CMonth( <dDate> )  --> cMonthName
```

Arguments

<dDate>

This is an expression returning a value of data type Date.

Return

The function returns a character string holding the month name of *<dDate>*. When *<dDate>* is an empty date, the function returns a null string ("").

Description

The CMonth() function converts a date value into the name of the corresponding month. Use CDoW() to obtain the weekday name as character value. Both functions are used to format date values in a textual way.

Note: xHarbour's national language support allows for returning the name of a month in various languages. Refer to HB_LangSelect() for selecting a language module.

Info

See also: CDoW(), Date(), Day(), DoW(), DtoC(), HB_LangSelect(), Month(), StoD(), Year()
Category: Conversion functions, Date and time
Source: rtl\datec.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example uses a user defined function to fill an array
// with all month names abbreviated to the first three letters.

   PROCEDURE Main
      LOCAL aMonths := MonthNames(3)

      // display all month names
      AEval( aMonths, {|c| QOut(c) } )
   RETURN

   FUNCTION MonthNames( nLen )
      LOCAL aArray[12]
      LOCAL i, cDate, cMonth

      FOR i:=1 TO 12
         cDate := "2000" + PadL( i, 2, "0" ) + "01"
         cMonth:= CMonth( StoD( cDate ) )

         IF Valtype( nLen ) == "N"
            aArray[i] := Left( cMonth, nLen )
         ELSE
            aArray[i] := cMonth
         ENDIF
      NEXT
```

```
RETURN aArray
```

Col()

Returns the current column position of the screen cursor

Syntax

```
Col()  --> nColPos
```

Return

The function returns a numeric value indicating the current column position of the screen cursor. The leftmost column has position 0 and the rightmost position is identified by MaxCol().

Description

The function returns the current cursor column position within a console window (text-mode). The cursor position changes with screen output using console output commands and functions.

Use function SetPos() to position the screen cursor at a defined row and column coordinate.

Info

See also:	@...CLEAR, @...GET, @...SAY, CLEAR SCREEN, MaxCol(), MaxRow(), PCol(), PRow(), Row(), SetMode(), SetPos()
Category:	Screen functions
Source:	rtl\setpos.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The examples displays and changes the screen cursor position.

   PROCEDURE Main
      ? Col()                          // result: 0

      ? "xHarbour"

      ? Col()                          // result: 8
   RETURN
```

ColorRepl()

Replaces color attributes on the screen.

Syntax

```
ColorRepl( [<xNewColor>], [<xOldColor>] ) --> cNull
```

Arguments

<xNewColor>

This is either a single color value (see SetColor()), or its numeric color attribute (see ColorToN()). It defines the new color and defaults to the return value of GetClearA().

<xOldColor>

Either a single color value or its numeric color attribute to be replaced can be specified. If omitted, all colors are replaced with *<xNewColor>*.

Return

The return value is always a null string ("").

Info

See also:	ColorToN(), ColorWin()
Category:	CT:Video, Screen functions
Source:	ct\screen3.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

ColorSelect()

Selects a color from the current SetColor() string.

Syntax

```
ColorSelect( <nColorIndex> ) --> NIL
```

Arguments

<nColorIndex>

A numeric value identifying the ordinal position of a color value in the SetColor() string. *<nColorIndex>*is zero based, i.e. the first color value has the ordinal position 0.

Return

The return value is always NIL.

Description

The function ColorSelect() selects a color value from the current SetColor() string for standard console output. ColorSelect() does not change the SetColor() string. Constants from the COLOR.CH file can be used to address a specific color value:

#defined constants for ColorSelect()

Constant	Value	Description
CLR_STANDARD	0	All screen output commands and functions
CLR_ENHANCED	1	GETs and selection highlights
CLR_BORDER	2	Screen border (not supported on EGA and VGA monitors)
CLR_BACKGROUND	3	Not supported
CLR_UNSELECTED	4	Unselected GETs

If the SetColor() string contains more than five color values, other colors can be selected by passing a value > 4 to the function.

Info

See also: SET COLOR, SetBlink(), SetColor()
Category: Screen functions
Header: color.ch
Source: rtl\setcolor.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates changing of colors for the ? command.

   #include "Color.ch"

   PROCEDURE Main
      LOCAL cColor := SetColor( "N/W,W+/N,W+/W,W+/B,GR+/B" )

      ? "SetColor():", SetColor()
      ?
      ? "ColorSelect( CLR_STANDARD )"

         ColorSelect( CLR_ENHANCED )
      ? "ColorSelect( CLR_ENHANCED )"
```

```
      ColorSelect( CLR_BORDER )
   ? "ColorSelect( CLR_BORDER )"

      ColorSelect( 3 )
   ? "ColorSelect( CLR_BACKGROUND )"

      ColorSelect( 4 )
   ? "ColorSelect( CLR_UNSELECTED )"

   ColorSelect( 0 )

   ? "Back to standard color"
RETURN
```

ColorToN()

Converts a color value to a numeric color attribute.

Syntax

```
ColorToN( <cColor> ) --> nColorAttribute
```

Arguments

<cColor>

This is a SetColor() compliant color string.

Return

The function converts a color value to the corresponding color attribute and returns the attribute as a numeric value.

Info

See also:	NtoColor()
Category:	CT:Video, Screen functions
Source:	ct\color.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

ColorWin()

Replaces a color attribute in a screen region.

Syntax

```
ColorWin( [<nTop>]     , ;
          [<nLeft>]    , ;
          [<nBottom>]  , ;
          [<nRight>]   , ;
          [<xNewColor>], ;
          [<xOldColor>]  ) --> cNull
```

Arguments

`<nTop> and <nLeft>`

Numeric values indicating the screen coordinates for the upper left corner of the screen area. The default value for both parameters is zero.

`<nBottom> and <nRight>`

Numeric values indicating the screen coordinates for the lower right corner of the screen area. The default value for both parameters is determined by MaxRow() and MaxCol().

`<xNewColor>`

This is either a single color value (see SetColor()), or its numeric color attribute (see ColorToN()). It defines the new color and defaults to the return value of GetClearA().

`<xOldColor>`

Either a single color value or its numeric color attribute to be replaced can be specified. If omitted, all colors are replaced with *<xNewColor>*.

Return

The return value is always a null string ("").

Info

See also: ColorRepl(), ColorToN()
Category: CT:Video, Screen functions
Source: ct\screen3.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays a box and changes its color.

   #include "Box.ch"

   PROCEDURE Main
      SET COLOR TO N/B
      CLS

      DispBox( 5, 5, 20, 75, B_SINGLE+ " ", "W+/R" )
      WAIT

      ColorWin( 20, 5, 5, 75, "N/BG" )
   RETURN
```

Complement()

Creates the complement for values of data type C, D, L, M, N

Syntax

```
Complement( <xValue> ) --> xComplement
```

Arguments

<xValue>

This is a value of data type C, D, L, M or N.

Return

The function returns the complement of the passed value, or NIL if an illegal data type is passed.

Info

Category: CT:Miscellaneous, Miscellaneous functions
Source: ct\misc2.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays results of Complement()

   PROCEDURE Main
      LOCAL cString := "xHarbour"
      LOCAL dDate   := StoD( "20070131" )
      LOCAL lLogic  := .T.
      LOCAL nNumber := 3.15

      ? Complement( cString )  // result: (not printable)
      ? Complement( dDate   )  // result: 11/30/92
      ? Complement( lLogic  )  // result: .F.
      ? Complement( nNumber )  // result: -3.15
   RETURN
```

ConvToAnsiCP()

Converts an OEM string to the ANSI character set.

Syntax

```
ConvToAnsiCP( <cOEM_String> ) --> cANSI_String
```

Arguments

<cOEM_String>

This is a character string encoded with the OEM character set.

Return

The function returns *<cOEM_String>* converted to the ANSI character set.

Description

The function ConvToAnsiCP() is a synonym for HB_OemToAnsi(). Refer to function HB_OemToAnsi() .

Info

See also:	HB_OemToAnsi()
Category:	Conversion functions
Source:	rtl\oemansi.c
LIB:	xhb.lib
DLL:	xhbdll.dll

ConvToOemCP()

Converts an ANSI string to the OEM character set.

Syntax

```
ConvToOemCP( <cANSI_String> ) --> cOEM_String
```

Arguments

<cANSI_String>

This is a character string encoded with the ANSI character set.

Return

The function returns *<cANSI_String>* converted to the OEM character set.

Description

The function ConvToOemCP() is a synonym for HB_AnsiToOem(). Refer to function HB_AnsiToOem().

Info

See also:	HB_AnsiToOem()
Category:	Conversion functions
Source:	rtl\oemansi.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Cos()

Calculates the cosine for an angle.

Syntax

```
Cos( <nAngle> ) --> nCosine
```

Arguments

<nAngle>

A numeric value specifies the angle as a fraction (or multiple) of Pi().

Return

The function returns the cosine of an angle in the range between -1 and +1.

Info

See also: CosH(), Cot(), DtoR(), RtoD(), SetPrec(), Sin(), Tan()
Category: CT:Math, Mathematical functions, Trigonometric functions
Source: ct\trig.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays results of function Cos().

   PROCEDURE Main
      ? Str( Cos( -Pi()*1.0 ), 18, 15)  // result: -1.000000000000000
      ? Str( Cos( -Pi()*0.5 ), 18, 15)  // result:  0.000000000000000
      ? Str( Cos( -Pi()*0.3 ), 18, 15)  // result:  0.587785252292473
      ? Str( Cos(  Pi()*0.0 ), 18, 15)  // result:  1.000000000000000
      ? Str( Cos(  Pi()*0.3 ), 18, 15)  // result:  0.587785252292473
      ? Str( Cos(  Pi()*0.5 ), 18, 15)  // result:  0.000000000000000
      ? Str( Cos(  Pi()*1.0 ), 18, 15)  // result: -1.000000000000000
   RETURN
```

CosH()

Calculates the hyperbolic cosine for an angle.

Syntax

```
CosH( <nRadians> ) --> nHyperbolicCosine
```

Arguments

<nRadians>

A numeric value specifying the angle value in radians.

Return

The function returns the hyperbolic cosine in the range of plus and minus Infinity().

Info

See also: Cos(), Cot(), DtoR(), RtoD(), SetPrec(), Sin(), Tan()
Category: CT:Math, Mathematical functions, Trigonometric functions, xHarbour extensions
Source: ct\trig.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example display results of CosH(). Note that there is a numeric overflow.

   PROCEDURE Main
      ? Str( CosH( 0.0 ), 18, 15)  // result:  1.000000000000000
      ? Str( CosH( 1.0 ), 18, 15)  // result:  1.543080634815244
      ? Str( CosH( 2.0 ), 18, 15)  // result:  3.762195691083631
      ? Str( CosH( 4.0 ), 18, 15)  // result: 27.308232836016490
      ? Str( CosH( 8.0 ), 18, 15)  // result: ******************
   RETURN
```

Cot()

Calculates the cotangent.

Syntax

```
Cot( <nAngle> ) --> nCotangent
```

Arguments

`<nAngle>`

A numeric value specifies the angle as a fraction (or multiple) of Pi().

Return

The function returns the cotangent of an angle in the range between minus and plus Infinity().

Info

See also:	Cos(), DtoR(), RtoD(), Sin(), Tan()
Category:	CT:Math, Mathematical functions, Trigonometric functions
Source:	ct\trig.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays results of function Cot().

   PROCEDURE Main
      ? Str( Cot( -Pi()*1.0 ), 18, 15)  // result: ******************
      ? Str( Cot( -Pi()*0.5 ), 18, 15)  // result:  0.000000000000000
      ? Str( Cot( -Pi()*0.3 ), 18, 15)  // result: -0.726542528005361
      ? Str( Cot(  Pi()*0.0 ), 18, 15)  // result: ******************
      ? Str( Cot(  Pi()*0.3 ), 18, 15)  // result:  0.726542528005361
      ? Str( Cot(  Pi()*0.5 ), 18, 15)  // result:  0.000000000000000
      ? Str( Cot(  Pi()*1.0 ), 18, 15)  // result: ******************
   RETURN
```

CountGets()

Returns the number of Get fields in the current Getlist array.

Syntax

```
CountGets() --> nGetCount
```

Return

The function returns the number of Get objects stored in the PUBLIC *Getlist* array.

Info

See also: @...GET, CurrentGet()
Category: CT:GetSys, Get system
Source: ct\getinfo.prg
LIB: xhb.lib
DLL: xhbdll.dll

CountLeft()

Counts a specified character from the left side of a string.

Syntax

```
CountLeft( <cString>, [<xChar>] ) --> nCount
```

Arguments

<cString>

This is the character string to process.

<xChar>

A single character or its numeric ASCII code can be passed. The default value is a space character (Chr(32)).

Return

The function counts *<xChar>* at the beginning of *<cString>* and returns as soon as a different character is detected. The number of times the character *<xChar>* is found consecutively at the beginning of *<cString>* is returned as a numeric value.

Info

See also: CountRight(), LTrim()
Category: CT:String manipulation, Character functions
Source: ct\count.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays results of CountLeft()

   PROCEDURE Main
      LOCAL cString1 := "   Hello World "
      LOCAL cString2 := "aaaaabbbbbcc"

      ? CountLeft( cString1 )        // result: 3
      ? CountLeft( cString2, "a" )   // result: 5

   RETURN
```

CountRight()

Counts a specified character from the right side of a string.

Syntax

```
CountRight( <cString>, [<xChar>] ) --> nCount
```

Arguments

<cString>

This is the character string to process.

<xChar>

A single character or its numeric ASCII code can be passed. The default value is a space character (Chr(32)).

Return

The function counts *<xChar>* at the end of *<cString>* and returns as soon as a different character is detected. The number of times the character *<xChar>* is found consecutively at the end of *<cString>* is returned as a numeric value.

Info

See also: CountLeft(), RTrim()
Category: CT:String manipulation, Character functions
Source: ct\count.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays results of CountRight()

   PROCEDURE Main
       LOCAL cString1 := "   Hello World "
       LOCAL cString2 := "aaaaabbbbbcc"

       ? CountRight( cString1 )        // result: 1
       ? CountRight( cString2, "c" )   // result: 2

   RETURN
```

CreateObject()

Instantiates a new OLE Automation object.

Syntax

```
CreateObject( <cProgID> ) --> oOleAuto
```

Arguments

<cProgID>

This is a character string holding the OLE ProgID of the application to use for automation. ProgIDs consist of a program name, usually followed by ".Application". For example:

```
cProgID := "Word.Application"              // MS Office Word
cProgID := "Excel.Application"             // MS Office Excel
cProgID := "InternetExplorer.Application"  // Internet Explorer
```

Return

The function returns a TOleAuto object which handles OLE automation on behalf of the xHarbour application. When the requested OLE object cannot be instantiated, a runtime error is raised.

Description

OLE automation is a fundamental technology on Windows platforms. It allows for reusing functionality built into existing applications not developed with xHarbour. CreateObject() is the only function required to instantiate an OLE Automation object. For example:

```
oOLE := CreateObject( "Word.Application" )
```

This function call loads an instance of MS Office Word into memory and makes the functionality of Word available to the xHarbour application. The returned xHarbour OLE object communicates with the Word application, sends requests to it and returns responses of Word to the xHarbour application.

An OLE Automation object exposes its interface(s), methods and properties to the calling xHarbour application. They are available via methods and instance variables of the xHarbour object returned from CreateObject(). When a method name is sent to this object, the method of the same name of the OLE Automation object is invoked. Likewise, when a value is assigned to an instance variable of the xHarbour object, this value is passed on to a property of the OLE Automation object.

A detailed explanation of the communication between xHarbour and an OLE Automation object goes far beyond this documentation. However, the problem of the developer is to know the names of properties, methods and their arguments available in the OLE Automation object. This information is stored in what is called a *Type Library* which is part of an OLE application.

Type libraries can be contained in separate files (.tlb, .olb) or can be linked to an executable file (.dll, .exe). The type Library information is extracted from the calling process and is interpreted for data exchange. Refer to the *ListOle.prg* file in the samples folder of your xHarbour installation for listing information avialable in type libraries.

Note: a commonly used TypeLib viewer is called *OleView.exe* and is included in the Microsoft Windows SDK. Two example programs below are helpful for an xHarbour user for extracting #define constants and finding help information on an OLE application. Default locations for type libraries of common Microsoft products are listed in the tables below:

Default MS Office directories

Version	Installation directory
Office 97	C:\Program Files\Microsoft Office\Office
Office 2000	C:\Program Files\Microsoft Office\Office
Office XP	C:\Program Files\Microsoft Office\Office10

Office 2003	C:\Program Files\Microsoft Office\Office11
Office 2007	C:\Program Files\Microsoft Office\Office12

Type library names

Office product	Ver 97	Ver 2000	Ver XP, 2003 and later
Access	Msacc8.olb	Msacc9.olb	Msacc.olb
Excel	Excel8.olb	Excel9.olb	Excel.exe
Graph	Graph8.olb	Graph9.olb	Graph.exe
Outlook	Msoutl97.olb	Msoutl9.olb	MSOutl.olb
PowerPoint	Msppt8.olb	Msppt9.olb	MSPpt.olb
Word	Msword8.olb	Msword9.olb	MSWord.olb

The type library of the Internet Explorer is stored in *C:\windows\system32\shdocvw.dll*.

Info

See also: GetActiveObject(), Ole2TxtError(), OleError()
Category: OLE Automation, xHarbour extensions
Source: rtl\win32ole.prg
LIB: xhb.lib, ole.lib
DLL: xhbdll.dll

Examples

Simple Office Automation

```
// The example outlines Office Automation using MS Office Word as
// automation object. The example creates a new Word document, and
// saves it as DOC, RTF and PDF file.

   #define CRLF   Chr(13)+Chr(10)

   #define wdFormatDocument   0  // .DOC
   #define wdFormatRtf        6  // .RTF

   PROCEDURE Main
      LOCAL cFileName := CurDrive()+":\"+CurDir()+"\test"
      LOCAL cDocFile  := cFileName+".doc"
      LOCAL cRtfFile  := cFileName+".rtf"
      LOCAL oWord, oDocument, oText

      TRY
         oWord := GetActiveObject( "Word.Application" )
      CATCH
         TRY
            oWord := CreateObject( "Word.Application" )
         CATCH
            Alert( "ERROR! Word not avialable. [" + Ole2TxtError()+ "]" )
            RETURN
         END
      END

      // create a new, empty Word document
      oDocument := oWord:documents:add()

      // get the text selection object
      oText := oWord:selection()

      // assign some text
```

```
      oText:Text := "OLE from xHarbour" + CRLF

      // select font name, size and attribute
      oText:font:name := "Arial"
      oText:font:size := 48
      oText:font:bold := .T.

      // display the Word window
      oWord:visible := .T.

      // maximize the word window
      ? oWord:windowState := 1

      // save file as regular DOC file
      oDocument:saveAs( cDocFile )

      // save file as rich text
      oDocument:saveAs( cRtfFile, wdFormatRtf )

      // hide Word window
      oWord:Visible := .F.
      oDocument:close()
      /****************** C A U T I O N **********************
       *
       * oDocument is automatically released from memory and
       * cannot be used anymore after being :close()ed
       *
       ****************** C A U T I O N **********************/

      // create new Document object
      oDocument := oWord:documents:open( cRtfFile )
      oWord:visible := .T.

      // select PDF printer driver
      oWord:activePrinter := "FreePDF XP"

      // print PDF file via PDF printer driver
      oWord:printOut()

      // terminate Word application
      oWord:quit()

   RETURN
```

Finding OLE Help files

```
// A good OLE Automation product comes with a help file explaining
// the interfaces, properties and methods. The following example
// implements a utility program that opens the online help file of
// an OLE application, if it is available.
//
// The complete file name of the file including the type library
// must be passed as command line parameter.
//
// e.g: C:\Program Files\Microsoft Office\Office12\MsWord.olb

   #define NO_HELP_FILE     "No help file available"
   #define SW_SHOW          5

   PROCEDURE Main( cFileName )
      LOCAL oTypeLib, cHelpFile, cSearch
      CLS
```

```
   IF Empty( cFileName )
      ? "Usage: FindHelp.exe <typeLibraryFileName>"
      QUIT
   ENDIF

   TRY
      ? "Loading Type Library, please wait..."
      oTypeLib := LoadTypeLib( cFileName, .T. )
   CATCH
      ? "Unable to load Type Library for:", cFileName
      QUIT
   END

   cHelpFile := OLEHelpFile( oTypeLib )

   IF cHelpFile == NO_HELP_FILE
      ? cHelpFile
      QUIT
   ENDIF

   IF .NOT. File( cHelpFile )
      cSearch := SearchHelpFile( cHelpFile )
      IF cSearch == NO_HELP_FILE
         ? "Help file not found:", cHelpFile
         QUIT
      ELSE
         cHelpFile := cSearch
      ENDIF
   ENDIF

   ? "Help file found:"
   ?
   ? cHelpFile
   ?
   ? "Press Y to open help file"
   ?
   WAIT "(Y/N) "

   IF Chr(LastKey()) $ "yY"
      OpenHelpFile( cHelpFile )
   ENDIF
RETURN

// Extracts the help file location from a type library
FUNCTION OLEHelpFile( oTypeLib )
   LOCAL cFile := NO_HELP_FILE
   LOCAL oOLE_CLASS

   FOR EACH oOLE_CLASS IN oTypeLib:Objects
      IF .NOT. Empty( oOLE_CLASS:HelpFile )
         cFile := oOLE_CLASS:HelpFile
         EXIT
      ENDIF
   NEXT
RETURN cFile

// Language specific help files are installed in
// sub-directories. Search sub-directories for help files.
FUNCTION SearchHelpFile( cHelpFile )
   LOCAL aDir
```

```
      aDir := DirectoryRecurse( cHelpFile )
   RETURN IIf( Empty( aDir ), NO_HELP_FILE, aDir[1,1] )

   // Open help file with associated viewer application
   FUNCTION OpenHelpFile( cHelpFile )
      LOCAL nRet, cPath, cFileName, cFileExt

      HB_FNameSplit( cHelpFile, @cPath, @cFileName, @cFileExt )

      nRet := _OpenHelpFile( cPath, cHelpFile )
   RETURN nRet

   #pragma BEGINDUMP
      #pragma comment( lib, "shell32.lib" )
      #include "hbapi.h"
      #include <windows.h>
      HB_FUNC( _OPENHELPFILE )
      {
        HINSTANCE hInst;
        LPCTSTR lpPath = (LPTSTR) hb_parc( 1 );
        LPCTSTR lpHelpFile = (LPTSTR) hb_parc( 2 );
        hInst = ShellExecute( 0, "open", lpHelpFile, 0, lpPath, SW_SHOW );
        hb_retnl( (LONG) hInst );
        return;
      }
   #pragma ENDDUMP
```

Extracting enumerated OLE constants

```
// This program extracts enumerated constants from a
// type library and creates a list of #define constants
// applicable for properties of an OLE Automation object.
// Output is directed to STDOUT.
//
// The complete file name of the file including the type library
// must be passed as command line parameter.
//
// e.g: C:\windows\system32\shdocvw.dll

   #translate ? [<x,...>]  => OutStd( Chr(13)+Chr(10) ); OutStd( <x> )
   #translate ?? [<x,...>] => OutStd( <x> )

   PROCEDURE Main( cFileName )
      LOCAL oTypeLib
      CLS

      IF Empty( cFileName )
         ? "Usage: OleConst.exe <typeLibraryFileName>"
         QUIT
      ENDIF

      TRY
         QOut( "Loading Type Library, please wait..." )
         oTypeLib := LoadTypeLib( cFileName, .T. )
      CATCH
         QOut( "Unable to load Type Library for:", cFileName )
         QUIT
      END

      IF .NOT. Empty( oTypeLib:Enumerations )
         ? "// "
         ? "// Enumerations Extracted from file:"
```

```
      ? "// "
      ? "//   ", cfileName
      ? "// "
      ?
      ?
      ListOleEnums( oTypeLib:Enumerations )
   ENDIF
RETURN

// Collect enumerations in hash for automatic alpha sort
STATIC PROCEDURE ListOleEnums( aEnums )
   LOCAL hEnums, oEnum, aConst, oConst
   LOCAL nAlign := 0
   LOCAL i, imax

   hEnums := Hash()
   HSetCaseMatch( hEnums, .F. )

   FOR EACH oEnum IN aEnums
      aConst := Array( Len( oEnum:constants ) )
      hEnums[ oEnum:name ] := aConst

      // Collect constants in array for value sort
      FOR EACH oConst IN oEnum:constants
         aConst[ HB_EnumIndex() ] := oConst
         nAlign := Max( nAlign, Len( oConst:name ) )
      NEXT
   NEXT

   // Output xHarbour code for CH file
   imax := Len( hEnums )
   FOR i:=1 TO imax
      ? "/*", PadL( " * ", nAlign + 20, "*" )
      ? PadR( " * ENUMERATION: "+ HGetKeyAt( hEnums, i ), nAlign + 20 ), "*"
      ? " *", PadL( " */", nAlign + 20, "*" )
      ListDefines( HGetValueAt( hEnums, i ), nAlign )
      ?
   NEXT
RETURN

// Output #define directives for an xHarbour #inlude file
STATIC PROCEDURE ListDefines( aConst, nPad )
   LOCAL i, imax := Len( aConst )
   LOCAL oConst

   // Sort by value, not by name, for readability
   ASort( aConst,,, {|o1,o2| o1:value < o2:value } )

   FOR EACH oConst IN aConst
      ? "#define ", PadR( oConst:name, nPad ), " 0x"+NumToHex(oConst:value,8)

      IF oConst:helpString <> NIL
         ?? "  //", oConst:helpString
      ENDIF
   NEXT
RETURN
```

Crypt()

Encrypts or decrypts a character string.

Syntax

```
Crypt( <cString>, <cPassWord> ) --> cResult
```

Arguments

`<cString>`

Theis is the character string to encrypt or decrypt.

`<cPassWord>`

A character string holding the password for the encryption or decryption. It should have at least six characters.

Return

The function encrypts a character string, and decrytps it using the same password. When a previously encrypted string is passed for *<cString>*, the return value is the original string.

Info

See also:	CharXOR(), HB_Crypt()
Category:	CT:String manipulation, Character functions
Source:	ct\ctcrypt.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The examples shows encryption end decryption of a character string

   PROCEDURE Main
      LOCAL cString  := "xHarbour compiler"
      LOCAL cPassWord:= "eeMCsq"
      LOCAL cCipher

      ? cCipher := Crypt( cString, cPassWord )

      ? cCipher == cString           // result: .F.

      cCipher := Crypt( cCipher, cPassWord )

      ? cCipher == cString           // result: .T.
   RETURN
```

CSetAtMuPa()

Queries or changes the multi-pass mode for At***() functions.

Syntax

```
CSetAtMuPa( [<lNewMode>] ) --> lOldMode
```

Arguments

`<lNewMode>`

Optionally, a logical value can be passed. .T. (true) switches the multi-pass mode on, while .F. (false) switches it off.

Return

The function returns the previous multi-pass mode as a logical value.

Description

The multi-pass mode is initially set to .F. (false) and can be switched on by passing .T. (true). The following At***() functions recognize the multi-pass mode: AfterAtNum(), AtNum(), AtRepl(), BeforAtNum(), NumAt(), WordToChar() and WordRepl().

If the multi-pass mode is On, these functions process a character string repeatedly until no further match can be found for a search condition.

Info

See also:	AfterAtNum(), AtNum(), AtRepl(), BeforAtNum(), CSetRef(), NumAt(), WordToChar(), WordRepl()
Category:	CT:String manipulation, Character functions
Source:	ct\ctstr.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates the effect of the multi-pass mode
// with function NumAt(). When the multi-pass mode is Off, the
// search continues at <nPos>+Len(<cSearch>), so that the search
// string is found only 2 times. When the mode is On, the search
// continues at <nPos>+1, so that the search string is found 4 times.

   PROCEDURE Main
      LOCAL cString := "aabbbbbccc"

      CSetAtMuPa( .F. )
      ? NumAt( "bb", cString )   // result: 2
      //      2_
      //  aabbbbbccc
      //    1_

      CSetAtMuPa( .T. )
      ? NumAt( "bb", cString )   // result: 4
      //     2_4_
      //  aabbbbbccc
      //    1_3_
   RETURN
```

CSetCent()

Queries or changes the SET CENTURY setting.

Syntax

```
CSetCent( [<lNewMode>] ) --> lOldMode
```

Arguments

<lNewMode>

This is an optional logical value defining the new value for the SET CENTURY setting. .T. (true) corresponds to ON, and .F. (false) means OFF.

Return

The function returns the previous SET CENTURY setting as a logical value.

Info

See also:	SET CENTURY
Category:	CT:Settings, Environment functions
Source:	ct\ctmisc.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

CSetCurs()

Queries or changes the SET CURSOR setting.

CSetCurs([*<lNewMode>*]) --> lOldMode !END

Arguments

`<lNewMode>`

This is an optional logical value defining the new value for the SET CURSOR setting. .T. (true) corresponds to ON, and .F. (false) means OFF.

Return

The function returns the previous SET CURSOR setting as a logical value.

Info

See also:	SET CURSOR
Category:	CT:Settings, Environment functions
Source:	ct\ctmisc.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

CSetKey()

Retrieves the code block associated with a key.

Syntax

```
CSetKey( <nKey> ) --> bCodeBlock
```

Arguments

<nKey>

This is the numeric Inkey() code of the key to query for an associated code block. Refer to the file Inkey.ch for #define constants that can be used for *<nKey>*.

Return

The function returns the code block associated with a key, or NIL, if the key is not associated with a code block.

Info

See also:	SET KEY, SetKey()
Category:	CT:Settings, Environment functions
Source:	ct\ctmisc.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

CSetRef()

Queries or changes the pass-by-reference mode for several string functions.

Syntax

```
CSetRef( [<lNewMode>] ) --> lOldMode
```

Arguments

<lNewSetting> [ ]

Optionally, a logical value can be passed. .T. (true) switches the pass-by-reference mode on, while .F. (false) switches it off.

Return

The function returns the previous pass-by-reference mode as a logical value.

Description

The previous pass-by-reference mode is initially set to .F. (false) and can be switched on by passing .T. (true). The following functions recognize the previous pass-by-reference mode:

Functions recognizing CSetRef()

AddAscii()	CharSort()	ReplAll()
Blank()	CharSwap()	ReplLeft()
CharAdd()	CharXor()	ReplRight()
CharAnd()	Crypt()	TokenLower()
CharMirr()	JustLeft()	TokenUpper()
CharNot()	JustRight()	WordRepl()
CharOr()	PosChar()	WordSwap()
CharRelRep()	PosRepl()	
CharRepl()	RangeRepl()	

If the pass-by-reference mode is On, these functions operate on the original input string when it is passed by reference, rather than creating a copy of the input string and returning the processed copy as a result. In addition, these functions do not return a character string but a logical value when the pass-by-reference mode is On. The input string passed by reference becomes the return value, thus saving memory on string operations.

Info

See also: CSetAtMuPa()
Category: CT:String manipulation, Character functions
Source: ct\ctstr.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates the effect of the pass-by-reference
// mode by sorting a character string.

   PROCEDURE Main
      LOCAL cString := "xHarbour compiler"
      LOCAL cInput  := cString
      LOCAL cResult

      // extra memory for cResult required
```

```
    CSetRef( .F. )
    cResult := CharSort( @cInput )

    ? Valtype( cResult )     // result: C
    ? cResult                // result:  Habceilmooprrrux
    ? cInput                 // result:  Habceilmooprrrux

    cInput := cString

    // no extra memory for cResult required
    CSetRef( .T. )
    cResult := CharSort( @cInput )

    ? Valtype( cResult )     // result: L
    ? cResult                // result: .F.
    ? cInput                 // result:  Habceilmooprrrux
RETURN
```

CSetSafety()

Retrieves and/or changes the safety switch used in CA-Tools file operations.

Syntax

```
CSetSafety( [<lNewMode>] ) --> lOldMode
```

Arguments

<lNewMode>

This is an optional logical value defining the safety switch for CA-Tools file operations. .T. (true) sets the safety switch ON, and .F. (false) sets it OFF.

Return

The function returns the previous safety switch as a logical value.

Description

When the file safety switch is set to ON, existing files are not overwritten. The switch is only recognized by CA-Tools compatible functions FileAppend(), FileCopy(), ScreenFile(), StrFile() and Volume().

Info

See also:	FileAppend(), FileCopy()
Category:	CT:Settings, File functions
Source:	ct\strfile.c
LIB:	xhb.lib
DLL:	xhbdll.dll

CStr()

Converts a value to a character string.

Syntax

```
CStr( <xValue> ) --> cString
```

Arguments

<xValue>

This is a value of any data type.

Return

The function returns a character string holding the converted value.

Description

Function CStr() accepts a value of any data type and returns a character string holding information about the value. This character string can be output to the screen or printer, and is of informational use only when *<xValue>* is a complex data type (Array, Code block, Hash or Object).

Info

See also:	CStrToVal(), DtoS(), HB_ValToStr(), Str(), Transform(), Valtype(), ValToPrg(), ValToPrgExp()
Category:	Conversion functions, Debug functions, xHarbour extensions
Source:	rtl\cstr.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates return values of function CStr()
// for all data types available in xHarbour.

   PROCEDURE Main
      LOCAL aArray   := { 1, 2, 3 }
      LOCAL bBlock   := {|| Time() }
      LOCAL cString  := "xHarbour"
      LOCAL dDate    := Date()
      LOCAL hHash    := { "A" => 1, "B" => 2 }
      LOCAL lLogic   := .T.
      LOCAL nNumber  := 1.2345
      LOCAL obj      := Get():new()
      LOCAL pPointer := ( @Main() )
      LOCAL undef    := NIL

      ? CStr( aArray   )     // result: { Array of 3 Items }
      ? CStr( bBlock   )     // result: {|| Block }
      ? CStr( cString  )     // result: xHarbour
      ? CStr( dDate    )     // result: 20061006
      ? CStr( hHash    )     // result: { Hash of 2 Items }
      ? CStr( lLogic   )     // result: .T.
      ? CStr( nNumber  )     // result:          1.2345
      ? CStr( obj      )     // result: { GET Object }
      ? CStr( pPointer )     // result: 4CB000
      ? CStr( undef    )     // result: NIL
   RETURN
```

CStrToVal()

Converts a character string to a value of specific data type.

Syntax

```
CStrToVal( <cString>, <cValtype> ) --> xValue
```

Arguments

<cString>

This is a character string to be converted to a value of a different data type.

<cValtype>

This is a single letter indicating the data type of the return value. The letters C, D, L, M, N, P and U are supported. Refer to function Valtype() for data type encoding.

Return

The function returns the converted value. If *<cValtype>* is not supported by CStrToVal(), a runtime error is raised.

Info

See also:	CStr(), StoD(), Val(), Valtype(), ValToPrg(), ValToPrgExp()
Category:	Conversion functions, xHarbour extensions
Source:	rtl\cstr.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

CtoBit()

Converts a character string to an integer based on a bit pattern.

Syntax

```
CtoBit( <cString> , <cBitPattern> ) --> nInteger
```

Arguments

<cString>

This is a character string to be traversed.

<cBitPattern>

This is a character string defining the bit pattern. It may contain a maximum of 16 characters. If a character of *<cBitPattern>* is found in *<cString>*, the bit at the corresponding position of the return value is set to 1.

Return

The function returns an integer value in the range of 0 to 65535.

Info

See also: BitToC(), NtoC()
Category: CT:NumBits, Numbers and Bits
Source: ct\numconv.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays results of CToBit() along with their
// binary representation.

   PROCEDURE Main

      ? n := CToBit( "C", "ABCD" )      // result: 2
      ? NtoC( n, 2, 4, "0" )            // result: 0010

      ? n := CToBit( "ACD", "ABCD" )    // result: 11
      ? NtoC( n, 2, 4, "0" )            // result: 1101
   RETURN
```

CtoD()

Converts a character string into a Date value

Syntax

```
CtoD( <cDateString> ) --> dDate
```

Arguments

<cDateString>

This is a character string formatted as a date. Digits for day, month and year plus the separator sign must comply with the current SET DATE format. If the year is not given as a four digit number, the rules for SET EPOCH apply for the date conversion.

Return

The function returns the converted Date value. If *<cDateString>* is not a valid date string, an empty date is returned.

Description

The conversion function CtoD() is used to create date values from string literals formatted as a Date. Formatting of the string literal must comply with the current SET DATE format. That is, digits for day, month and year in *<cDateString>* are interpreted according to the sequence of the letters dd, mm and yy in the SET DATE format string.

The conversion rules of SET EPOCH apply for the year when it is not specified with four digits.

Use function StoD() to convert a string to a date value independently of SET DATE.

Info

See also:	Date(), DtoC(), DtoS(), SET CENTURY, SET DATE, SET EPOCH, StoD(), TtoC()
Category:	Conversion functions, Date and time
Source:	rtl\dateshb.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example outlines the importance of the SET DATE format
// for the conversion of string literals to Date values.

   PROCEDURE Main

      SET DATE ITALIAN                // select Italian date format

      ? CMonth( CtoD( "05-01-06" ) )  // result: January

      SET DATE AMERICAN               // select US date format

      ? CMonth( CtoD( "05/01/06" ) )  // result: May

   RETURN
```

CtoDoW()

Returns the number of a week day from its name.

Syntax

```
CtoDoW( <cDayName> ) --> nDayOfWeek
```

Arguments

<cDayName>

This is a character string holding a week's day name as returned by CDoW(). The case of *<cDayName>* is ignored.

Return

The function returns the number of a week day, or 0 if an invalid name is specified.

Description

Function CtoDoW() is the reverse function of CDow(), which returns the name of a week day. Numbering days of a week starts with Sunday = 1 (see Dow()).

This function depends on the SET EXACT setting. If it is OFF, *<cDayName>* can be abbreviated. If set to ON, *<cDayName>* must conain a complete day name.

Info

See also:	CDoW(), CtoMonth(), NtoCDoW()
Category:	CT:DateTime, Date and time
Source:	ct\dattime2.c
LIB:	xhb.lib
DLL:	xhbdll.dll

CtoF()

Converts an 8 byte string to a floating point number.

Syntax

```
CtoF( <cFloat> ) --> nNumber
```

Arguments

<cFloat>

A character string holding the binary representation of a floating point number. It is returned from FtoC() or XtoC().

Return

The function returns the converted floating point number as a numeric value.

Info

See also: FtoC(), U2Bin(), W2Bin(), XtoC()
Category: CT:NumBits, Binary functions, Conversion functions, Numbers and Bits
Source: ct\ftoc.c
LIB: xhb.lib
DLL: xhbdll.dll

CtoMonth()

Returns the number of a month from its name.

Syntax

```
CtoMonth( <cMonthName> ) --> nMonth
```

Arguments

<cMonthName>

This is a character string holding a month's name as returned by CMonth(). The case of *<cMonthName>* is ignored.

Return

The function returns the numeric month of a year, or 0 if an invalid month name is specified.

Description

Function CtoMont() is the reverse function of CMonth(), which returns the name of a month.

This function depends on the SET EXACT setting. If it is OFF, *<cMonthName>* can be abbreviated. If set to ON, *<cMonthName>* must contain a complete month name.

Info

See also:	CMonth(), CtoDow(), NtoCMonth()
Category:	CT:DateTime, Date and time
Source:	ct\dattime2.c
LIB:	xhb.lib
DLL:	xhbdll.dll

CtoN()

Converts a string of digits to an integer of the specified base.

Syntax

```
CtoN( <cDigits> , ;
     [<nBase>]  , ;
     [<lNegative>]) --> nInteger
```

Arguments

<cDigits>

This is a character string to be converted. It may contain all characters allowed for the specified base. For example, if *<nBase>* is 16 the characters "0123456789ABCDEF" are allowed. The case of *<cDigits>* is ignored.

<nBase>

This is a numeric value specifying the base for the conversion. It must be in the range between 2 and 36, and defaults to base 10.

<lNegative>

If .T. (true) is passed, the function takes a minus sign into account. The default is .F. (false), so that only positive values are returned.

Return

The function returns a numeric integer value.

Info

See also: NtoC(), NumToHex()
Category: CT:NumBits, Numbers and Bits
Source: ct\numconv.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays results of number conversions.

   PROCEDURE Main
      ? CtoN( "123" )        // result:        123

      ? CtoN( "123A", 16 )   // result:       4666
      ? NumToHex( 4666 )     // result: 123A

      ? CtoN( "1101", 2 )    // result:         13
      ? NtoC( 11    , 2 )    // result: 1101
   RETURN
```

CtoT()

Converts a character string into a DateTime value

Syntax

```
CtoT( <cDateTimeString> ) --> dDateTime
```

Arguments

<cDateTimeString>

This is a character string formatted as a DateTime value. Digits for day, month and year plus the separator sign must comply with the current SET DATE format. If the year is not given as a four digit number, the rules for SET EPOCH apply for the DateTime conversion.

Return

The function returns the converted DateTime value. If *<cDateTimeString>* is not a valid DateTime string, an empty DateTime value is returned.

Description

The conversion function CtoT() is used to create DateTime values from string literals formatted as a DateTime. Such literals are returned from function TtoC(). Formatting of the string literal must comply with the current SET DATE format. That is, digits for day, month and year in *<cDateString>* are interpreted according to the sequence of the letters dd, mm and yy in the SET DATE format string.

The Time part of the DateTime value must be formatted as "hh:mm:ss.ccc". (refer to SET TIME). If the Time part is omitted from *<cDateTimeString>*, the Time part in the return value is set to 00:00:00.

The conversion rules of SET EPOCH apply for the year when it is not specified with four digits.

Use function StoT() to convert a string to a DateTime value independently of SET DATE.

Info

See also:	CtoD(), DateTime(), SET CENTURY, SET DATE, SET TIME, SET EPOCH, StoT(), TtoC(), TtoS()
Category:	Conversion functions, Date and time, xHarbour extensions
Source:	rtl\dateshb.c
LIB:	xhb.lib
DLL:	xhbdll.dll

CurDir()

Returns the current directory of a drive

Syntax

```
CurDir( [<cDrive>] ) --> cDirPath
```

Arguments

<cDrive>

A single character, optionally followed by a colon, specifies the disk drive to query. If omitted, the current drive is queried.

Return

The function returns the current directory of the specified disk drive as a character string without leading and trailing directory separator.

Description

The function is used to determine the current directory of a disk drive. If the drive *<cDrive>* does not exist or the root directory is current, CurDir() returns a null string ("").

Note that CurDir() reports the current directory of the operating system for a specified drive. It does not report directories set with SET DEFAULT or SET PATH.

Use function DirChange() to change the current directory.

Info

See also:	CurDirX(), CurDrive(), DirChange(), Directory(), DirRemove(), File(), Getenv(), MakeDir()
Category:	Directory functions, File functions
Source:	rtl\philes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example lists the current directories for drive C: to I:

   PROCEDURE Main
      LOCAL nDrive := Asc( "C" )
      LOCAL i

      FOR i:=1 TO 7
         ? Chr(nDrive) + ":\"+ CurDir( Chr(nDrive) )
         nDrive ++
      NEXT
   RETURN
```

CurDirX()

Returns the current directory of a drive including directory separators.

Syntax

```
CurDir( [<cDrive>] ) --> cDirPath
```

Arguments

<cDrive>

A single character, optionally followed by a colon, specifies the disk drive to query. If omitted, the current drive is queried.

Return

The function returns the current directory of the specified disk drive as a character string including leading and trailing directory separator.

Description

CurDirX() works exacly like CurDir() but adds a leading and trailing directory separator to the returned directory string

Info

See also:	CurDir(), CurDrive(), DirChange(), Directory(), DirRemove(), File(), Getenv(), MakeDir()
Category:	Directory functions, File functions, xHarbour extensions
Source:	rtl\philes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example compares CurDir() and CurDirX()

   PROCEDURE Main
      LOCAL nDrive := Asc( "C" )
      LOCAL i

      // directory with CurDir()
      FOR i:=1 TO 7
         ? Chr( nDrive ) + ":\"+ CurDir( Chr(nDrive) ) + "\"
         nDrive ++
      NEXT

      nDrive := Asc( "C" )
      // directory with CurDirX()
      FOR i:=1 TO 7
         ? Chr( nDrive ) + ":"+ CurDirX( Chr(nDrive) )
         nDrive ++
      NEXT

   RETURN
```

CurDrive()

Determines or changes the current disk drive

Syntax

```
CurDrive( [<cNewDrive>] ) --> cOldDrive
```

Arguments

<cNewDrive>

A single character, optionally followed by a colon, specifies the disk drive to select as current. If omitted, the disk drive is not changed.

Return

The function returns the a single letter specifying the disk drive that is current before the function is called (the previous disk drive).

Description

CurDrive() queries the current disk drive and optionally changes to a new one. The new disk drive to change to is specified with *<cNewDrive>*.

Use function IsDisk() to check for the existence of a disk drive.

Info

See also:	CurDir(), DiskChange(), DiskName(), DiskSpace(), Getenv(), IsDisk(), SET DEFAULT, SET PATH
Category:	Disks and Drives, File functions
Source:	rtl\philesx.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example queries and changes the current disk drive
// and displays the current directory for each drive.

   PROCEDURE Main

      ? CurDrive( "C:" )              // result: D
      ? CurDrive()                    // result: C

      ? CurDir()                      // result: temp
      ? CurDrive( "D" )               // result: C

      ? CurDir()                      // result: xhb\source\samples
      ? CurDrive()                    // result: D

      ? CurDir()                      // result: xhb\source\samples

   RETURN
```

CurrentGet()

Returns the position of the current Get field in the Getlist array.

Syntax

```
CurrentGet() --> nCurrentGetPos
```

Return

The function returns the ordinal position of the current Get field, held in the PUBLIC *Getlist* array, as a numeric value.

Info

See also:	@...GET, CountGets(), GetFldCol(), GetFldRow()
Category:	CT:GetSys, Get system
Source:	ct\getinfo.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Date()

Returns the current date from the operating system.

Syntax

```
Date() --> dDate
```

Return

The function returns the operating system's date setting.

Description

The Date() function retrieves the current system date as reported by the system clock. The returned value is of Valtype() == "D".

The display of Date values depends on the SET DATE format.

Info

See also: CtoD(), DateTime(), DtoC(), DtoS(), SET CENTURY, SET DATE, SET EPOCH, StoD(), Time()
Category: Date and time
Source: rtl\dateshb.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example extracts various information from the current date

   PROCEDURE Main

      ? CDoW( Date() )
      ?? ",", LTrim( Str( Day( Date() ) ) )
      ?? ".", CMonth( Date() )
      ?? ""  , LTrim( Str( Year( Date() ) ) )

      // result: Wednesday, 25. January 2006

      ? Date() + 7                      // result: 02/01/06
   RETURN
```

DateTime()

Returns the current date and time from the operating system.

Syntax

```
DateTime() --> dDateTime
```

Return

The function returns the operating system's date and time setting as a DateTime value.

Description

The DateTime() function retrieves the current system date and time as reported by the system clock. The returned value is of Valtype() == "D". Use function HB_IsDateTime() to ditinguish a regular Date() value from a DateTime() value.

The display of DateTime values depends on the SET DATE format.

Note: refer to the DateTime operator \{^ \} for a detailed explanation of DateTime values.

Info

See also:	{^ }, CtoT(), Date(), SET CENTURY, SET DATE, SET EPOCH, StoT(), Time(), TtoC(), TtoS()
Category:	Conversion functions, Date and time, xHarbour extensions
Source:	rtl\dateshb.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Day()

Extracts the numeric day number from a Date value.

Syntax

```
Day( <dDate> ) --> nDay
```

Arguments

<dDate>

This is an expression returning a value of data type Date.

Return

The function returns the numeric day number of *<dDate>*. When *<dDate>* is an empty date, the function returns zero.

Description

This function is used to extract the numeric day of a month from a Date value.

Info

See also:	CDow(), CMonth(), CtoD(), Date(), Days(), DoW(), DtoC(), DtoS(), Hour(), Minute(), Month(), Secs(), Str(), Year()
Category:	Conversion functions, Date and time
Source:	rtl\dateshb.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows results of the function Day()

   PROCEDURE Main

     ? Date()                          // result: 01/25/06
     ? Day( Date())                    // result: 25
     ? Day( Date() + 7 )               // result:  1

   RETURN
```

Days()

Calculates the number of days from elapsed seconds.

Syntax

```
Days( <nSeconds> ) --> nDays
```

Arguments

<nSeconds>

A numeric value indicating the number of seconds to convert to days.

Return

The function returns the converted value as an integer number of days.

Description

This function converts a number of *<nSeconds>* seconds to their equivalent number of days as integer value. One day has 86400 seconds.

Info

See also:	Date(), ElapTime(), Seconds(), Secs(), Time()
Category:	Date and time
Source:	rtl\samples.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows the conversion of seconds representing time spans
// of Minute, Hour, Day and Year into equivalent number of days.

   PROCEDURE Main

      ? Days( 60 )                    // result: 0
      ? Days( 3600 )                  // result: 0
      ? Days( 86400 )                 // result: 1
      ? Days( 31536000 )              // result: 365

   RETURN
```

DaysInMonth()

Returns the number of days in a month.

Syntax

```
DaysInMonth( <nMonth>, [<lLeapYear>] ) --> nDaysInMonth
```

Arguments

<nMonth>

This is a numeric value in the range of 1 to 12 specifying the month.

<lLeapYear>

If *<nMonth>* is 2 (February), the second, logical parameter indicates a leap year. This can be the return value of IsLeap().

Return

The function returns the number of days in the specified month as a numeric value.

Info

See also: DaysToMonth(), IsLeap(), Month()
Category: CT:DateTime, Date and time
Source: ct\dattime2.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays the days in the current month.

   PROCEDURE Main
      LOCAL dDate := Date()

      ? DaysInMonth( Month(dDate), IsLeap() )
      // result: 28, 29, 30 or 31

   RETURN
```

DaysToMonth()

Returns the number of days from first January to the beginning of a month.

Syntax

```
DaysToMonth( <nMonth>, [<lLeapYear>] ) --> nDays
```

Arguments

<nMonth>

This is a numeric value in the range of 1 to 12 specifying the month.

<lLeapYear>

If *<nMonth>* is 2 (February), the second, logical parameter indicates a leap year. This can be the return value of IsLeap().

Return

The function returns the number of days from first January to the first day of the specified month as a numeric value.

Info

See also:	Day(), DaysInMonth(), DoY(), IsLeap(), Month()
Category:	CT:DateTime, Date and time
Source:	ct\dattime2.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays results of DaysToMonth()

   PROCEDURE Main
      LOCAL dDate := StoD( "20061101" )

      ? DaysToMonth( 1 )         // result: 0
      ? DaysToMonth( 3 )         // result: 59
      ? DaysToMonth( 3, .T. )    // result: 60

      ? DaysToMonth( 11 )        // result: 304

      ? DoY( dDate )             // result: 305
   RETURN
```

DbAppend()

Appends a new record to a database open in a work area.

Syntax

```
DbAppend( [<lUnlockRecords>] ) --> NIL
```

Arguments

`<lUnlockRecords>`

This parameter defaults to .T. (true). It causes all pending record locks set with RLock() or DbRLock() be released before a new record is appended to the database. Pass .F. (false) to keep all record locks intact.

Return

The return value is always NIL.

Description

The DbAppend() function adds a new record to the end of the database in the current work area. Use an aliased expression to append a record in a different work area.

All fields of the new record are filled with empty data of their respective data types. That is, Character fields are filled with blank spaces, Date fields contain empty dates, logical fields contain .F. (false) and numeric fields zero. Memo fields contain a null string ("").

The DbAppend() function is guaranteed to add a new record when the database file is open for EXCLUSIVE access by the application. This, however, is rarely the case. In a multi-user environment, databases are usually open for SHARED access. As a result, DbAppend() tries to lock the new record so that no other user can change it during the operation. If this lock fails, there is no new record available for the application. Instead, the function NetErr() is set to .T. (true) indicating a failure of locking the new record. A failure of this kind can happen, for example, when another application has obtained a file lock prior to DbAppend(). Such an error condition is gracefully handled if NetErr() is checked after DbAppend().

If DbAppend() has successfully locked the new record, all other records locks set in the database are released by default. To keep pending record locks in place, the value .F. (false) must be passed to DbAppend().

Info

See also:	APPEND BLANK, DbRLock(), DbUnlock(), DbUnlockAll(), NetAppend(), RLock()
Category:	Database functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows a typical coding pattern for appending a new
// record to a database open in SHARED mode

   PROCEDURE Main
      USE Customer SHARED ALIAS Cust
      SET INDEX TO Cust01, Cust02

      Cust->( DbAppend() )

      IF NetErr()
```

```
      ? "Unabled to append new record"
   ELSE
      ? "Fill data to new record"
      REPLACE Cust->Firstame WITH "Paul"
      REPLACE Cust->Lastname WITH "Newman"
      DbCommit()
      DbUnlock()
   ENDIF

   WAIT
   Browse()

   CLOSE Customer
RETURN
```

DbClearFilter()

Clears the current filter condition in a work area.

Syntax

```
DbClearFilter() --> NIL
```

Return

The return value is always NIL.

Description

The function clears an active filter condition in the current work area. As a result, all records previously filtered become visible again in database navigation.

Use an aliased expression to clear a filter in a different work area.

Info

See also:	DbFilter(), DbSetFilter(), SET DELETED, SET FILTER, SET SCOPE
Category:	Database functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates visibility of filtered records

   PROCEDURE Main

      USE Customer ALIAS Cust SHARED
      INDEX ON Upper(Lastname+Firstname) TO Cust01

      GO TOP
      ? Cust->Lastname                  // result: Alberts

      SET FILTER TO Cust->Lastname > "E"

      GO TOP
      ? Cust->Lastname                  // result: Feldman

      Cust->(DbClearFilter())

      GO TOP
      ? Cust->Lastname                  // result: Alberts

      CLOSE Cust
   RETURN
```

DbClearIndex()

Closes all indexes open in the current work area.

Syntax

```
DbClearIndex() --> NIL
```

Return

The return value is always NIL.

Description

The function DbClearIndex() closes all indexes open in the current work area. When the operation is complete, all pending index updates are written to disk and the records become accessible in physical order again.

Use an aliased expression to close indexes in a different work area.

Note: This function is superseded by the OrdListClear() function.

Info

See also:	DbCreateIndex(), DbReindex(), DbSetIndex(), DbSetOrder(), OrdListClear(), SET INDEX
Category:	Database functions, Index functions
Source:	rdd\rddord.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// refer to the example for OrdListClear()
```

DbClearRelation()

Clears all active relations in a work area.

Syntax

```
DbClearRelation() --> NIL
```

Return

The return value is always NIL.

Description

DbClearRelation() clears all active relations in the current work area. Database navigation is then no longer synchronized with child work areas.

The function has the same effect as specifying SET RELATION TO with no further argument. Refer to the SET RELATION command for more information.

Use an aliased expression to clear relations in a different work area.

Info

See also:	DbRelation(), DbSetRelation(), OrdSetRelation(), SET RELATION
Category:	Database functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

DbCloseAll()

Close all open files in all work areas.

Syntax

```
DbCloseAll() --> NIL
```

Return

The return value is always NIL.

Description

The function closes all open databases and all associated files like index or memo files. In addition, all format files are closed and work area #1 is selected as the current work area.

Info

See also:	CLOSE, DbCloseArea(), DbUseArea(), SELECT, Select(), USE, Used()
Category:	Database functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates closing of all databases with one
// function call.

   PROCEDURE Main()

      USE Test NEW
      ? Select()                      // result: 1

      USE Test1 NEW
      ? Select()                      // result: 2

      ? (1)->(Used())                 // result: .T.
      ? (2)->(Used())                 // result: .T.

      DbCloseAll()

      ? Select()                      // result: 1
      ? (1)->(Used())                 // result: .F.
      ? (2)->(Used())                 // result: .F.
   RETURN
```

DbCloseArea()

Closes a database file in a work area.

Syntax

```
DbCloseArea() --> NIL
```

Return

The return value is always NIL.

Description

The DbCloseArea() function closes the database open in the current work area and all associated files, like index or memo files. Use an aliased expression to close a database in a different work area.

Closing a database causes all pending file buffers being flushed to disk before files are closed. Also, all file and record locks are released when files are closed.

Info

See also: CLOSE, DbCloseAll(), DbCommit(), DbUseArea(), USE
Category: Database functions
Source: rdd\dbcmd.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
PROCEDURE Main()
   USE Customer ALIAS Cust
   ? (1)->(Used())                  // result: .T.

   DbEdit()

   Cust->(DbCloseArea())

   ? (1)->(Used())                  // result: .F.
RETURN
```

DbCommit()

Writes database and index buffers to disk.

Syntax

```
DbCommit() --> NIL
```

Return

The return value is always NIL.

Description

The DbCommit() function writes all database and index buffers held in memory to disk. The function operates in the current work area, unless it is used in an aliased expression.

Memory buffers serve as cache for field variables and index values so that assignments to field variables become immediately visible to an application without any file I/O. When a database is open in SHARED mode, however, such changes are not visible to other applications in a multi-user environment until the changes are commited to disk.

Info

See also:	CLOSE, COMMIT, DbCloseAll(), DbCommitAll(), DbRUnlock(), DbUnlock(), NetCommitAll(), RLock()
Category:	Database functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows a typical coding pattern for updating records
// in a multi-user environment.

   PROCEDURE Main()
      LOCAL cLastName, cFirstName

      USE Customer ALIAS Cust SHARED NEW
      SET INDEX TO Cust01, Cust02

      cFirstname := Cust->Firstname
      cLastname  := Cust->Lastname

      @ 10, 10 SAY "First name" GET cFirstname
      @ 11, 10 SAY " Last name" GET cLastname
      READ

      IF Updated() .AND. RLock()       // obtain record lock
         REPLACE Cust->Firstname WITH cFirstname
         REPLACE Cust->Lastname  WITH cLastname
         DbCommit()                    // flush memory buffers
         DbUnlock()                    // release record lock
      ENDIF

      CLOSE Cust
   RETURN NIL
```

DbCommitAll()

Writes database and index buffers of all used work areas to disk.

Syntax

```
DbCommitAll() --> NIL
```

Return

The return value is always NIL.

Description

The DbCommitAll() function writes buffers of all work areas to disk. It performs the same operations as DbCommit(), except that all work areas with open databases are iterated in one function call.

Info

See also:	CLOSE, COMMIT, DbCloseAll(), DbCommit(), DbRUnlock(), DbUnlock(), NetCommitAll(), RLock()
Category:	Database functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// refer to function DbCommit()
```

DbCopyExtStruct()

Creates a structure extended database file.

Syntax

```
DbCopyExtStruct( <cDatabaseExt> ) --> lSuccess
```

Arguments

TO <cDatabaseExt>

This is name of the database file to create. It can include path and file extension. When no path is given, the file is created in the current directory. The default file extension is DBF.

Return

The function returns .T. (true) when the structure extended file is sucessfully created, or .F. (false) when no database is open in a work area.

Description

Function DbCopyExtStruct() creates a new structure extended database file. It is the functional equivalent of command COPY STRUCTURE EXTENDED. Refer to this command for more information on structure extended database files.

Info

See also:	COPY STRUCTURE, COPY STRUCTURE EXTENDED, CREATE, CREATE FROM, DbCopyStruct(), DbCreate(), DbStruct()
Category:	Database functions, xHarbour extensions
Source:	rdd\dbstrux.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example creates a structure extended database file
// and displays its contents.

   PROCEDURE Main

      USE Customer
      DbCopyExtStruct( "Test.dbf" )

      USE Test

      Browse()

      USE
   RETURN
```

DbCopyStruct()

Creates a new database based on the current database structure.

Syntax

```
DbCopyStruct( <cDatabase>, [<aFieldList>] ) --> NIL
```

Arguments

`<cDatabase>`

This is a character expression holding the name of the database file to create. It can include path and file extension. When no path is given, the file is created in the current directory.

`<aFieldList>`

Optionally, a one dimensional array can be specified which contains the field names of the current work area to include in the new database. If *<aFieldList>* is omitted, all fields are included.

Return

The return value is always NIL.

Description

Function DbCopyStruct() create a new, empty database file with a structure that is based on the database currently open in a work area. If *<aFieldList>* is empty, the new file has the same structure as the open database. Otherwise, the new file contains only the fields listed in *<aFieldList>*.

DbCopyStruct() can be used to create a subset of the currently open database, based on a given field list.

Info

See also:	COPY STRUCTURE, COPY STRUCTURE EXTENDED, DbCopyExtStruct(), DbCreate(), DbStruct()
Category:	Database functions, xHarbour extensions
Source:	rdd\dbstrux.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example creates a new database holding only two fields of
// the Customer database.

   PROCEDURE Main

     USE Customer
     DbCopyStruct( "Test.dbf", { "Firstname", "Lastname" } )

     USE Test
     APPEND FROM Customer

     Browse()

     USE
  RETURN
```

DbCreate()

Creates an empty database from a structure definition array.

Syntax

```
DbCreate( <cDatabase> , ;
          <aStructure>, ;
         [<cDriver>]  , ;
         [<lNewArea>] , ;
         [<cAlias>]     ) --> NIL
```

Arguments

<cDatabase>

This is a character expression holding the name of the database file to create. It can include path and file extension. When no path is given, the file is created in the current directory.

<aStructure>

A variable holding the reference of a two-dimensional array that has at least four columns. The first column must contain character strings holding the field names, the second column specifies the field type using single letters of C, D, L, N, M, the third column holds numeric values with the length of the fields, and the fourth column is only relevant for numeric fields. It specifies the number of decimal places that can be stored in a numeric field.

The following #define constants exist in file DBSTRUCT.CH to address each column of *<aStructure>*.

#define constants for the structure definition array

Constant	Value	Meaning
DBS_NAME	1	Field name
DBS_TYPE	2	Field type
DBS_LEN	3	Field length
DBS_DEC	4	Field decimals

Note: only the first four columns of *<aStructure>* are used. If the array has more columns, they are ignored.

<cDriver>

The replaceable database driver to use for creating the new database file can be specified as a character string. It defaults to the return value of RddSetDefault().

<lOpenNew>

This optional parameter instructs the RDD if and how to open the database after creation. Three values are possible:

Open modes for new database files

Value	Description
NIL (*)	the database is not opened.
.F.	the database is opened in the current work area.
.T.	the database is opened in a new, unused work area.

(*) default

<cAlias>

A character expression specifying the symbolic alias name to assign to the work area where <*cDatabase*> is opened can be optionally passed. If not given, <*cAlias*> is build from the file name of <*cDatabase*> without extension.

Return

The return value is always NIL.

Description

This function creates the database file specified as <*cDatabase*> from the two-dimensional array <*aStructure*>. If no file extension is included with <*cDatabase*> the .DBF extension is assumed. The array specified with <*aStructure*> must contan at least four columns as outlined above.

The DbCreate( ) function does not use the decimal field to calculate the length of a character field that can store than 256 characters. Values of up to the maximum length of a character field, which is 65,519 bytes, can be assigned directly to the DBS_LEN column of <*aStructure*>.

The <*cDriver*> parameter specifies the name of the Replaceable Database Driver to use for database creation. If not specified, the return value of RddSetDefault() is used.

The <*lOpenNew*> parameter specifies if the already created database is to be opened, and where. If NIL, the file is not opened. If .T. (true), it is opened in a new work area, and if .F. (false) it is opened in the current work area, closing any file already occupying that area.

The <*cAlias*> parameter specifies the alias name for the newly opened database.

Info

See also:	AFields(), COPY STRUCTURE EXTENDED, CREATE, CREATE FROM, DbStruct()
Category:	Database functions
Header:	Dbstruct.ch
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows how to create a new database from a
// structure definition array using the DBFCDX driver.

   REQUEST DBFCDX

   PROCEDURE Main()
      LOCAL aStruct := { ;
                { "CHARACTER", "C", 25, 0 }, ;
                { "NUMERIC"  , "N",  8, 0 }, ;
                { "DOUBLE"   , "N",  8, 2 }, ;
                { "DATE"     , "D",  8, 0 }, ;
                { "LOGICAL"  , "L",  1, 0 }, ;
                { "MEMO1"    , "M", 10, 0 }, ;
                { "MEMO2"    , "M", 10, 0 }  ;
             }

      DbCreate( "testdbf", aStruct, "DBFCDX", .T., "MYALIAS" )

      Browse()
   RETURN
```

DbCreateIndex()

Creates a new index and/or index file.

Syntax

```
DbCreateIndex( <cIndexFile> , ;
                <cIndexExpr> , ;
               [<bIndexExpr>], ;
               [<lUnique>]   , ;
               [<cIndexName>]  ) --> NIL
```

Arguments

`<cIndexFile>`

<cIndexFile> is a character string with the name of the file that stores the new index. When the file extension is omitted, it is determined by the database driver that creates the file.

`<cIndexExpr>`

This is a character expression which describes the index expression in textual form. The expression is evaluated for the records in the current work area. The return value of *<cIndexExpr>* determines the logical order of records when the index is the controlling index. The data type of the index may be Character, Date, Numeric or Logical. The maximum length of an index expression and its value is determined by the replaceable database driver used to create the index.

`<bIndexExpr>`

It is the same like *<cIndexExpr>* but is specified as a code block that can be evaluated. If omitted, the code block is created from *<cIndexExpr>*.

`<lUnique>`

If the optional parameter is set to .T. (true), it suppresses inclusion of records that yield duplicate index values. When an index value exists already in an index, a second record resulting in the same index value is not added to the index. When *<lUnique>* is omitted, the current SET UNIQUE setting is used as default.

`<cIndexName>`

This is a character string holding the symbolic name of the index to create in an index file. It is analogous to the alias name of a work area. Support for *<cIndexName>* depends on the RDD used to create the index. Usually, RDDs that are able to maintain multiple indexes in one index file support symbolic index names, such as DBFCDX, for example.

Return

The return value is always NIL.

Description

The index function DbCreateIndex() creates an index for the database open in the current work area. Use an aliased expression to create an index in a different work area.

If the work area has indexes open, they are closed prior to creating the new index. During index creation, the code block *<bIndexExpr>* is evaluated for each record of the work area and its return value is added to the index. When index creation is complete, the new index becomes the controlling index. As a result, records are viewed in logical index order and no longer in physical order.

Info

See also: DbClearIndex(), DbReindex(), DbSetIndex(), DbSetOrder(), INDEX, OrdCreate(), OrdListAdd(), OrdSetFocus()
Category: Database functions, Index functions
Source: rdd\rddord.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example creates an index file Emp01 on the expression
// Upper(Lastname+Firstname). Note the two notations for the index
// expression.

   PROCEDURE Main
      USE Employees NEW EXCLUSIVE
      DbCreateIndex( "Emp01", ;
                     "Upper(Lastname+Firstname)", ;
                     {|| Upper(Lastname+Firstname) } ;
                   )

      ? IndexKey()                    // result: Upper(Lastname+Firstname)

      Browse()
   RETURN
```

DbDelete()

Marks records for deletion.

Syntax

```
DbDelete() --> NIL
```

Return

The return value is always NIL.

Description

This function marks a record for deletion in the current or aliased work area. If SET DELETED is ON, the record will continue to be visible until the record pointer in the work area is moved.

Note that DbDelete() sets a flag on the current record which marks it only as deleted. Call PACK to physically remove records marked as deleted from the database.

Once set, the Deleted flag can be removed again using the DbRecall() function.

In a networking situation, this function requires the current record be locked prior to calling the DbDelete() function.

Info

See also: DbRecall(), DELETE, Deleted(), NetDelete(), PACK, RECALL
Category: Database functions
Source: rdd\dbcmd.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example shows a typical coding pattern used for setting the
// deletion flag in a networking situation.

   PROCEDURE Main
      LOCAL cCustID := "10"

      USE Customer ALIAS Cust INDEX CustId NEW

      IF Cust->(DbSeek( cCustId ))
         IF Cust->(RLock())
            Cust->(DbDelete())
            Cust->(DbCommit())
            Cust->(DbUnLock())
         ENDIF
      ENDIF

      USE
   RETURN
```

DbEdit()

Browse records in a table.

Syntax

```
DbEdit( [<nTop>]            , ;
        [<nLeft>]           , ;
        [<nBottom>]         , ;
        [<nRight>]          , ;
        [<aColumns>]        , ;
        [<bcUserFunc>]      , ;
        [<xSayPictures>]    , ;
        [<xColumnHeaders>] , ;
        [<xHeadingSep>      , ;
        [<xColumnSep>       , ;
        [<xFootingSep>      , ;
        [<xColumnFootings> , ;
        [<xColumnPreBlock> , ;
        [<xColumnPostBlock>  ) --> NIL
```

Arguments

`<nTop> and <nLeft>`

Numeric values indicating the screen coordinates for the upper left corner of the DbEdit() window. Both default to zero.

`<nBottom>`

Numeric value indicating the bottom row screen coordinate. It defaults to MaxRow().

`<nRight>`

Numeric value indicating the right column screen coordinate. It defaults to MaxCol().

`<aColumns>`

An array whose elements define the data displayed in each browser column. Data definition is very flexible and can be accomplished using different data types stored in each element of *<aColumns>*.

Data definition for DbEdit()

Data type	Description
Character	A macro expression whose return value is displayed.
Code block	A code block whose return value is displayed.
Array	A two element array. The first element contains a character string or a code block as decribed above, the second element is a code block compliant with TBColumn():colorBlock.

If *<aColumns>* is not specified, the browser displays all fields of the current work area in its columns.

`<bcUserFunction>`

A character string holding the symbolic name of a user function, or a code block which calls a user function. The user function is called with three parameters: the current DbEdit() mode (numeric), the current column position (numeric) and the Tbrowse() object which represents the browser. The DbEdit() behavior depends on the presence or absence of a user function. See the discussion below.

`<xSayPictures>`

This parameter can be specified as a character string defining a picture formatting string for the entire browser, or an array of Len(<*aColumn*>) elements that hold picture strings for each individual browser column. Refer to @...SAY or Transform() for a description of picture formatting strings. If <*xSayPictures*> is not specified, proper data formatting is automatically applied.

`<xColumnHeaders>`

This parameter defines the headers displayed for each column. If it is not specified, headers are derived from the contents of <*aColumns*>. A single character string is used as header for each column, while an array of Len(<*aColumn*>) elements containing character strings defines individual headers for each column. Note that when a semi-colon (;) is contained in a heading string, it is interpreted as line break character, so that column headings consisting of multiple lines can be defined.

`<xHeadingSep>`

<*xHeadingSep*> is a parallel array of character expressions whose character values are used to draw horizontal lines separating column headings from the data display area. Specifying <*xHeadingSep*> as a single character expression defines the same heading separator for all columns. By default. a single-line separator is displayed.

`<xColumnSep>`

This parameter has the same meaning and data types as <*xHeadingSep*> except that it defines the vertical separating lines between columns. It defaults to a single separating line.

`<xFootingSep>`

The parameter is used like <*xHeadingSep*> for the display of a horizontal line separating the footing area of columns from the data area of the browser. If not specified, no footing separator is displayed.

`<xColumnFootings>`

The parameter is used like <*xColumnHeaders*> for the display of column footings. If omitted, no column footings are displayed.

`<aColumnPreBlock>`

An optional array of Len(<*aColumn*>) elements containing logical values or code blocks can be specified. A code block must return a logical value. <*aColumnPreBlock*> is used for prevalidating data entry in the current browser cell. It is compliant with TBColumn:preBlock.

`<aColumnPostBlock>`

The parameter is used in the same way as <*aColumnPreBlock*>, but is used for post-validation and complies with TBColumn:postBlock.

Return

DbEdit() returns .T. (true) if data can be displayed, otherwise .F. (false) is returned.

Description

The DbEdit() function displays a database browser in a console window navigating the record pointer in the current work area, unless it is called in an aliased expression. Data from a work area is displayed in the browser columns, and can be defined very flexible using the <*aColumns*> array. If no columns are explicitely defined, the browser displays all fields from the work area. The width of each column is determined at initial display by the number of characters required for displaying headers (<*xColumnHeaders*>), data area (<*aColumns*>) and footings (<*xColumnFootings*>).

When called without a user fundtion, DbEdit() handles the following keys for navigating the browse cursor in the display:

Table navigation with DbEdit()

Key	Description
Up	Move up one row (previous record)
Down	Move down one row (next record)
PgUp	Move to the previous screen
PgDn	Move to the next screen
Left	Move one column to the left (previous field)
Right	Move one column to the right (next field)
Home	Move to the leftmost visible column
End	Move to the rightmost visible column
Ctrl+Home	Move to the leftmost column
Ctrl+End	Move to the rightmost column
Ctrl+Left	Pan one column to the left
Ctrl+Right	Pan one column to the right
Ctrl+PgUp	Move to the top of the file
Ctrl+PgDn	Move to the end of the file
Ctrl+Up	Move current column to the left
Ctrl+Down	Move current column to the right
Enter	Terminate DbEdit()
Esc	Terminate DbEdit()
Left click *)	Go to clicked row and column

**) Mouse keys must be activated with SET EVENTMASK*

DbEdit() user function

If a user function *<bcUserFunction>* is specified, all keys but Enter and Esc are automatically handled by DbEdit(). After each key stroke, the user function is called with three parameters :

1) The current DbEdit() mode (see table below).

2) A numeric value indicating the current column position of the browse cursor.

3) The TBrowse() object representing the browser.

DbEdit() modes passed to the user function indicate the internal state. They are encoded as numeric values for which #define constants exist in the DBEDIT.CH file.

DbEdit() modes

Constant	Mode	Description
DE_INIT	-1	Initial browse display
DE_IDLE	0	Idle, any cursor movement keystrokes are handled and no keystrokes are pending
DE_HITTOP	1	Attempt to move the record pointer past top of file
DE_HITBOTTOM	2	Attempt to move the record pointer past bottom of file
DE_EMPTY	3	Work area has no records
DE_EXCEPT	4	Key exception

The return value of the user function instructs DbEdit() how to proceed with browsing. The following #define constants from DBEDIT.CH can be used as return values:

User-function return values for DbEdit()

Constant	Value	Description
DE_ABORT	0	Abort DbEdit().
DE_CONT	1	Continue DbEdit() as is.
DE_REFRESH	2	Force reread/redisplay of all data rows.

Info

See also: AChoice(), Browse(), MemoEdit(), TBrowse(), TBrowseDB(), TBColumn(), USE
Category: UI functions, Database functions
Header: dbedit.ch
Source: rtl\dbedit.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example calls DbEdit() with a user function and initializes
// the TBrowse object inits initial display with custom colors.
// A color block is defined so that the rows in the browser are
// displayed with alernating colors.

   #include "Inkey.ch"
   #include "Dbedit.ch"

   #ifndef DE_INIT
      #define DE_INIT   -1
   #endif

   PROCEDURE Main
      LOCAL bColor := {|x| IIf( Recno() % 2 == 0, {1,2}, {3,4} ) }
      Local aCols  := { ;
        { "LASTNAME" , bColor }, ;
        { "FIRSTNAME", bColor }, ;
        { "CITY"     , bColor }, ;
        { "ZIP"      , bColor }  ;
      }

      USE Customer NEW

      DbEdit(,,,, aCols, "UserFunc", ,{ "Lastname","Firstname", "City", "Zip"} )

      CLOSE ALL

   RETURN

   FUNCTION UserFunc( nMode, nCol, oTBrowse )
      LOCAL GetList := {}
      LOCAL nReturn := DE_CONT

      DO CASE
      CASE nMode == DE_INIT
         oTBrowse:colorSpec := "n/bg,w+/r,w+/bg,w+/r,w+/gr"

      CASE nMode == DE_HITTOP
         Tone(1000)

      CASE nMode == DE_HITBOTTOM
         Tone(500)
```

```
      CASE LastKey() == K_ESC
         nReturn := DE_ABORT

      CASE LastKey() == K_ENTER
         SetCursor(1)
         @ Row(), Col() Get &(oTBrowse:getColumn(nCol):heading)
         READ
         SetCursor(0)
         CLEAR TYPEAHEAD
   ENDCASE
RETURN nReturn
```

DbEval()

Evaluates a code block in a work area.

Syntax

```
DbEval( <bBlock>          , ;
       [<bForCondition>]  , ;
       [<bWhileCondition>], ;
       [<nNextRecords>]   , ;
       [<nRecord>]        , ;
       [<lRest>]            ) --> NIL
```

Arguments

`<bBlock>`

The parameter is a code block which is executed for every record matching with the conditions specified with the following parameters. If no other parameters are passed, *<bBlock>* is evaluated for all records.

`<bForCondition>`

This is an optional code block which must return a logical value. *<bBlock>* is evaluated for all records where *<bForCondition>* yields .T. (true). *<bBlock>* is not executed for records where *<bForCondition>* yields .F. (false). This is equivalent to the FOR condition of database commands.

`<bWhileCondition>`

This is an optional code block which must return a logical value. *<bBlock>* is evaluated while *<bWhileCondition>* yields .T. (true). DbEval() returns immediately as soon as *<bWhileCondition>* yields .F. (false). This is equivalent to the WHILE condition of database commands.

`<nNextRecords>`

An optional numeric parameter restricting the number of records to process to *<nNextRecords>*, beginning with the current record. It is equivalent to the NEXT clause of database commands.

`<nRecord>`

Only a single record is processed when the record number *<nRecord>* is specified. It is equivalent to the RECORD clause of database commands.

`<lRest>`

An optional logical value that defaults to .F. (false). In this case, *<bBlock>* is evaluated for all records. Specifying .T. (true) causes DbEval() to start processing with the current record and continue until the end of file is reached. *<lRest>* is equivalent to the ALL (.F.) and REST (.T.) clauses of database commands.

Return

The return value is always NIL.

Description

The DbEval() function iterates the records in a work area and evaluates a code block for all records matching the scope and/or condition defined with parameters #2 to #5.

By default, DbEval() navigates the record pointer in the current work area. Use an aliased expression to process records in a different work area.

Info

See also: AEval(), Eval(), HEval()
Category: Database functions, Code block functions
Source: rdd\dbcmd.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example uses DbEval() to determine the number of deleted records
// in a database to calculate the space that could be freed with PACK.

   PROCEDURE Main
      LOCAL nCount := 0

      USE Customer NEW
      DbEval( {|| nCount++ }, ;
              {|| Deleted() } )

      ? "File size could be reduced by", 100*nCount/Lastrec(),"percent"

      CLOSE Customer
   RETURN
```

Dbf()

Retrieves the alias name of the current work area.

Syntax

```
Dbf() --> <cAliasName>
```

Return

The function returns the alias name of the current work area as a character string.

Description

The Dbf() function exists for compatibility reasons only. It is superseded by the Alias() function.

Info

See also:	Alias(), Used()
Category:	Database functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

DbFieldInfo()

Retrieves information about a database field

Syntax

```
DbFieldInfo( <nInfo>, <nFieldPos> ) --> xFieldInfo
```

Arguments

`<nInfo>`

The parameter is a numeric value specifying the type of information to query about a database field. #define constants listed in the table below must be used for *<nInfo>*.

`<nFieldPos>`

A numeric value specifying the ordinal position of the field to query. Valid values are in the range from 1 to FCount().

Return

DbFieldInfo() returns the queried information as supplied by the replaceable database driver (RDD).

Description

DbFieldInfo() allows for querying detailed field information from the replaceable database driver (RDD). Depending on database and RDD used, different information may be available. The type of information to retrieve is specified with #define constants available in the DBINFO.CH and DBSTRUCT.CH header files.

Constants for DbFieldInfo()

Constant	Return value
DBSTRUCT.CH	
DBS_DEC	Number of decimal places for the field.
DBS_LEN	Numeric length of the field.
DBS_NAME	Character string holding the name of the field.
DBS_TYPE	Character identifying the data type of the field.
DBINFO.CH	
DBS_BLOB_LEN	Number of bytes stored in a memo field (BLOB).
DBS_BLOB_OFFSET	Numeric file offset into memo file of the data in a memo field.
DBS_BLOB_POINTER	Numeric value for usage with e.g. BlobDirectGet(), BlobDirectImport().
DBS_BLOB_TYPE	Character indicating the data type of a memo field (BLOB).

By default, DbFieldInfo() operates in the current work area. Use an aliased expression to query field information in different work areas.

Info

See also: BlobDirectGet(), BlobDirectImport(), DbInfo(), DbOrderInfo(), DbRecordInfo(), DbStruct(), FieldDec(), FieldLen(), FieldName(), FieldType()
Category: Database functions, Field functions
Header: Dbinfo.ch, Dbstruct.ch
Source: rdd\dbcmd.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example retrieves structural information for a
// single database with a user defined function.

   #include "Dbstruct.ch"

   PROCEDURE Main

      USE Customer
      AEval( FieldStruct( 1 ), {|x| QOut( x ) } )
      USE

   RETURN

   FUNCTION FieldStruct( nFieldPos )
      LOCAL aFStruct[4]

      aFStruct[DBS_NAME] := DbFieldInfo( DBS_NAME, nFieldPos )
      aFStruct[DBS_TYPE] := DbFieldInfo( DBS_TYPE, nFieldPos )
      aFStruct[DBS_LEN ] := DbFieldInfo( DBS_LEN , nFieldPos )
      aFStruct[DBS_DEC ] := DbFieldInfo( DBS_DEC , nFieldPos )

   RETURN aFStruct
```

DbFileGet()

Copies data from a field into an external file.

Syntax

```
DbFileGet( <nFieldPos>, <cTargetFile>, <nMode> ) --> lSuccess
```

Arguments

<nFieldPos>

A numeric value specifying the ordinal position of the field to copy. Valid values are in the range from 1 to FCount().

<cTargetFile>

Character string holding the name of the external file to copy a field's data to. The file name may include drive and directory and must include an extension. The file is created if it does not exist. Otherwise, the function attempts to open the file exclusively. When this fails, NetErr() is set to .T. (true).

<nMode>

The parameter triggers if a new file is created or data is appended to a file. The following constants from DBINFO.CH must be used:

Constants for the append mode

Constant	Value	Description
FILEGET_APPEND	1	Appends data to the file.
FILEGET_OVERWRITE	0	Overwrites file with data.

Return

The return value is .T. (true) when the operation is successful, otherwise .F. (false) is returned.

Description

The DbFileGet() and DbFilePut() functions are mainly used to transfer data between memo fields holding large amounts of data (BLOB) and external files. Data written to an external file is then programmatically processed.

Info

See also: BlobDirectExport(), BlobExport(), DbFilePut()
Category: Database functions, Field functions
Header: Dbinfo.ch
Source: rdd\dbcmd.c
LIB: xhb.lib
DLL: xhbdll.dll

DbFilePut()

Copies the contents of an external file into a field.

Syntax

```
DbFilePut( <nFieldPos>, <cSourceFile> [,<Mode>] ) --> lSuccess
```

Arguments

<nFieldPos>

A numeric value specifying the ordinal position of the field to copy to. Valid values are in the range from 1 to FCount().

<cSourceFile>

A character string holding the name of the external file to copy into the field. The file name may include drive and directory and must include an extension. If the file does not exist, a runtime error is raised. The function attempts to open the file in shared mode. When this fails, NetErr() is set to .T. (true).

<nMode>

The parameter triggers how the contents of the file is stored in the field. The following constants from DBINFO.CH can be used:

Constants for the import mode

Constant	Value	Description
FILEPUT_COMPRESS	1	Data is compressed
FILEPUT_ENCRYPT	2	Data is encrypted

Return

The return value is .T. (true) when the operation is successful, otherwise .F. (false) is returned.

Description

The DbFilePut() and DbFileGet() functions are mainly used to transfer data between memo fields holding large amounts of data (BLOB) and external files. Data imported from an external file is then programmatically processed.

Info

See also: BlobDirectImport(), BlobImport(), DbFileGet()
Category: Database functions, Field functions
Header: Dbinfo.ch
Source: rdd\dbcmd.c
LIB: xhb.lib
DLL: xhbdll.dll

DbFilter()

Returns the filter expression of a work area-

Syntax

```
DbFilter() --> cFilter
```

Return

The function returns a character string containing the filter expression of a work area, or a null string ("") when no filter is set.

Description

The DbFilter() function can be used to query the filter condition set in a work area as a character string. This allows for temporarily disabling a filter and restore it later using the macro operator (&). Note, however, that this works only if a filter condition does not refer to lexical variables (GLOBAL, LOCAL, STATIC).

By default, DbFilter() operates in the current work area. Use an aliased expression to query filter expressions of different work areas.

Info

See also:	DbClearFilter(), DbRelation(), DbRSelect(), DbSetFilter(), SET FILTER
Category:	Database functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example outlines a programming technique for saving
// and restoring a filter condition in a work area

   PROCEDURE Main
      LOCAL cFilter

      USE Customer ALIAS CUst SHARED
      INDEX ON Upper(Lastname+Firstname) TO Cust01

      SET FILTER TO Cust->Lastname = "S"
      GO TOP
      ? Cust->Lastname                  // result: Shiller

      ? cFilter := DbFilter()           // result: Cust->Lastname = "S"

      DbClearFilter()

      GO TOP
      ? Cust->Lastname                  // result: Alberts

      SET FILTER TO &cFilter

      GO TOP
      ? Cust->Lastname                  // result: Shiller

      CLOSE Cust
   RETURN
```

DbfSize()

Returns the size of a database file in memory that is opened in a workarea.

Syntax

```
DbfSize() --> nFileSize
```

Return

The function returns the size of a database open in a work area as a numeric value.

Info

See also:	DiskFree(), Header(), LastRec(), RecSize()
Category:	CT:Database, Database functions
Source:	ct\dbftools.c
LIB:	xhb.lib
DLL:	xhbdll.dll

DbGoBottom()

Moves the record pointer to last record.

Syntax

```
DbGoBottom() --> NIL
```

Return

The return value is always NIL.

Description

The function moves the record pointer in the current or aliased work area to the last logical record. If no index or filter is set, this is the last physical record. Otherwise, it is the last record in logical order.

Info

See also: Bof(), DbGoto(), DbGoTop(), DbSeek(), DbSkip(), Eof(), GO, INDEX, SET FILTER, SKIP
Category: Database functions
Source: rdd\dbcmd.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates the effect of DbGobottom()

   PROCEDURE Main
      USE Customer

      ? Recno(), LastRec()             // result:   1   225

      DbGobottom()

      ? Recno(), LastRec()             // result: 225   225

      ? Eof()                          // result: .F.
      SKIP
      ? Eof()                          // result: .T.

      USE
   RETURN
```

DbGoto()

Positions the record pointer on a particular record.

Syntax

```
DbGoto( <xRecno> ) --> NIL
```

Arguments

<xRecno>

The parameter identifies a single record in a database. For DBF files, this is the numeric record number, as returned by Recno() for a particular record.

Return

The return value is always NIL.

Description

The DbGoto() function moves the record pointer to the record identified with *<xRecno>*. This position is independent of any index or filter currently active in the work area, i.e. *<xRecno>* does not change with the logical order of records but identifies records physically.

Issuing a DbGoto(Recno()) call in a network enviroment will refresh the database and index buffers. This is the same as a DbSkip(0) call. The parameter *<xRecno>* may be something other than a record number, depending on the RDD used. In some data formats, for example, the value of *<xRecno>* is a unique primary key, while in other formats, *<xRecno>* could be an array offset if the data set was an array.

If the record *<xRecno>* does not exist in the work area, the record pointer is positioned on the "ghost record" (Lastrec()+1), and both, BoF() and Eof(), are set to .T. (true).

Info

See also:	Bof(), DbGoBottom(), DbGoTop(), DbSeek(), DbSkip(), Eof(), GO, OrdKeyGoTo()
Category:	Database functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example physically moves the record pointer and outlines
// some important state variables of a work area.

   PROCEDURE Main
      USE Customer

      ? Recno(), LastRec()             // result:   1   225

      DbGoto(10)
      ? Recno(), LastRec()             // result:  10   225

      DbGoto(1000)
      ? Recno(), Bof(), Eof()          // result: 226   .T. .T.

      DbGoto(-1)
      ? Recno(), Bof(), Eof()          // result: 226   .T. .T.
```

```
      USE
   RETURN
```

DbGoTop()

Moves the record pointer to first record.

Syntax

```
DbGoTop() --> NIL
```

Return

The return value is always NIL.

Description

The function moves the record pointer in the current or aliased work area to the first logical record. If no index or filter is set, this is the first physical record. Otherwise, it is the first record in logical order.

Info

See also:	Bof(), DbGoBottom(), DbGoto(), DbSeek(), DbSkip(), Eof(), GO, INDEX, SET FILTER, SKIP
Category:	Database functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates the effect of DbGoTop()

   PROCEDURE Main
      USE Customer

      DbGobottom()
      ? Recno(), LastRec()             // result: 225   225

      DbGoTop()
      ? Recno(), LastRec()             // result:   1   225

      ? Bof()                          // result: .F.
      SKIP -1
      ? Bof()                          // result: .T.

      USE
   RETURN
```

DbInfo()

Queries and/or changes information about a database file open in a work area.

Syntax

```
DbInfo( <nDefine>, [<xNewSetting>] ) --> xCurrentSetting
```

Arguments

<nDefine>

This is a numeric parameter for which #define constants exist in the file DbInfo.ch. They identify the numerous settings that can be queried or changed in a work area and begin with the prefix DBI_ (see below).

<xNewSetting>

<xNewSetting> is an optional argument specifying a new value for the work area setting identified by *<nDefine>*. The data type for *<xNewSetting>* depends on the work area setting to change (see below). If the work area setting is READONLY, the parameter is ignored.

Return

The function returns the value of the specified work area setting which is set before the function is called.

Description

DbInfo() is a universal function managing numerous work area settings available in xHarbour. Many of these settings can be changed via commands or functions, which is the recommended way. If a work area is not used, DbInfo() generates a runtime error. Also, if an RDD does not support a setting, it may create a runtime error or simply ignore the function call upon its own discretion. In the latter case, the return value of DbInfo() is NIL.

The constants that can be used for *<nDefine>* are listed below:

DBI_ALIAS	--> cAliasName (READONLY) **See also:** function Alias(). The return value is a character string holding the symbolic alias name of the work area.
DBI_BLOB_DIRECT_EXPORT	{ *<nBlobID>*, *<cTargetFile>*, [*<nMode>*] } --> lSuccess **See also:** function BlobDirectExport(). DbInfo() expects as second parameter an array with three elements. They are identical with the arguments of function BlobDirectExport(). The return value is of logical data type and indicates a successful operation.
DBI_BLOB_DIRECT_GET	{ *<nBlobID>*, [*<nStart>*], [*<nCount>*] } --> xBlobData **See also:** function BlobDirectGet(). DbInfo() expects as second parameter an array with three elements. They are identical with the arguments of function BlobDirectGet(). The return value is the data of the retrieved binary large object (BLOB).
DBI_BLOB_DIRECT_IMPORT	{ *<nOldBlobID>*, *<cSourceFile>* } --> nNewBlobID **See also:** function BlobDirectImport(). DbInfo() expects as second parameter an array with two elements. They are identical with the arguments of function BlobDirectImport(). The return value is the numeric BLOB identifier of the new BLOB.
DBI_BLOB_DIRECT_LEN	*<nBlobID>* --> nBytes

	The second parameter must be the numeric identifier of a binary large object (BLOB) which can be queried using DbFieldInfo(DBS_BLOB_POINTER, *<nFieldPos>*). The return value is the number of bytes occupied by the BLOB in the BLOB file.
DBI_BLOB_DIRECT_PUT	{ *<nOldblobID>*, *<xBlobData>* } --> nNewBlobID **See also:** function BlobDirectPut(). DbInfo() expects as second parameter an array with two elements. They are identical with the arguments of function BlobDirectPut(). The return value is the numeric BLOB identifier of the new BLOB.
DBI_BLOB_DIRECT_TYPE	*<nBlobID>* --> cDataType The second parameter must be the numeric identifier of a binary large object (BLOB) which can be queried using DbFieldInfo(DBS_BLOB_POINTER, *<nFieldPos>*). The return value is a single letter indicating the data type stored in the BLOB. This letter has the same meaning as the return value of function Valtype().
DBI_BLOB_INTEGRITY	--> lSuccess This setting causes DbInfo() to test the integrity of internal tables referring to the data stored in a BLOB file. If the test fails, the return value is .F. (false) and the internal tables can be rebuild using Dbinfo(DBI_BLOB_RECOVER).
DBI_BLOB_OFFSET	*<nBlobID>* --> nFileOffset The second parameter must be the numeric identifier of a binary large object (BLOB) returned from function BlobDirectPut(). It can also be queried using DbFieldInfo(DBS_BLOB_POINTER, *<nFieldPos>*). The return value is the numeric file offset where the BLOB is stored in the BLOB file.
DBI_BLOB_RECOVER	--> lSuccess Rebuilds the internal tables referring to the data stored in a BLOB file. This should only be called when DbInfo(DBI_BLOB_INTEGRITY) returns .F. (false). DbInfo() can only recover the internal tables referring to the data, it cannot recover the data when a BLOB file becomes corrupted.
DBI_BLOB_ROOT_GET	--> xBlobData **See also:** function BlobRootGet(). The return value is the data stored in the root area of a BLOB file.
DBI_BLOB_ROOT_LOCK	--> lSuccess **See also:** function BlobRootLock(). The return value is .T. (true) when a write lock is obtained for the root area of a BLOB file in concurrent file access. The root area must be locked before DbInfo(DBI_BLOB_ROOT_PUT) is executed. The lock is later released using DbInfo(DBI_BLOB_ROOT_UNLOCK).
DBI_BLOB_ROOT_PUT	*<xBlobData>* --> lSuccess **See also:** function BlobRootPut(). DbInfo() expects as second parameter the data to be stored in the root area of a BLOB file. The return value is .T. (true) when data is successfully written.
DBI_BLOB_ROOT_UNLOCK	--> NIL

	See also: function BlobRootUnlock(). Releases a lock placed on the root area of a BLOB file with DbInfo(BLOB_ROOT_LOCK). The return value is NIL.
DBI_BOF	--> lIsBeginOfFile (READONLY) **See also:** function Bof(). The return value is the logical begin-of-file status of a work area.
DBI_CANPUTREC	--> lCanWriteData **See also:** function RddList(). The return value is .T. (true) when the RDD maintaining the file open in a work area supports writing data to the file. This is the case for all RDDs listed by RddList(RDT_FULL).
DBI_CHILDCOUNT	--> nChildCount **See also:** command SET RELATION. The return value is numeric, indicating the number of child work areas related to the current work area.
DBI_DBFILTER	--> cFilter **See also:** function DbFilter(). Returns the filter expression of the work area as a character string.
DBI_DB_VERSION	[<*nOne*>] --> cRddVersionInfo Returns version information of the RDD maintaining files in a work area. When 1 is passed for the optional second parameter, the returned character string contains extended version information.
DBI_EOF	--> lIsEndOfFile (READONLY) **See also:** function Eof(). The return value is the logical end-of-file status of a work area.
DBI_FCOUNT	--> nFieldCount (READONLY) **See also:** function FCount(). The return value is the numeric field count in a work area.
DBI_FILEHANDLE	--> nFileHandle (READONLY) **See also:** function FOpen(). The return value is the numeric low-level file handle of the database file open in a work area.
DBI_FOUND	--> lIsFound (READONLY) **See also:** function Found(). The return value is the logical found status of the last search operation in a work area.
DBI_FULLPATH	--> cFullFileName The return value is a character string holding the fully qualified file name of the database open in a work area if the file resides in the directory specified with SET DEFAULT. When the file is located in another directory, a character string holding the file name without directory information is returned.
DBI_GETHEADERSIZE	--> nBytes (READONLY) **See also:** function Header(). The return value is the number of bytes occupied by the file header of the database open in a work area.

DBI_GETLOCKARRAY	--> aLockedRecords (READONLY) **See also:** function DbRLockList(). The return value is an array holding the record numbers of currently locked records.
DBI_GETRECSIZE	--> nRecordSize (READONLY) **See also:** function RecSize(). The return value is the number of bytes required to store a single record in the database open in a work area.
DBI_ISDBF	--> lIsDbf (READONLY) **See also:** function RddList(). The return value is .T. (true) when the the file open in a work area is a full functional database. This is the case for all RDDs listed by RddList(RDT_FULL).
DBI_ISFLOCK	--> lIsFileLock (READONLY) **See also:** function FLock(). The return value is .T. (true) when a file lock is placed on a database open in SHARED mode in the current work area.
DBI_ISREADONLY	--> lIsReadOnly (READONLY) **See also:** command USE. The return value is .T. (true) when a database file is open in READONLY mode in the current work area.
DBI_LASTUPDATE	--> dLastUpdate (READONLY) **See also:** function LUpdate(). The return value is the date of the last update of the database open in the current work area.
DBI_LOCKCOUNT	--> nLockedRecords (READONLY) **See also:** function DbRlockList(). The return value is the number of currently locked records.
DBI_LOCKOFFSET	--> nLockOffset (READONLY) **See also:** command SET DBFLOCKSCHEME. The return value is a numeric value indicating the virtual lock offset used to place write locks on records in concurrent database access. The virtual lock offset must be identical for all applications accessing the same database files. The lock offset can be changed using DbInfo(DBI_LOCKSCHEME).
DBI_LOCKSCHEME	[<*nNewLockScheme*>] --> nOldLockScheme **See also:** command SET DBFLOCKSCHEME. The second parameter is optional and can be a numeric value between 0 and 5. It selects the locking scheme to use for aquiring write locks on records in concurrent file access. #define constants are available in DbInfo.ch for <*nNewLockScheme*>. They begin with the prefix DBFLOCK_.
DBI_MEMOBLOCKSIZE	[<*nNewBlockSize*>] --> nOldBlockSize **See also:** command SET MEMOBLOCK. The second parameter is optional and defines the block size for memo fields used by the RDD that opens database files in a work area. The return value is the current memo block size as a numeric value. Note that changing the memo block size is not supported by all RDDs.

DBI_MEMOEXT	[*<cNewExtension>*] --> *<cOldExtension>* **See also:** command SET MFILEEXT. The second parameter is optional and file extension for memo files used by the RDD that opens database files in a work area. The return value is the current memo file extension as a character string.
DBI_MEMOHANDLE	--> *<nMemoFileHandle>* (READONLY) **See also:** function FOpen(). The return value is the numeric low-level file handle of the memo file open in a work area.
DBI_MEMOTYPE	--> nMemoType (READONLY) **See also:** function RddInfo(). The return value is numeric and indicates the type of a memo file associated with a database open in a work area. #define constants are available in DbInfo.ch that identify a memo file type. They begin with the prefix DB_MEMO_.
DBI_MEMOVERSION	--> nMemoVersion (READONLY) **See also:** function RddInfo(). The return value is numeric and indicates the version of a memo file associated with a database open in a work area. #define constants are available in DbInfo.ch that identify a memo file version. They begin with the prefix DB_MEMOVER_.
DBI_PASSWORD	[*<cPassword>*] --> NIL This setting defines a password of up to eight characters in length which is used for data encryption in the database file. Note that only data in a DBF file is encrypted. Data stored in index and/or memo files are not encrypted with the password.
DBI_RDD_VERSION	[*<nvalue>*] --> cVersion Returns version information of the RDD maintaining files in a work area. When 1 is passed for the optional second parameter, the returned character string contains extended version information.
DBI_ROLLBACK	--> NIL **See also:** command REPLACE. Voids the last changes applied to a record with REPLACE or FieldPut(), which is like an Undo operation. Changes are discarded and the original values are assigned back to a record. This is possible until DbCommit() is executed, a write lock is released using DbUnlock(), or the record pointer is moved.
DBI_SCOPEDRELATION	*<nRelation>* --> lIsScoped (READONLY) **See also:** function DbSetRelation(). DbInfo() expects as second parameter a numeric value indicating the ordinal position in the list of relations to query the SCOPED attribute for. Relations are numbered in the sequence they are defined, beginning with 1. The return value is .T. (true) when the relation to the child work area is scoped, when the .T. was passed for the fourth parameter of DbSetRelation().
DBI_SHARED	--> lIsShared (READONLY) **See also:** command USE. The return value is .T. (true) when a database is open in SHARED mode with the USE command.

DBI_TABLEEXT	--> cFileExtension (READONLY) **See also:** function DbTableExt(). The return value is the file extension of the database file open in a work area as a character string.
DBI_TABLETYPE	--> nTableType (READONLY) **See also:** function RddInfo(). The return value is numeric, indicating the type of the database file open in a work area. #define constants are avaialble in DbInfo.ch to identify the database type. They begin with the prefix DB_DBF_.
DBI_VALIDBUFFER	--> lBufferIsValid (READONLY) The return value is .T. (true) when the internal memory buffer holding data of a record on disk is valid. A valid buffer is unchanged, i.e. it contains the same data as stored in the database file.

Info

See also: Alias(), DbFieldInfo(), DbOrderInfo(), DbRecordInfo(), DbUseArea(), RddInfo(), Select()
Category: Database functions, Info functions
Header: Dbinfo.ch
Source: rdd\dbcmd.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates some DbInfo() settings and how
// to undo changes applied to a record.

   #include "DbInfo.ch"

   PROCEDURE Main
      USE Customer ALIAS Cust SHARED

      ? DbInfo( DBI_SHARED )                 // result: .T.
      ? DbInfo( DBI_ISREADONLY )             // result: .F.

      ? FIELD->LastName                      // result: Miller
      ? DbInfo( DBI_VALIDBUFFER )            // result: .T.

      RLock()
      ? ValToPrg( DbInfo(DBI_GETLOCKARRAY) )  // result: { 1 }

      REPLACE FIELD->LastName WITH "JONES"

      ? FIELD->LastName                      // result: JONES
      ? DbInfo( DBI_ROLLBACK )               // result: NIL
      ? FIELD->LastName                      // result: Miller

      DbUnlock()

      USE
   RETURN
```

DbJoin()

Merges records of two work areas into a new database.

Syntax

```
DbJoin( <cAlias>   , ;
        <cDatabase>, ;
       [<aFields>] , ;
       [<bFor>]      ) --> lSuccess
```

Arguments

<cAlias>

This a character string holding the alias name of the second work area to merge records of the current work area with.

<cDatabase>

This a character string holding name of the database file to create. It can include path and file extension. When no path is given, the file is created in the current directory. The default file extension is DBF.

<aFields>

A one dimensional array holding in its elements the field names of the fields to include in the resulting database. When this parameter is omitted, all fields of the primary and secondary work area are included. Use aliased field names to specify fields from the secondary work area.

<bFor>

An optional code block returning a logical value can be specified. It is evaluated for all records during the operation. Those records where *<bFor>* yields .T. (true) are included in the resulting database. If omitted, all records are

Return

The function returns .T. (true) on success, and .F. (false) on failure.

Description

DbJoin() is the functional equivalent of the JOIN command. Refer to JOIN for a detailed description of joining two databases into a new database.

Info

See also:	JOIN
Category:	Database functions, xHarbour extensions
Source:	rdd\dbjoin.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Dblist()

Displays records of a work area to the console, printer or file.

Syntax

```
DbList( [<lOff>]    , ;
        [<aBlocks>] , ;
        [<lAll>]    , ;
        [<bFor>]    , ;
        [<bWhile>]  , ;
        [<nNext>]   , ;
        [<nRecord>] , ;
        [<lRest>]   , ;
        [<lToPrint>], ;
        [<cOutFile>]  ) --> lSuccess
```

Arguments

`<lOff>`

This parameter defaults to .F. (false) so that record numbers are displayed in the first column of the data output. Passing .T. (true) suppresses the output of record numbers

`<aBlocks>`

An optional one dimensional array holding code blocks can be specified. All code blocks are evaluated for each record. The return values of the code blocks are output. If *<aBlocks>* is not given, the values of all fields of records is displayed

`<lAll>`

This parameter defaults to .T. (true) so that all records are processed.

`<bFor>`

This is an optional code block which must return a logical value. The code blocks in *<aBlock>* are evaluated for all records where *<bFor>* yields .T. (true). *<aBlocks>* is not executed for records where *<bFor>* yields .F. (false).

`<bWhile>`

This is an optional code block which must return a logical value. *<aBlocks>* is evaluated while *<bWhile>* yields .T. (true). DbList() returns immediately as soon as *<bWhile>* yields .F. (false).

`<nNext>`

An optional numeric parameter restricting the number of records to process to *<nNext>*, beginning with the current record.

`<nRecord>`

Only a single record is processed when the record number *<nRecord>* is specified.

`<lRest>`

An optional logical value that defaults to .F. (false). In this case, *<aBlocks>* is evaluated for all records. Specifying .T. (true) causes DbList() to start output with the current record and continue until the end of file is reached.

`<lToPrint>`

This parameter defaults to .F. (false). When set to .T. (true), output is directed to the printer.

`<cOutFile>`

The output can be directed to a file with the name *<cOutFile>*. Note that *<lToPrint>* and *<cOutFile>* are mutually exclusive.

Return

The function returns .T. (true) on success, and .F. (false) on failure.

Description

DbList() is the functional equivalent of the LIST command. Refer to LIST for a detailed description of displaying records to the console, printer or file.

Info

See also:	DbEval(), LIST, QOut()|QQOut()
Category:	Database functions, xHarbour extensions
Source:	rdd\dblist.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

DbOrderInfo()

Queries and/or changes information about indexes open in a work area.

Syntax

```
DbOrderInfo( <nDefine>                , ;
            [<cIndexFile>]            , ;
            [<nOrder>|<cIndexName>], ;
            [<xNewSetting>]           ) --> xCurrentSetting
```

Arguments

<nDefine>

This is a numeric parameter for which #define constants exist in the file DbInfo.ch. They identify the numerous settings that can be queried or changed for indexes and begin with the prefix DBOI_ (see below).

<cIndexFile>

<cIndexFile> is a character string with the name of the file that stores the index. It is only required when multiple index files are open that contain indexes having the same *<cIndexName>*.

<nOrder>

A numeric value specifying the ordinal position of the index open in a work area to query information for. Indexes are numbered in the sequence of opening, beginning with 1. The value zero identifies the controlling index.

<cIndexName>

Alternatively, a character string holding the symbolic name of the index to query can be passed. It is the string passed to the TAG option of the INDEX command and analogous to the alias name of a work area. Support for *<cIndexName>* depends on the RDD used to open the index. Usually, RDDs that are able to maintain multiple indexes in one index file support symbolic index names, such as DBFCDX, for example.

<xNewSetting>

<xNewSetting> is an optional argument specifying a new value for the index setting identified by *<nDefine>*. The data type for *<xNewSetting>* depends on the index setting to change (see below). If the index setting is READONLY, the parameter is ignored.

Return

The function returns the value of the specified index setting which is set before the function is called.

Description

DbOrderInfo() is a universal function managing numerous index settings available in xHarbour. Note, however, that not all settings are supported by all RDDs. If a setting is not supported by an RDD, a call to DbOrderInfo() may be ignored or may rise a runtime error.

When DbOrderInfo() is called in an unused work area, a runtime error is raised.

The constants that can be used for *<nDefine>* are listed below:

DBOI_BAGCOUNT	--> nOpenIndexFiles (READONLY) **See also:** command SET INDEX. This setting is supported by the DBFNTX RDD and returns the number of NTX index files open in a work area. Note that if an NTX index file is created with multiple TAG names, it may contain multiple indexes.

DBOI_BAGEXT	--> cFileExtension (READONLY) **See also:** function OrdBagExt(). The return value is the index file extension used by the RDD that opened a database in a work area as a character string.
DBOI_BAGNAME	--> cIndexFilename (READONLY) **See also:** function OrdBagName(). The return value is the name of the index file that contains the specified index as a character string.
DBOI_BAGNUMBER	--> nIndexFilePosition (READONLY) **See also:** command SET INDEX. This setting is supported by the DBFNTX RDD and returns the ordinal position of a single NTX index file in the list of open index files. Index files are numbered in the sequence they are opened beginning with one. DBOI_BAGNUMBER may yield a different result from DBOI_NUMBER when an NTX index file is created with multiple TAG names. In this case, the number of open NTX files may be different from the number of indexes available in a work area.
DBOI_BAGORDER	--> nFirstIndex (READONLY) The return value is the ordinal position of the firs index within a multi-key index file.
DBOI_CONDITION	--> cForCondition (READONLY) **See also:** function OrdFor(). The return value is a character string holding the FOR condition of the specified index. If the index is created without FOR condition, an empty string ("") is returned.
DBOI_CUSTOM	[<*lMakeCustomIndex*>] --> lIsCustomIndex **See also:** command INDEX. This setting is equivalent with the CUSTOM option of the INDEX command. When it is set to .T. (true) before an index is created, a custom index will be created with function OrdCreate(). If DBOI_CUSTOM is queried for an existing index file, its custom flag is returned as a logical value. This flag cannot be changed after index creation so that <*lMakeCustomIndex*> is ignored for existing indexes.
DBOI_EVALSTEP	--> nEvalStep (READONLY) **See also:** command INDEX. The return value is numeric and represents the value for the EVERY option when an index is created. The setting is supported by the DBFNTX and DBFCDX RDDs.
DBOI_EXPRESSION	--> cIndexKey (READONLY) **See also:** function OrdKey(). The return value is the key expression of the specified index as a character string.
DBOI_FILEHANDLE	--> nFileHandle (READONLY) **See also:** function FOpen(). The return value is the numeric low-level file handle of the index file open in a work area.
DBOI_FINDREC	<*nRecno*> --> lFound **See also:** function OrdFindRec(). This setting is equivalent with function call OrdFindRec(nRecno, .F.).

DBOI_FINDRECCONT	<*nRecno*> --> lFound **See also:** function OrdFindRec(). This setting is equivalent with function call OrdFindRec(nRecno, .T.).
DBOI_FULLPATH	--> cFullFileName (READONLY) The return value is a character string holding the name of an index file, including file extension, that contains the specified index.
DBOI_INDEXEXT	--> cIndexFileExtension (READONLY) DBOI_INDEXEXT is a synonym for DBOI_ORDBAGEXT and returns the file extension for an index file.
DBOI_INDEXNAME	--> cIndexFileName (READONLY) DBOI_INDEXNAME is a synonym for DBOI_ORDBAGNAME and returns the file name for an index file.
DBOI_ISCOND	--> lIsForCondition (READONLY) **See also:** command INDEX. The return value is .T. (true) when the specified index is created with a FOR condition. Otherwise .F. (false) is returned.
DBOI_ISDESC	[<*lNewDescending*>] --> lOldDescending **See also:** command INDEX. The return value is .T. (true) when the specified index is created with the DESCENDING option. If an RDD supports to change ascending and descending indexes dynamically "on the fly", this setting works exactly the same as function OrdDescend().
DBOI_ISMULTITAG	--> lIsMultiTag (READONLY) This setting is supported by the DBFNTX RDD. The return value is .T. (true) when an NTX file can contain multiple indexes, otherwise .F. (false) is returned. Multiple indexes can be created when indexes with different TAG names are created for the same index file.
DBOI_ISREADONLY	--> lIsReadOnly (READONLY) **See also:** command USE. This setting is supported by the DBFNTX RDD. The return value is .T. (true) when the database file, that is associated with the index, is open in READONLY mode in the current work area. The READONLY status is valid for both, datbase ans index files.
DBOI_ISREINDEX	--> lIsReindexing (READONLY) **See also:** function OrdListRebuild(). The return value is .T. (true) while reindexing is in progress, otherwise .F. (false) is returned.
DBOI_KEYADD	[<*xNewKeyValue*>] --> lKeyIsAdded **See also:** function OrdKeyAdd(). This setting can only be used with custom built indexes, i.e. when an index is created with the CUSTOM option of the INDEX command. If the index is not a custom index, a runtime error is raised. The return value is .T. (true), when the new key value <*xNewKeyValue*> is successfully added to a custom index. Otherwise, .F. (false) is returned. If <*xNewKeyValue*> is omitted, it is created from the index key for the current record.

DBOI_KEYCOUNT	--> nKeyCount (READONLY) **See also:** function OrdKeyCount(). This setting returns the number of index keys included in the specified index as a numeric value.
DBOI_KEYDEC	--> nDecimals (READONLY) This setting can only be queried for DBFNTX. It returns the number of decimal places in a numeric index.
DBOI_KEYDELETE	[<*xDelKeyValue*>] --> lKeyIsDeleted **See also:** function OrdKeyDel(). This setting can only be used with custom built indexes, i.e. when an index is created with the CUSTOM option of the INDEX command. If the index is not a custom index, a runtime error is raised. The return value is .T. (true), when the specified key value <*xDelKeyValue*> is successfully removed from a custom index. Otherwise, .F. (false) is returned. If <*xDelKeyValue*> is omitted, the index key of the current record is removed from the index.
DBOI_KEYGOTO	<*nLogicalRecord*> --> lSuccess **See also:** function OrdKeyGoTo(). This setting moves the record pointer to the logical record position specified with <*nLogicalRecord*>. The return value is .T. (true) when the record pointer is successfully positioned, otherwise .F. (false). Note that if a filter condition is active in a work area, all records matching the filter are excluded.
DBOI_KEYGOTORAW	<*nLogicalRecord*> --> lSuccess **See also:** function OrdKeyGoto(). This setting differs from DBOI_KEYGOTO only by ignoring any filter condition that may be set in a work area. DBOI_KEYGOTORAW includes records that match a filter condition.
DBOI_KEYNO	--> nOrdKeyNo (READONLY) **See also:** function OrdKeyNo(). This setting returns a numeric value representing the logical record number of the current record. If a filter is active in the work area, all records matching the filter condition are excluded.
DBOI_KEYNORAW	--> nOrdKeyNo (READONLY) **See also:** function OrdKeyNo(). This setting differs from DBOI_KEYNO only by ignoring any filter condition that may be set in a work area. DBOI_KEYNORAW includes records that match a filter condition.
DBOI_KEYSINCLUDED	--> nAddedKeys (READONLY) **See also:** command INDEX. This setting is only relevant during index creation. It returns the number of index keys already added to an index. The setting is used to display indexing progress by calling a user defined routine via the EVAL option of the index command.
DBOI_KEYSIZE	--> nKeySize (READONLY) **See also:** function OrdKey(). This setting returns the number of bytes an index value of a single record occupies in the index file.
DBOI_KEYTYPE	--> cKeyType (READONLY)

	See also: function OrdKey(). This setting returns a single letter representing the data type of an index expression. The returned letter is equivalent with the return value of Valtype().
DBOI_KEYVAL	--> xIndexVal (READONLY) **See also:** function OrdKeyVal(). The setting returns the index value of the current record.
DBOI_LOCKOFFSET	--> nLockOffset (READONLY) **See also:** command SET DBFLOCKSCHEME. This setting returns the locking offset used by the selected locking scheme as a numeric value.
DBOI_NAME	--> cIndexName (READONLY) **See also:** function OrdName(). This setting returns the symbolic name of an index as a character string. The name is specified with the TAG option of the INDEX command when the index is created.
DBOI_NUMBER	--> nOrder (READONLY) **See also:** function OrdNumber(). This setting returns the ordinal position of an index in the list of open indexes based on its symbolic name. The name is specified with the TAG option of the INDEX command when the index is created.
DBOI_ORDERCOUNT	--> nIndexCount (READONLY) **See also:** function OrdCount(). This setting returns the number of indexes open in a work area as a numeric value.
DBOI_POSITION	[<*nNewIndexKeyNo*>] --> nIndexKeyNo **See also:** function OrdKeyGoto() and OrdKeyNo(). This setting queries and/or changes the logical position of the record pointer. If <*nNewIndexKeyNo*> is specified, it must be a numeric value in the range between 1 and OrdKeyCount().
DBOI_RELKEYPOS	[<*nNewRelativePos*>] --> nRelativePos **See also:** function OrdKeyRelPos(). This setting queries and/or changes the relative position of the record pointer. If <*nNewRelativePos*> is specified, it must be a numeric value in the range between 0 and 1.
DBOI_SCOPEBOTTOM	[<*xNewBottomScope*>] --> xBottomScope **See also:** function OrdScope(). This setting queries and/or changes the bottom scope value for navigating the record pointer. If <*xNewBottomScope*> is specified, it must be of the same data type as the value of the index expression.
DBOI_SCOPEBOTTOMCLEAR	--> xBottomScope **See also:** function OrdScope(). This setting releases the bottom scope value for navigating the record pointer. The return value is the previously set bottom scope value.
DBOI_SCOPECLEAR	--> NIL **See also:** command SET SCOPE. This setting releases both, the top and bottom scope value for navigating the record pointer. It is

	equivalent with SET SCOPE TO and no arguments apecified. The return value is always NIL.
DBOI_SCOPESET	[<*xBothScopes*>] --> NIL **See also:** function OrdScope(). This setting allows for defining the same value for both, the top and bottom scope, with one function call. If <*xBothScopes*> is specified, its data type must be the same as the value of the index expression. The return value is always NIL.
DBOI_SCOPETOP	[<*xNewTopScope*>] --> xTopScope **See also:** function OrdScope(). This setting queries and/or changes the top scope value for navigating the record pointer. If <*xNewTopScope*> is specified, it must be of the same data type as the value of the index expression.
DBOI_SCOPETOPCLEAR	--> xTopScope **See also:** function OrdScope(). This setting releases the top scope value for navigating the record pointer. The return value is the previously set top scope value.
DBOI_SKIPUNIQUE	[<*nDirection*>] --> lSuccess This setting is equivalent with function OrdSkipUnique().
DBOI_SKIPWILD	<*cWildCardString*> --> lFound **See also:** function OrdWildSeek(). This setting is equivalent with function call OrdWildSeek(cWildCardString, .F., .F.).
DBOI_SKIPWILDBACK	<*cWildCardString*> --> lFound **See also:** function OrdWildSeek(). This setting is equivalent with function call OrdWildSeek(cWildCardString, .F., .T.).
DBOI_UNIQUE	--> lIsUnique (READONLY) **See also:** function OrdIsUnique(). The return value is .T. (true) when the specified index is created with the UNIQUE option of the INDEX command, otherwise .F. (false) is returned.

Info

See also:	DbInfo(), DbRecordInfo(), OrdBagExt(), OrdKey(), OrdName(), RddInfo()
Category:	Database functions, Info functions
Header:	Dbinfo.ch
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

DbRecall()

Recalls a record previousy marked for deletion.

Syntax

```
DbRecall() --> NIL
```

Return

The return value is always NIL.

Description

The DbDelete() function removes the deletion flag from a record marked as deleted. The deletion flag is set with function DbDelete().

Not that if SET DELETED is set to ON, functions and commands for relative database navigation ignore all records having the deleted flag set. This way, they become unvisible.

In a networking situation, this function requires the current record be locked prior to calling the DbRecall() function.

Info

See also:	DbDelete(), DELETE, Deleted(), NetRecall(), RECALL
Category:	Database functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates how to set and detect the deletion flag

   PROCEDURE Main
      USE Customer EXCLUSIVE
      DbGoto(10)

      DbDelete()
      ? Deleted()                     // result: .T.

      DbRecall()
      ? Deleted()                     // result: .F.

      USE
   RETURN
```

DbRecordInfo()

Queries and/or changes information about a record of an open database file.

Syntax

```
DbRecordInfo( <nDefine>, [<nRecord>], [<xNewSetting>] ) --> xOldSetting
```

Arguments

`<nDefine>`

This is a numeric parameter for which #define constants exist in the file DbInfo.ch. They identify the individual data that can be queried for a single record.

`<nRecord>`

The parameter identifies the record in a database to query data for. It defaults to the return value of Recno(), the current record.

`<xNewSetting>`

This parameter is reserved for RDDs that allow the record information to be changed.

Return

DbRecordInfo() returns the queried record information as supplied by the replaceable database driver (RDD).

Description

DbRecordInfo() queries detailed record information from the replaceable database driver (RDD). Depending on database and RDD used, different information may be available. The type of information to retrieve is specified with #define constants available in the DBINFO.CH header file.

Constants for DbRecordInfo()

Constant	Return value
DBRI_DELETED	The Deleted() flag of a record as a logical value
DBRI_ENCRYPTED	The encrypted flag of a record as a logical value
DBRI_LOCKED	The write lock status of a record as a logical value
DBRI_RAWDATA	The raw data of a record as a character string
DBRI_RAWMEMOS	The raw memo field data of a record as a character string
DBRI_RAWRECORD	The raw data of a record except memo fields as a character string
DBRI_RECNO	The numeric relative record position, see OrdKeyNo()
DBRI_RECSIZE	The numeric size of a record, see RecSize()
DBRI_UPDATED	The updated flag of a record as a logical value

By default, DbRecordInfo() operates in the current work area. Use an aliased expression to query field information in different work areas.

Info

See also:	Alias(), DbInfo(), DbFieldInfo(), DbOrderInfo(), DbUseArea(), Deleted(), OrdKeyNo(), RecSize(), RLock(), Select()
Category:	Database functions, Info functions
Header:	Dbinfo.ch
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

DbReindex()

Rebuilds indexes open in a work area.

Syntax

```
DbReindex() --> NIL
```

Return

The return value is always NIL.

Description

The function DbReindex() exists for compatibility reasons. Is is superseded by function OrdListRebuild().

DbReindex() rebuilds all open indexes in the current work area. Use an aliased expression to rebuild indexes in other work areas.

Info

See also:	DbClearIndex(), DbCreateIndex(), DbSetIndex(), DbSetOrder(), INDEX, OrdCreate(), OrdListRebuild(), REINDEX
Category:	Database functions, Index functions
Source:	rdd\rddord.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// refer to the example of OrdListRebuild()
```

DbRelation()

Returns the linking expression of a specified relation.

Syntax

```
DbRelation( <nRelation> ) --> cLinkExpression
```

Arguments

<nRelation>

The parameter is a numeric value indicating the ordinal position in the list of relations to retrieve the relation expression from. Relations are numbered in the sequence they are defined, beginning with 1.

Return

The function returns a character string containing the relation expression specified with <*nRelation*>, or a null string ("") when no relation exists for <*nRelation*>.

Description

The DbRelation() function can be used to query the relation expression for a given relation. In conjunction with DbRSelect(), all data required to save and restore relation data using the macro operator (&) can be obtained. Note, however, that this works only if a relation does not refer to lexical variables (GLOBAL, LOCAL, STATIC).

DbRelation() retrieves the expression of the TO clause of the SET RELATION command, while DbRSelect() retrieves the data of the INTO clause. Refer to the SET RELATION command for more information on relations.

By default, DbRelation() operates in the current work area. Use an aliased expression to query relation expressions of different work areas.

Info

See also:	DbFilter(), DbRSelect(), DbSetRelation(), OrdSetRelation(), SET RELATION
Category:	Database functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example opens three databases and defines two relations.
// The result of various DbRelation() calls is shown.

   PROCEDURE Main
      USE Customer ALIAS Cust NEW EXCLUSIVE
      INDEX ON CustNo TO CustNo

      USE Parts ALIAS Part NEW EXCLUSIVE
      INDEX ON PartNo TO PartNo

      USE Order ALIAS Ord NEW

      SET RELATION TO CustNo INTO Cust, ;
                   TO PartNo INTO Part

      ? Alias()                      // result: ORD
      ? DbRelation(1)                // result: CustNo
```

```
   ? DbRelation(2)                // result: PartNo

   SELECT Part
   ? Alias()                      // result: PART
   ? DbRelation(1)                // result: null string ("")

   ? Ord->( DbRelation(1) )       // result: CustNo
   ? Ord->( DbRelation(2) )       // result: PartNo

   CLOSE ALL
RETURN
```

DbRLock()

Locks a record for write access.

Syntax

```
DbRLock( [<xRecno>] ) --> lSuccess
```

Arguments

<xRecno>

The parameter identifies a single record in a database. For DBF files, this is the numeric record number, as returned by Recno() for a particular record. It defaults to the current record.

Return

The return value is .T. (true) when the record *<xRecno>* is successfully locked, otherwise .F. (false) is returned.

Description

The DbRLock() function attempts to lock the record identified by *<xRecno>* in the current work area, and return .T. (true) if the record is locked.

If *<xRecno>* is not specified, all active record locks are removed and the current record is locked. When a particular record is specified with *<xRecno>*, the function places a lock on this record and adds it to the list of active record locks. This list can be retrieved with function DbRLockList().

Use DbRUnLock() to release locks from individual records.

Info

See also:	DbRLockList(), DbRUnlock(), DbUnlock(), DbUnlockAll(), FLock(), NetRecLock(), RLock(), UNLOCK
Category:	Database functions, Network functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example attempts to lock multiple records belonging to a
// transaction and unlocks all if a single lock is not successful

   PROCEDURE Main
      LOCAL cTransActID := "01234"
      LOCAL nCount      := 0

      USE Accounts INDEX Transact

      SEEK (cTransActID)

      DO WHILE FIELD->TRANSACTID == cTransActID
         IF DbRLock( Recno() )
            nCount ++              // successful lock
            SKIP                   // proceed to next record
         ELSE
            nCount := -1           // lock failed on this record
            DbRUnlock()            // release all previous locks
         ENDIF
      ENDDO
```

```
   IF nCount > 0
      <transaction code with locked records goes here>
   ENDIF

RETURN
```

DbRLockList()

Returns a list of locked records.

Syntax

```
DbRLockList() --> aLockedRecords
```

Return

The function returns a one-dimensional array holding the record identifiers of locked records, or an empty array if no record is locked.

Description

The function is used to determine which and how many records are currently locked for shared access. It operates in the current work area unless used in an aliased expression.

Info

See also:	DbRLock(), DbRUnlock(), DbUnlock(), DbUnlockAll(), FLock(), RLock(), UNLOCK
Category:	Database functions, Network functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// Theexample deonstrates the return value of DbRLockList()

   PROCEDURE Main()
      LOCAL aList, n

      USE Customer NEW
      ? LastRec()                // result: 225

      DbGoto(10)
      ? DbRLock( Recno() )       // result: .T.

      DbGoto( 180 )
      ? DbRLock( Recno() )       // result: .T.

      aList := DbRLockList()
      ? Len(aList)               // result: 2

      FOR n:=1 TO Len( aList )
         ?? aList[n]             // result: 10  180
      NEXT

      USE
   RETURN
```

DbRSelect()

Returns the child work area number of a relation.

Syntax

```
DbRSelect( <nRelation> ) --> nWorkArea
```

Arguments

<nRelation>

The parameter is a numeric value indicating the ordinal position in the list of relations to retrieve the child work area number. Relations are numbered in the sequence they are defined, beginning with 1.

Return

The function returns as a numeric value the work area number of the relation specified with *<nRelation>*. The return value is zero if no relation is defined.

Description

The function DbRSelect() determines the work area number of a child work area, the current work area is related to. It determines it from the ordinal position of a relation which is numbered according to the sequence a relation is defined with the SET RELATION command.

DbRSelect() is used in conjunction with DbRelation() which retrieves the relation expression of a relation. Pass the return value of DbRSelect() to Alias() to find out the alias name of the child work area.

Info

See also:	DbFilter(), DbRelation(), DbSetRelation(), OrdSetRelation(), SET RELATION
Category:	Database functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example opens three databases and defines two relations.
// The result of various DbRSelect() calls is shown.

   PROCEDURE Main
      USE Order ALIAS Ord NEW

      USE Parts ALIAS Part NEW EXCLUSIVE
      INDEX ON PartNo TO PartNo

      USE Customer ALIAS Cust NEW EXCLUSIVE
      INDEX ON CustNo TO CustNo

      SELECT Ord
      SET RELATION TO CustNo INTO Cust, ;
                   TO PartNo INTO Part

      ? Alias()                      // result: ORD
      ? DbRSelect(1)                 // result: 3
      ? DbRSelect(2)                 // result: 2
      ? Alias(DbRSelect(1))          // result: CUST
      ? Alias(DbRSelect(2))          // result: PART
```

```
    SELECT Part
    ? Alias()                      // result: PART
    ? DbRSelect(1)                 // result: 0

    CLOSE ALL
 RETURN
```

DbRUnlock()

Unlocks a record based on its indentifier.

Syntax

```
DbRUnlock( [<xRecno>] ) --> NIL
```

Arguments

<xRecno>

The parameter identifies a single record in a database. For DBF files, this is the numeric record number, as returned by Recno() for a particular record. If not specified, all active record locks are released.

Return

The return value is always NIL.

Description

The DbRUnlock() function releases a record lock for an individual record specified with *<xRecno>*. If no record identifier is passed to the function, DbRUnlock() bahaves like DbUnlock() since all active record locks are released.

Locking and unlocking one or more records for shared write access is the task of DbRlock() and DbRUnlock(), while RLock() works with the current record and DbUnlock() releases all locks.

Info

See also:	DbRLock(), DbRLockList(), DbUnlock(), RLock(), UNLOCK
Category:	Database functions, Network functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows a user defined function that accepts as
// parameter an array of record identifiers. These records are
// unlocked, unless the parameter is no array. In this case, all
// record locks are released.

   FUNCTION UnlockMultiple( xRecords )

      IF Valtype( xRecords ) == "A"
                                 // release individual locka
         AEval( xRecords, {|nRecno| DbRUnlock( nRecno ) } )
      ELSE
         DbRUnlock()             // release all locks
      ENDIF

   RETURN DbRLockList()          // return remaining locks
```

DbSeek()

Searches a value in the controlling index.

Syntax

```
DbSeek( <xValue>, [<lSoftSeek>], [<lFindLast>] ) --> lFound
```

Arguments

<xValue>

The value to search for. Its data type must match the data type of the index expression of the controlling index.

<lSoftSeek>

This optional value defaults to .F. (false) causing the DbSeek() function to position the record pointer at Eof() if *<xValue>* is not found in the index. When .T. (true) is passed for *<lSoftSeek>* and *<xValue>* is not found in the index, the record pointer is positioned on the record with the next higher index value.

<lFindLast>

<lFindLast> is only relevant when the database contains multiple records having identical index values. It defaults to .F. (false) causing the DbSeek() function to position the record pointer on the first record found. .T. (true) instructs DbSeek() to position the record pointer on the last of multiple records having the same index value.

Return

DbSeek() returns .T. (true) if *<xValue>* is found, otherwise .F. (false).

Description

The DbSeek() function is used to perform fast searches in databases. To accomplish this, the database must be indexed, since DbSeek() searches the value *<xValue>* in the controlling index, rather than in the database. It operates in the current work area, unless it is used in an aliased expression.

When DbSeek() finds *<xValue>* in the controlling index, it returns .T. (true) and positions the record pointer to the corresponding record. The parameter *<lFindLast>* optionally specifies which record to find if there are multiple records having the same index value. By default, the first record is found. If *<lFindLast>* is .T. (true), DbSeek() positions the record pointer on the last of the records having identical index values.

After a successful search, the function Found() returns .T. (true) until the record pointer is moved again. In addition, both functions, BoF() and EoF() return .F. (false).

If the searched value is not found, DbSeek() positions the record pointer on the "ghost record" (Lastrec()+1) by default, and the function Found() returns .F. (false), while Eof() returns .T. (true). This behavior, however, depends on the SET SOFTSEEK setting, or the parameter *<lSoftSeek>*, respectively. If SOFTSEEK is set to ON, or *<lSoftSeek>* is .T. (true), the record pointer is moved to the record yielding the next higher index value. If such a record exists, both functions, Found() and Eof() return .F. (false). If no record with a higher index value than *<xValue>* exists, the record pointer ends up at the "ghost record" (see above).

Info

See also:	BoF(), DbGoBottom(), DbGoTop(), DbSkip(), Eof(), Found(), LOCATE, OrdFindRec(), OrdKeyGoto(), OrdWildSeek(), SEEK
Category:	Database functions, Index functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example outlines various possibilities of searching a value
// with DbSeek() and the resulting states of Found() and Eof().

   PROCEDURE Main
      USE Customer NEW
      INDEX ON Upper(Lastname+Firstname) TO Cust01

      // standard search
      ? DbSeek( "GRAY" )                  // result: .T.
      ? Found(), Eof()                    // result: .T.  .F.
      ? Lastname, Firstname               // result: Gray      Eileen

      // find last matching record
      ? DbSeek( "GRAY", .F., .T. )        // result: .T.
      ? Found(), Eof()                    // result: .T.  .F.
      ? Lastname, Firstname               // result: Gray      Robert

      // no matching record
      ? DbSeek( "Gray" )                  // result: .F.
      ? Found(), Eof()                    // result: .F.  .T.
      ? Lastname, Firstname               // result: (empty string)

      // no matching record, with softseek
      ? DbSeek( "GREGOR", .T. )           // result: .F.
      ? Found(), Eof()                    // result: .F.  .F.
      ? Lastname, Firstname               // result: Hellstrom  Eric

      USE
   RETURN
```

DbSelectArea()

Selects the current work area.

Syntax

```
DbSelectArea( <cAlias> | <nWorkArea> ) --> NIL
```

Arguments

`<cAlias>`

A character string holding the alias name of the work area to select as current.

`<nWorkArea>`

The numeric ordinal position of the work area to select. Work areas are numbered from 1 to 65535. The value 0 has a special meaning: it selects the next unused work area, irrespective of its ordinal number.

Return

The return value is always NIL.

Description

DbSelectArea() selects the current work area. The scope of all database commands and all unaliased database functions is the current work area. A work area can be thought of as the place where databases and accompanying files are open and become accessible. One work area can hold one open database. To operate with multiple open databases at a time requires an application to switch between work areas using DbSelectArea().

An alternative is given by aliased expressions that are prefixed with the alias name or work area number of the work area to select temporarily before executing a database function. Note that aliased expressions are only possible with database functions. Database commands operate always in the current work area.

Info

See also: DbUseArea(), SELECT, Select(), USE, Used()
Category: Database functions
Source: rdd\dbcmd.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example opens two databases in two different work areas
// and outlines aliased expressions.

   PROCEDURE Main
      USE Customer ALIAS Cust

      DbSelectArea(10)
      USE Invoice ALIAS Inv

      ? Select()                     // result: 10
      ? Cust->( Select() )           // result:  1

      DbSelectArea(0)
      ? Select()                     // result:  2

      ? (10)->( Alias() )            // result: INV
```

```
   CLOSE ALL
RETURN
```

DbSetDriver()

Retrieves and/or selects the default replaceable database driver.

Syntax

```
DbSetDriver( [<cRddName>] ) --> cPreviousRDD
```

Arguments

<cRddName>

This optional parameter is a character string holding the name of the RDD to select as current.

Return

DbSetDriver() returns a character string with the name of driver that is active before the function is called.

Description

The function exists for compatibility reasons and is replaced by function RddSetDefault().

Info

See also:	RddSetDefault()
Category:	Database functions, Database drivers
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// see function RddSetDefault() for an example
```

DbSetFilter()

Defines a filter condition for a work area.

Syntax

```
DbSetFilter( <bFilter>, [<cFilter>] ) --> NIL
```

Arguments

`<bFilter>`

A code block containing the filter expression. The expression must yield a logical value.

`<cFilter>`

The filter expression in form of a character string.

Return

The return value is always NIL.

Description

The DbFilter() function defines a filter condition for the current work area in form of the code block *<bFilter>*. All records in the work area where the filter condition yields .F. (false) are ignored during database navigation. As a result, these records become invisible and are filtered.

Although the second parameter *<cFilter>* is optional, it is recommended to specify the filter condition a second time as a character string. Otherwise, the DbFilter() function cannot obtain the filter condition and returns a null string (""), despite of a filter condition being active.

It is recommended to move the record pointer once after calling DbSetFilter() to ensure that the current record is not visible when it does not match the filter condition. This is usually done with DbGoTop().

Optimization: filtering records can be a time consuming task since the filter condition is evaluated for each record during database navigation. This can be optimized when the filter condition matches the index expression of an open index and SET OPTIMIZE ist set to ON.

Info

See also:	DbClearFilter(), DbFilter(), SET DELETED, SET FILTER, SET OPTIMIZE
Category:	Database functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates the effect of a filter condition

   PROCEDURE Main
      USE Customer
      INDEX ON Upper(LastName) TO Cust01

      GO TOP
      ? Recno(), Lastname              // result: 28 Abbey

      DbSetFilter( {|| Upper(LastName) > "L" }, ;
                   'Upper(LastName) > "L"'   )

      GO TOP
      ? Recno(), Lastname              // result: 182 MacDonald
```

```
   ? DbFilter()                      // result: Upper(LastName) > "L"

   CLOSE Customer
RETURN
```

DbSetIndex()

Opens an index file.

Syntax

```
DbSetIndex( <cIndexFile> ) --> NIL
```

Arguments

`<cIndexFile>`

<cIndexFile> is a character string with the name of the file containing the index(es) to add to the list of open indexes in a work area. The file name can be specified including path information and extension. When the file extension is omitted, it is determined by the database driver that opened the database file in the work area.

Return

The return value is always NIL.

Description

DbSetIndex() exists for compatibility reasons. It is superseded by OrdListAdd().

Note that DBSetIndex() opens an index file only, it does not change the controlling index, unless it is the first index to open.

Info

See also:	DbClearIndex(), DbCreateIndex(), DbSetOrder(), OrdListAdd(), SET INDEX
Category:	Database functions, Index functions
Source:	rdd\rddord.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// refer to the OrdListAdd() function for an example
```

DbSetOrder()

Selects the controlling index.

Syntax

```
DbSetOrder( <nIndexPos> ) --> NIL
```

Arguments

<nIndexPos>

This is the numeric ordinal position of the index to activate as the controlling index. Indexes are numbered in the sequence they are opened with OrdListAdd(), beginning with 1. The value 0 has a special meaning: it de-activates the controlling index while leaving it open. Records are navigated in physical order when *<nIndexPos>* is set to 0.

Return

The return value is always NIL.

Description

DbSetOrder() exists for compatibility reasons. It is superseded by OrdSetFocus().

Info

See also:	DbClearIndex(), DbCreateIndex(), DbSetIndex(), OrdSetFocus(), SET ORDER
Category:	Database functions, Index functions
Source:	rdd\rddord.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// refer to the OrdSetFocus() function for an example
```

DbSetRelation()

Relates a child work area with a parent work area.

Syntax

```
DbSetRelation( <nArea> | <cAlias>, ;
               <bRelation>       , ;
               <cRelation>       , ;
              [<lScoped>]          ) --> NIL
```

Arguments

`<nArea>`

The numeric ordinal position of the child work area to relate to the current work area.

`<cAlias>`

Optionally, the child work area can be specified as a character string holding the symbolic alias name of the child work area.

`<bRelation>`

A code block containing the relation expression.

`<cRelation>`

The relation expression in form of a character string.

`<lScoped>`

The default value for *<lScoped>* is .F. (false), showing no effect on the relation. If .T. (true) is passed, database navigation in the child work area is restricted (scoped) to the records that relate to the current record in the parent work area.

Return

The return value is always NIL.

Description

The DbSetRelation() function relates a parent work area with the child work area specified as *<nArea>* or *<cAlias>*. This causes the record pointer in a child work area to be synchronized with the record pointer in the parent work area, based on the expression *<bRelation>*. All existing relations remain active.

Synchronization of the record pointer in a child work area is accomplished either relative via an index, or absolute via a record number.

Relative synchronization

This requires a controlling index in the child work area. Each time the record pointer moves in the parent work area, the return value of *<bRelation>* is SEEKed in the child work area. As a consequence, the data type of *<bRelation>* must match the data type of the controlling index in the child work area.

Absolute synchronization

When the child work area has no controlling index, or when the type of the index expression is not numeric and the relation expression is numeric, the child work area is synchronized via GOTO with the record number of the parent work area.

Scoped relation

If .T. (true) is passed for *<lScoped>* and the child work area is selected as current work area, record pointer navigation in the child work area is restricted to the records where *<bRelation>* yields the same

values in both, parent and child work area. As a result, the child work area presents a subset of records that match with the current record in the parent work area.

Notes

The record pointer in the child work area is positioned on Lastrec()+1 when there is no match with the relation expression.

It is illegal to relate a parent work area directly or indirectly with itself.

DbSetRelation() does not support SOFTSEEK. It always acts as if SOFTSEEK is set to OFF.

Info

See also:	Bof(), DbClearRelation(), DbGoto(), DbRelation(), DbRSelect(), DbSeek(), Eof(), Found(), OrdSetRelation(), SET RELATION
Category:	Database functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example takes advantage of a scoped relation between an Invoice
// database and an Invoice Items database. The items belonging to an
// invoice are printed in a DO WHILE .NOT. Eof() loop, since the
// relation is scoped in the child work area.

   PROCEDURE Main
      LOCAL i

      USE Invoice ALIAS Inv NEW

      USE InvItems ALIAS Items NEW
      INDEX ON InvoiceNo + ItemID TO InvItemsA

      SELECT Inv
      DBSetRelation( "Items", {|| InvoiceNo }, "InvoiceNo", .T. )

      FOR i:=1 TO 10
         ? Inv->InvoiceNo, Inv->Date
         SELECT Items

         DO WHILE .NOT. Eof()
           ? Space(6), Items->Item_ID, Items->Descript, Items->Price
           SKIP
         ENDDO

         SELECT Inv
         SKIP
      NEXT

   RETURN
```

DbSkip()

Moves the record pointer in a work area.

Syntax

```
DbSkip( [<nRecords>] ) --> NIL
```

Arguments

<nRecords>

Numbers of records to move the record pointer. Positive values for *<nRecords>* move the record pointer forwards (towards the end of file), negative values move it backwards. The default value is 1, i.e. calling DbSkip() with no parameter advances the record pointer to the next record.

Return

The return value is always NIL.

Description

This function moves the record pointer by the number of *<nRecords>* records in the current work area, unless prefixed with an alias. Record pointer movement is relative to the current record and follows the logical order active in a work area. That is, indexes, filters and scopes define the order in which records are navigated.

The record pointer cannot be moved beyond the begin or end of file. An attempt to do so causes either Bof() or Eof() be set to .T. (true), and the record pointer remains unchanged.

Note: DbSkip(0) flushes and refreshes the internal database buffers. All changes made to the record become visible without moving the record pointer in either direction.

Info

See also:	Bof(), DbGoBottom(), DbGoto(), DbGoTop(), DbSeek(), Eof(), GO, OrdSkipRaw(), SKIP
Category:	Database functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows a typical coding pattern for scanning a database
// in a DO WHILE loop based on a condition. DbSkip() is always called
// at the end of the loop, when the condition must be tested for the
// next record.

   PROCEDURE Main()
      USE Customer ALIAS Cust NEW

      DO WHILE .NOT. EoF()
         ? Cust->Firstname,Cust->Lastname
         DbSkip()
      ENDDO

      USE
   RETURN
```

DbSkipper()

Helper function for browse objects to skip a database

Syntax

```
DbSkipper( <nSkipRequest> ) --> nSkipResult
```

Arguments

<nSkipRequest>

A numeric value indicating the number of records to skip the record pointer.

Return

The return value is the number of records that are actually skipped.

Description

DbSkipper() is a helper function used in the :skipBlock code block of the TBrowse object. It provides for standard skip behaviour when browsing data in a work area.

Info

See also:	DbSkip(), SKIP, TBrowse()
Category:	Object functions, Database functions, xHarbour extensions
Source:	rtl\browdbx.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example builds a simple TBrowse object using standard
// navigation code blocks.

   #include "inkey.ch"

   PROCEDURE Main
      LOCAL oTBrowse, aFields, cField, nKey

      USE Customer

      aFields := Array( FCount() )
      AEval( aFields, {|x,i| aFields[i] := FieldName(i) } )

      oTBrowse                := TBrowseNew()
      oTBrowse:skipBlock      := {|n| DbSkipper(n) }
      oTBrowse:goTopBlock     := {|| DbGoTop() }
      oTBrowse:goBottomBlock := {|| DbGoBottom() }

      WITH OBJECT oTBrowse
         FOR EACH cField IN aFields
            :addColumn( TBColumnNew( cField, FieldBlock( cField ) ) )
         NEXT
      END

      nKey := 0
      DO WHILE nKey <> K_ESC

         WITH OBJECT oTBrowse
            DO WHILE .NOT. :stabilize()
```

```
         ENDDO

         IF oTBrowse:hitTop
            Tone(1000)
         ELSEIF oTBrowse:hitBottom
            Tone(500)
         ENDIF

         nKey := Inkey(0)

         SWITCH nKey
         CASE K_UP
           :up()       ; EXIT
         CASE K_DOWN
           :down()     ; EXIT
         CASE K_LEFT
           :left()     ; EXIT
         CASE K_RIGHT
           :right()    ; EXIT
         CASE K_PGUP
           :pageUp()   ; EXIT
         CASE K_PGDN
           :pageDown() ; EXIT
         CASE K_CTRL_PGUP
           :goTop()    ; EXIT
         CASE K_CTRL_PGDN
           :goBottom() ; EXIT
         CASE K_HOME
           :home()     ; EXIT
         CASE K_END
           :end()      ; EXIT
         END
      END
   ENDDO

   CLOSE Customer
RETURN
```

DbSort()

Creates a new, physically sorted database.

Syntax

```
DbSort( <cDatabase>, ;
         <aFields>  , ;
        [<bFor>]    , ;
        [<bWhile>]  , ;
        [<nNext>]   , ;
        [<nRecord>] , ;
        [<lRest>]     ) --> lSuccess
```

Arguments

<cDatabase>

This is a character string holding the name of the database file to create. It can include path and file extension. When no path is given, the file is created in the current directory. The default file extension is DBF.

<aFields>

A one dimensional array holding the names of database fields to sort must be specified. Each element of the array holds a character string with a field name. The character string may be extended by a flag specifying the sort order. /A is the ascending flag (default), /D means descending sort, and /C is relevant for case insensitive sorting of character fields.

<bFor>

This is an optional code block which must return a logical value. The sorted records are added to the target database when *<bFor>* yields .T. (true), otherwise they are ignored.

<bWhile>

This is an optional code block which must return a logical value. DbSort() returns immediately as soon as *<bWhile>* yields .F. (false).

<nNext>

An optional numeric parameter restricting the number of records to sort to *<nNext>*, beginning with the current record.

<nRecord>

Only a single record is processed when the record number *<nRecord>* is specified.

<lRest>

An optional logical value that defaults to .F. (false). Specifying .T. (true) causes DbList() to start output with the current record and continue until the end of file is reached.

Return

The function returns .T. (true) on success, and .F. (false) on failure.

Description

DbSort() is the functional equivalent of the SORT command. Refer to SORT for a detailed description of sorting records into a new database file.

Info

See also: INDEX, OrdCreate(), SORT
Category: Database functions, xHarbour extensions
Source: rdd\dbsort.prg
LIB: xhb.lib
DLL: xhbdll.dll

DbStruct()

Loads structural information of a database into an array.

Syntax

```
DbStruct() --> aStructure
```

Return

DbStruct() returns a two-dimensional array of four columns, filled with structural information of a database.

Description

The DbStruct() function returns a two-dimensional array with FCount() elements. Each element, again, contains a sub-array with four elements holding information about each field of the database open in the current work area. The elements in the sub-arrays can be accessed using #define constants found in the DBSTRUCT.CH file.

Constants for the DbStruct() array

Constant	Value	Meaning
DBS_NAME	1	Field name
DBS_TYPE	2	Field type
DBS_LEN	3	Field length
DBS_DEC	4	Field decimals

Info

See also: AFields(), COPY STRUCTURE EXTENDED, CREATE FROM, DbCopyStruct(), DbCopyExtStruct(), DbCreate()
Category: Database functions
Header: DbStruct.ch
Source: rdd\dbcmd.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example is the code for a useful command line utility
// that lists the database structure in a shell window.

   #include "DbStruct.ch"

   PROCEDURE Main( cDbfFile )
      LOCAL aStruct

      IF Empty( cDbfFile ) .OR. .NOT. File( cDbfFile )
         CLS
         ? "Usage: dbstruct.exe <dbf file>"
         QUIT
      ENDIF
      USE (cDbfFile)

      aStruct := DbStruct()
      AEval( aStruct, {|a| QOut( PadR( a[DBS_NAME], 10 ), ;
                                      a[DBS_TYPE]      , ;
                                 Str( a[DBS_LEN ],  3 ), ;
                                 Str( a[DBS_DEC ],  3 )  ;
```

```
                                    ) } )
      USE
   RETURN
```

DbTableExt()

Retrieves the default database file extension of the current RDD.

Syntax

```
DbTableExt() --> cFileExtension
```

Return

The function returns the default extension for database files used by the current RDD as a character string.

Description

DbTableExt() returns the extension for database files used by the RDD that opens a database in the current work area. Different RDDs may maintain different default file extensions. They are used when a database file is created or opened and the file name is specified without extension.

Note: DbTableExt() does not return the file extension of an open database.

Info

See also:	DbInfo(), DbOrderInfo(), OrdBagExt(), RddInfo()
Category:	Database functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

DbTotal()

Creates a new database summarizing numeric fields by an expression.

Syntax

```
DbTotal( <cDatabase>  , ;
         <bExpression>, ;
        [<aFields>]   , ;
        [<bFor>]      , ;
        [<bWhile>]    , ;
        [<nNext>]     , ;
        [<nRecord>]   , ;
        [<lRest>]       ) --> lSuccess
```

Arguments

<cDatabase>

This is a character string holding the name of the database file to create. It can include path and file extension. When no path is given, the file is created in the current directory. The default file extension is DBF.

<bExpression>

This is a code block whose return value identifies groups of records to summarize. Each time the return value of *<bExpression>* changes, a new record is added to the target database and a new calculation begins. Records in a work area should be indexed or sorted by *<bExpression>*.

<aFields>

A one dimensional array holding the names of database fields to use in the calculation must be specified. Each element of the array holds a character string with the name of a numeric field

<bFor>

This is an optional code block which must return a logical value. The records are used in the calculation when *<bFor>* yields .T. (true), otherwise they are ignored.

<bWhile>

This is an optional code block which must return a logical value. DbTotal() returns immediately as soon as *<bWhile>* yields .F. (false).

<nNext>

An optional numeric parameter restricting the number of records to use in the calculation to *<nNext>*, beginning with the current record.

<nRecord>

Only a single record is processed when the record number *<nRecord>* is specified.

<lRest>

An optional logical value that defaults to .F. (false). Specifying .T. (true) causes DbTotal() to start calculating with the current record and continue until the end of file is reached.

Return

The function returns .T. (true) on success, and .F. (false) on failure.

Description

DbTotal() is the functional equivalent of the TOTAL command. Refer to TOTAL for a detailed description of calculating totals from groups of numeric fields and collecting the results in a new database file.

Info

See also:	AVERAGE, DbEval(), DbUpdate(), SUM, TOTAL, UPDATE
Category:	Database functions, xHarbour extensions
Source:	rdd\dbtotal.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

DbUnlock()

Releases file and all record locks in a work area.

Syntax

```
DbUnlock() --> NIL
```

Return

The return value is always NIL.

Description

This function releases all locks, i.e. file and record locks, in the current work area. Use an aliased expression to release locks in a different work area.

File locks are set with FLock() and are required when multiple records are changed during a single database operation, while record locks are set with RLock() or DbRLock().

Info

See also:	DbRLock(), DbRLockList(), DbUnlockAll(), FLock(), RLock(), UNLOCK
Category:	Database functions, Network functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

DbUnlockAll()

Unlocks all records and releases all file locks in all work areas.

Syntax

```
DbUnlockAll() --> NIL
```

Return

The return value is always NIL.

Description

The function iterates over all open work areas and releases file and record locks in each work area. It does the same as function DbUnlock() so that all work areas are processed within one function call.

Info

See also:	DbRLock(), DbRLockList(), DbUnlock(), FLock(), RLock(), UNLOCK
Category:	Database functions, Network functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

DbUpdate()

Updates records in the current work area from a second work area.

Syntax

```
DbUpdate( <cAlias>     , ;
          <bReplace>   , ;
          <bExpression>, ;
         [<lRandom>]     ) --> lSuccess
```

Arguments

<cAlias>

This is a character string holding the the alias name of the second work area used update the current work area.

<bReplace>

This code block must contain the assignment operations for updating fields.

<bExpression>

This is a code block whose return value is searched in the second work area to find the records for updating the current work area. The current work area must be indexed or sorted in *<bExpression>* order.

<lRandom>

When *<lRandom>* is set to .T. (true) records in the *<cAlias>* work area can be in any order. The default is .F. (false), so thath the *<cAlias>* work area must be sorted or indexed in *<bExpression>* order.

Return

The function returns .T. (true) on success, and .F. (false) on failure.

Description

DbUpdate() is the functional equivalent of the UPDATE command. Refer to UPDATE for a detailed description of updating records from a second database file.

Info

See also:	DbEval(), DbSort(), DbTotal(), FieldPut(), INDEX, OrdCreate(), SUM, UPDATE
Category:	Database functions, xHarbour extensions
Source:	rdd\dbupdat.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

DbUseArea()

Opens a database file in a work area.

Syntax

```
DbUseArea( [<lNewArea>]   , ;
           [<cRddName>]   , ;
           <cDatabase>    , ;
           [<cAlias>]     , ;
           [<lShared>]    , ;
           [<lReadonly>] , ;
           [<cCodePage>] , ;
           [<nConnection>] ) --> NIL
```

Arguments

<lNewArea>

If .T. (true) is passed for *<lNewArea>*, the function selects the next unused work area before the database is opened. The default value is .F. (false), which opens the database in the current work area. If the current work area is used, all files are closed before the new database is opened.

<cRddName>

<cRddName> is an optional character string with the name of the RDD to use for opening the database file. It defaults to the return value of RddSetDefault().

<cDatabase>

This is a character string holding the name of the database file to open. It can include path and file extension. The default file extension is DBF.

<cAlias>

This is the symbolic alias name of the work area as a character string. It defaults to the file name of *<cDatabase>* without extension.

<lShared>

Specifying .T. (true) for *<lShared>* opens the database in SHARED mode. The default value depends on the SET EXCLUSIVE setting. If set to ON, *<lShared>* defaults to .F. (false).

<lReadonly>

Specifying .T. (true) for *<lReadonly>* opens the database in READONLY mode. The default value is .F. (false) which opens the file for read and write access.

<cCodePage>

This is a character string specifying the code page to use for character strings stored in the database. It defaults to the return value of HB_SetCodePage().

<nConnection>

This parameter specifies a numeric server connection handle. It is returned by a server connection function which establishes a connection to a database server, such as SR_AddConnection() of the xHarbour Builder SQLRDD. When *<nConnection>* is passed, the function opens a database on the server.

Return

The return value is always NIL.

Description

DbUseArea() opens an existing database file named <*cDatabase*> in the current work area, or in the next unused work area when <*lNewArea*> is set to .T. (true). The file is searched in the following directories: first in the current directory, then in the SET DEFAULT directory and finally in all directories specified with SET PATH. If the file cannot be found, a runtime error is raised.

When a database is to be accessed in a network, it should be opened in SHARED mode, since EXCLUSIVE usage of a database file prevents other work stations from accessing the file. Shared database access requires to lock a database before changes can be written to a file. Refer to RLock() and FLock() for information about record and file locks in shared database access.

Info

See also: CLOSE, DbCloseArea(), NetDbUse(), RddSetDefault(), Select(), SET DEFAULT, SET PATH, Set(), USE
Category: Database functions
Source: rdd\dbcmd.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example opens a database file using command
// and function syntax.

   REQUEST DBFCDX

   PROCEDURE Main
      USE Customer ALIAS Cust NEW
      ? Alias()                        // result: CUST

      DbUseArea( .T., "DBFCDX" , "Orders", "Ord" )
      ? Alias()                        // result: ORD

      CLOSE ALL
   RETURN
```

Default()

Assigns a default value to a variable.

Syntax

```
Default( @<varName>, <xDefaultValue> ) --> NIL
```

Arguments

`@<varName>`

This is the symbolic names of the variable to assign a default value to. The variable must be passed by reference.

`<xDefaultValue>`

This value is assigned to *<varName>* when the variable has the value NIL. i.e. when it is not initialized.

Return

The function returns always NIL.

Info

See also: DEFAULT TO
Category: CT:Miscellaneous, Miscellaneous functions, xHarbour extensions
Source: ct\util.prg
LIB: xhb.lib
DLL: xhbdll.dll

DefPath()

Returns the SET DEFAULT directory.

Syntax

```
DefPath() --> cDefaultPath
```

Return

The function returns the SET DEFAULT directory as a character string.

Description

Function DefPath() queries the SET DEFAULT directory. If SET DEFAULT is not set, a null string ("") is returned

Info

See also:	CurDir(), CurDrive(), SET DEFAULT
Category:	Directory functions, File functions, xHarbour extensions
Source:	rtl\defpath.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Deleted()

Queries the Deleted flag of the current record

Syntax

```
Deleted() --> lIsDeleted
```

Return

The function returns .T. (true) if the current record is marked for deletion, otherwise .F. (false) is returned.

Description

The Deleted() function queries the Deleted flag of the current record. By default, it operates in the current work area. Use an aliased expression to retrieve the flag from a different work area.

The Deleted flag is set with function DbDelete() and can be removed with DbRecall(). Whether or not a record carries this flag is the task of the Deleted() function.

Note that records marked for deletion are not visible during database navigation when SET DELETED is set to ON.

Info

See also:	DbRecall(), DbRecordInfo(), DELETE, PACK, RECALL, SET DELETED
Category:	Database functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// This example uses Deleted() as part of an index expression so that
// all records carrying the deleted flag appear at the end of a database.
// This allows for an elegant technique of record recycling when new
// data must be stored. If a record is Deleted() its data is overwritten,
// otherwise a new record is appended.

   PROCEDURE Main

      USE Customer NEW
      INDEX ON IIf(Deleted(), Chr(255)+LastName, LastName+Chr(32) ) ;
            TO Cust01

      GO BOTTOM
      IF Deleted()
         nRecno := Recno()
         DbRecall()
      ELSE
         APPEND BLANK
         nRecno := Recno()
      ENDIF

      GOTO nRecno

      <assign new customer data here>

      USE
   RETURN
```

DeleteFile()

Deletes a file with error handling.

Syntax

```
DeleteFile( <cFileName> ) --> nErrorCode
```

Arguments

<cFileName>

This is a character string holding the name of the file to delete. It must include path and file extension. If the path is omitted from *<cFileName>*, the file is searched and deleted in the current directory.

Return

The function returns zero on success or a numeric error code on failure.

Info

See also:	FErase(), FileDelete(), RenameFile()
Category:	CT:DiskUtil, Directory functions, Disks and Drives
Source:	ct\disk.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Descend()

Converts a value for a descending index.

Syntax

```
Descend( <xValue> ) --> xDescend
```

Arguments

<xValue>

An expression whose value must be of data type Character, Date, Logic or Numeric.

Return

The function returns the converted value suitable for creating a descending index. The data type is the same as for *<xValue>*, except for Date values. The converted value for Dates is of numeric data type.

Description

Descend() is a utility function that can be used for the creation of a descending index. The argument passed to Descend() is converted so that the resulting value is sorted logically like the initial value in descending order. Any portion of an index expression can include the Descend() function. This allows for sorting partial index expressions in descending order. Any subsequent SEEK operations must convert the searched value using Descend().

If the entire index should be created in descending order, it is recommended to use the DESCENDING option of the INDEX command. This is advantageous since searched values do not need to be converted with Descend().

Info

See also:	CtoD(), DbSeek(), DtoC(), DtoS(), INDEX, OrdCondSet(), OrdCreate(), OrdDescend(), SEEK, Str(), Val()
Category:	Conversion functions
Source:	rtl\descend.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example creates a descending index using the Descend()
// function and outlines how to search in such an index with SEEK.

   PROCEDURE Main

      ? Descend( 789 )               // result: -789

      USE Customer NEW
      INDEX ON Descend(Upper(Lastname)) TO Temp

      GO TOP
      ? FIELD->Lastname              // result: Waters

      GO BOTTOM
      ? FIELD->Lastname              // result: Alberts

      SEEK Descend( "KELLER" )

      ? FIELD->Lastname              // result: Keller
```

```
   USE
RETURN
```

DevOut()

Outputs a value to the current device.

Syntax

```
DevOut( <expression>, [<cColorString>], [<nRow>, <nCol>] ) --> NIL
```

Arguments

`<expression>`

Any expression whose value should be output to the current output device selected with the SET DEVICE command.

`<cColorString>`

This is an optional SetColor() compliant color string that defines the output color. If omitted, the standard color of the currently selected color string is used.

`<nRow> and <nCol>`

Two optional numeric values can be passed that specify the row and column coordinates for output. They default to Row() and Col() for the screen, and PRow() and PCol() for the printer output device.

Return

The return value is always NIL.

Description

DevOut() is a console function that outputs the value of an expression to the screen or the printer.

Info

See also:	@...SAY, Col(), DevOutPict(), DevPos(), QOut() | QQOut(), Row(), SET DEVICE, SetPos()
Category:	Output functions
Source:	rtl\console.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows the output of a value with
// DevOut() and with the command @...SAY:

   PROCEDURE Main

      DevOut( "xHarbour compiler", "W+/R", MaxRow(), 2  )

      @ MaxRow(), 2 SAY "xHarbour compiler" COLOR "W+/R"

   RETURN
```

DevOutPict()

Outputs a PICTURE formatted value to the current device.

Syntax

```
DevOutPict( <expression>, <cPicture> [,<cColorString>] ) --> NIL
```

Arguments

<expression>

Any expression whose value should be output to the current output device selected with the SET DEVICE command.

<cPicture>

A character string holding a PICTURE format which specifies how to format the value of *<expression>* for output. Refer to Transform() for picture formats.

<cColorString>

This is an optional SetColor() compliant color string that defines the output color. If omitted, the standard color of the currently selected color string is used.

Return

The return value is always NIL.

Description

DevOutPict() is a console function that outputs the value of any expression using a picture transformation rule instead of using the default transformation for the type of *<expression>*.

Info

See also:	@...SAY, Col(), DevOut(), DevPos(), QOut() | QQOut(), Row(), SET DEVICE, SetPos(), Transform()
Category:	Output functions
Source:	rtl\devoutp.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// Output a negative dollar amount using debit notation.

   PROCEDURE Main

      DevOutPict( -12.45, "@X $ 99,999.99" )   // result: $     12.45 DB

   RETURN
```

DevPos()

Moves the cursor or printhead to a row and column coordinate

Syntax

```
DevPos( <nRow>, <nCol> ) --> NIL
```

Arguments

`@ <nRow>, <nCol>`

The parameters are numeric values specifying the row and column coordinates for output. The range for rows on the screen is 0 to MaxRow(), and for columns it is 0 to MaxCol(). The coordinate 0,0 is the upper left corner of the screen.

When SET DEVICE TO PRINTER is active, the largest coordinate for both, row and column, is 32766.

Return

The return value is always NIL.

Description

The DevPos() function accepts numeric row and column coordinates and positions the screen cursor or printhead accordingly. Coordinates begin at point 0, 0 which is the upper left corner of the screen, or paper.

When SET DEVICE is set to SCREEN, DevPos() moves the screen cursor to the specified coordinates and updates Row() and Col().

If the current device is the PRINTER, DevPos() moves the printhead to the new row and column coordinate, taking any SET MARGIN setting into account. In addition, the current printhead position as reported by PRow() and PCol() is considered. That is, if *<nRow>* id smaller than PRow(), a formfeed (Chr(12)) is sent to the printer causing the printhead be positioned on a new page. If *<nCol>* is smaller than PCol(), the printhead is positioned on a new line.

Notes: use SetPrc() to adjust the internal counters of PRow() and PCol() if required.

When printer output is redirected to a file, DevPos() output is recorded in that file.

Info

See also:	@...SAY, Col(), DevOut(), PCol(), PRow(), QOut() | QQOut(), ROW(), SET DEVICE, SetPos(), SetPrc()
Category:	Output functions
Source:	rtl\console.c
LIB:	xhb.lib
DLL:	xhbdll.dll

DirChange()

Changes the current directory.

Syntax

```
DirChange( <cDirectory> ) --> nOSError
```

Arguments

<cDirectory>

A character expression specifying the directory to select as the current directory. The directory may include a drive letter followed by a colon.

Return

The function returns a numeric value representing the operating system error code (DOS error). A value of 0 indicates a successful operation.

Description

The function attempts to change the current directory to the one specified with *<cDirectory>*. If this operation fails, the function returns the OS error code indicating the reason for failure. See the FError() function for a description of OS errors.

Info

See also:	DirRemove(), DiskChange(), DiskName(), FError(), IsDisk(), MakeDir()
Category:	Directory functions, File functions
Source:	rtl\dirdrive.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example changes to existing and non-existing directories.

   PROCEDURE Main
      LOCAL cCurDir := CurDrive() + ":\" + CurDir()

      ? DirChange( "c:\temp" )          // result: 0

      ? DirChange( ".\data" )           // result: 2

      ? DirChange( cCurDir )            // result: 0

   RETURN
```

Directory()

Loads file and directory information into a two-dimensional array.

Syntax

```
Directory( <cDirSpec>, [<cAttributes>] ) --> aDirectory
```

Arguments

<cDirSpec>

This is a character string holding the drive, directory and/or file specification to retrieve information for. It defaults to the string "*.*" which retrieves all available information from the operating system.

<cAttributes>

Optionally, a character string holding file attributes can be specified. Information about files carrying these attributes is retrieved. One or more characters of the table below can be included in *<cAttributes>*.

Attributes for Directory()

Attribute	Meaning
D	Include directories
H	Include hidden files
S	Include system files
V	Search for the DOS volume label and exclude all other files

Return

The function returns a two-dimensional array holding information about files that match *<cDirSpec>*. If no matching file is found, or if an error occurs, the return value is an empty array.

Description

The Directory() function is used to collect file and/or directory information in a two dimensional array of five columns. Each column can be accessed using #define constants from the DIRECTRY.CH file.

Constants for the Directory() array

Constant	Position	Description	Data type
F_NAME	1	File name	Character
F_SIZE	2	File size in bytes	Numeric
F_DATE	3	Creation date	Date
F_TIME	4	Creation time	Character
F_ATTR	5	File attributes	Character

Note that *<cDirSpec>* can include wild card characters to specify a group of files.

Info

See also: ADir(), AEval(), CurDir(), DirChange(), DirectoryRecurse(), DirRemove(), File(), FileStats(), IsDirectory(), MakeDir()
Category: Directory functions, File functions
Header: Directry.ch
Source: rtl\direct.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example first retrieves information about a group of files and
// then the volume information of the C: drive

   #include "Directry.ch"

   PROCEDURE Main
      LOCAL aDir

      aDir := Directory( "*.prg" )

      AEval( aDir, {|a| QOut( a[F_NAME] ) } )

      aDir := Directory( "C:", "V" )

      AEval( aDir[1], {|x| QOut( x ) } )

   RETURN
```

DirectoryRecurse()

Loads file information recursively into a two-dimensional array.

Syntax

```
DirectoryRecurse( <cDirSpec>, [<cAttributes>] ) --> aDirectory
```

Arguments

<cDirSpec>

This is a character string holding the drive, directory and/or file specification to retrieve information for. It defaults to the string "*.*" which retrieves all avalable information from the operating system.

<cAttributes>

Optionally, a character string holding file attributes can be specified. Information about files carrying these attributes is retrieved. One or more characters of the table below can be included in *<cAttributes>*.

Attributes for DirectoryRecurse()

Attribute	Meaning
D	Include directories
H	Include hidden files
S	Include system files
V	Search for the DOS volume label and exclude all other files

Return

The function returns a two-dimensional array holding information about files that match *<cDirSpec>* in the current directory and its sub-directories. If no matching file is found, or if an error occurs, the return value is an empty array.

Description

The DirectoryRecurse() function is used in the same way as the Directory(). The only difference is that DirectoryRecurse() scans all sub-directories in addition to the directory specified with *<cDirSpec>*. The result is a two dimensional array of five columns. Each column can be accessed using #define constants from the DIRECTRY.CH file.

Constants for the DirectoryRecurse() array

Constant	Position	Description	Data type
F_NAME	1	File name	Character
F_SIZE	2	File size in bytes	Numeric
F_DATE	3	Creation date	Date
F_TIME	4	Creation time	Character
F_ATTR	5	File attributes	Character

Note that *<cDirSpec>* can include wild card characters to specify a group of files.

Info

See also: ADir(), AEval(), CurDir(), DirChange(), Directory(), DirRemove(), File(), FileStats(), IsDirectory(), MakeDir()
Category: Directory functions, File functions, xHarbour extensions
Header: Directry.ch
Source: rtl\direct.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example collects file in formation about PRG files in the
// current directory and its sub-directories, and writes the result
// to a file.

   PROCEDURE Main
      LOCAL aFiles

      SET ALTERNATE TO PrgFiles.txt
      SET ALTERNATE ON

      aFiles := DirectoryRecurse( "*.prg" )
      AEval( aFiles, {|a| QOut( ValToPrg( a ) ) } )

      SET ALTERNATE OFF
      SET ALTERNATE TO
   RETURN
```

DirMake()

Creates a directory.

Syntax

```
DirMake( <cDirectory> ) --> nErrorCode
```

Arguments

<cDirectory>

A character expression specifying the directory to create. The directory can be specified relative to the current directory, or absolute, including a drive letter followed by a colon.

Return

The function returns zero on success or a numeric error code on failure.

Info

See also: DirChange(), DirName(), DirRemove(), MakeDir()
Category: CT:DiskUtil, Directory functions, Disks and Drives
Source: ct\disk.c
LIB: xhb.lib
DLL: xhbdll.dll

DirName()

Returns the current directory.

Syntax

```
DirName() --> cDirectory
```

Return

The function returns the name of the current directory as a character string. The string does not include a drive letter.

Info

See also:	CurDir(), DirChange(), DirMake(), DirRemove()
Category:	CT:DiskUtil, Directory functions, Disks and Drives
Source:	ct\disk.c
LIB:	xhb.lib
DLL:	xhbdll.dll

DirRemove()

Removes a directory.

Syntax

```
DirRemove( <cDirectory> ) --> nOSError
```

Arguments

<cDirectory>

A character expression specifying the directory to remove. The directory may include a drive letter followed by a colon.

Return

The function returns a numeric value representing the operating system error code (DOS error). A value of 0 indicates a successful operation.

Description

The function attempts to remove the directory specified with *<cDirectory>*. If this operation fails, the function returns the OS error code indicating the reason for failure. See the FError() function for a description of OS errors.

Info

See also:	CurDir(), DirChange(), Directory(), IsDisk(), DiskChange(), DiskName(), MakeDir(), FError()
Category:	Directory functions, File functions
Source:	rtl\dirdrive.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example creates and deletes directories and outlines possible
// error conditions.

   PROCEDURE Main
      ? CurDrive()+":\"+CurDir()    // result: C:\xhb\tests

      ? MakeDir( "C:\Temp\Data" )   // result: 0

      ? DirRemove( "Data" )         // result: 2

      ? DirRemove( "C:\Temp\data" ) // result: 0

      ? DirRemove( "C:\temp" )      // result: 145
   RETURN
```

DisableWaitLocks()

Toggles the exclusive file opening mode.

Syntax

```
DisableWaitLocks( [<lNewMode>] ) --> lOldMode
```

Arguments

`<lNewMode>`

This is a logical value determining if an application should wait until it gains exclusive access when opening a file with FOpen().

Return

The function returns the previous mode for exclusive file opening as a logical value.

Description

DisableWaitLocks() is a low-level file function maintaining an internal setting for Fopen(). It is only relevant when a process requests exclusive access to file when opening it. By default, FOpen() waits if the requested file to open is currently in use by another process until it is closed. FOpen() returns only after exclusive access to the requested file is obtained. This behaviour can be changed by passing .T. (true) to DisableWaitLocks(). In this case, FOpen() returns immediately when exclusive access to a file to open is currently not possible, and a file open error is set (see FError()).

Info

See also:	FError(), FOpen()
Category:	File functions, Low level file functions, xHarbour extensions
Source:	rtl\philes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

DiskChange()

Changes the current disk drive.

Syntax

```
DiskChange( <cDrive> ) --> lSuccess
```

Arguments

<cDrive>

This is a single character (A-Z) specifying the disk drive to select as current drive.

Return

The funtion returns .T. (true) if the drive *<cDrive>* is selected as current drive, otherwise .F. (false) is returned.

Description

The function attempts to change the current disk drive and returns .T. (true) when the operation is successful.

Note: DiskChange() uses operating system functionalities. If a disk drive exists but has no disk inserted, the operating system prompts the user for inserting a disk.

To suppress this behavior, call SetErrorMode() and pass 1 to it.

Info

See also: CurDrive(), CurDir(), DirChange(), DiskSpace(), IsDisk(), SetErrorMode()
Category: Directory functions, File functions, xHarbour extensions
Source: rtl\dirdrive.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates basic usage of DiskChange() and
// includes a user defined function that queries disk drives.

   PROCEDURE Main
      LOCAL aDisk, cDisk

      ? CurDrive()                     // result: C
      ? CurDir()                       // result: xhb\tests

      ? DiskChange( "D" )              // result: .T.

      ? CurDrive()                     // result: D
      ? CurDir()                       // result: apps\data

      ? DiskChange( "K")               // result: .F.

      ? CurDrive()                     // result: D
      ? CurDir()                       // result: apps\data

      aDisk := GetDrives()

      FOR EACH cDisk IN aDisk
         ? cDisk
      END
```

```
RETURN

FUNCTION GetDrives()
   LOCAL cDrive  := CurDrive()
   LOCAL aDrives := {}
   LOCAL nDrive  := 1

   FOR nDrive := 1 TO 26
      IF DiskChange( Chr( 64 + nDrive ) )
         AAdd( aDrives, Chr( 64 + nDrive ) )
      ENDIF
   NEXT

   DiskChange( cDrive )
RETURN aDrives
```

DiskFormat()

Formats a floppy disk.

Syntax

```
DiskFormat( [<cDrive>]       , ;
            [<nCapacity>]    , ;
            [<cUDF>]         , ;
            [<cBootText>]    , ;
            [<nRepetitions>], ;
            [<cVolLabel>]    , ;
            [<lBoot>]        , ;
            [<lQuickFormat>]  ) --> nErrorCode
```

Arguments

<cDrive>

This parameter specifies the floppy drive to use. It can be either an upper case *A* or *B*, indicating the drive letter. The default value is *A*.

<nCapacity>

An optional numeric value specifying the capacity of the floppy disk. It defaults to 0. Valid values are 160, 180, 320, 360, 640, 720, 1200, 1440 and 2880.

<cUDF>

This parameter exists for compatibility reasons but is ignored.

<cBootText>

This parameter exists for compatibility reasons but is ignored.

<nRepetitions>

This parameter exists for compatibility reasons but is ignored.

<cVolLabel>

An optional character string of max. 11 characters can be specified for the volume label of the floppy disk.

<lBoot>

This parameter exists for compatibility reasons but is ignored.

<lQuickFormat>

This parameter defaults to .F. (false). When set to .T. (true) the floppy disk is formatted in "quick format" mode.

Return

The function returns a DosError() compatible error code as a numeric value. When the floppy disk is successfully formatted, the error code is zero.

Info

See also: DiskReady(), DosError(), DriveType(), FloppyType()
Category: CT:DiskUtil, Disks and Drives
Source: ct\diskutil.prg
LIB: xhb.lib
DLL: xhbdll.dll

DiskFree()

Returns the free storage space of a disk drive in bytes.

Syntax

```
DiskFree( [<cDrive>] ) --> nFreeDiskSpace
```

Arguments

<cDrive>

This parameter defaults to the current drive letter. It can be specified as a drive letter from A to Z without a colon.

Return

The function returns a numeric value indicating the free storage space of the specified disk.

Info

See also:	CurDrive(), DiskSpace(), DiskTotal(), DiskUsed()
Category:	CT:DiskUtil, Disks and Drives
Source:	ct\diskutil.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

DiskName()

Returns the current disk drive.

Syntax

```
DiskName() --> cDrive
```

Return

The return value is a single character specifying the current disk drive.

Description

The function is used to retrieve the drive letter of the current disk drive.

Info

See also:	CurDrive(), CurDir(), DiskChange(), DiskSpace(), IsDisk()
Category:	Directory functions, File functions
Source:	rtl\dirdrive.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example queries and changes the current disk drive

   PROCEDURE Main
      ? DiskName()                     // result: C

      ? DiskChange( "D" )              // result: .T.

      ? DiskName()                     // result: D
   RETURN
```

DiskReady()

Test if a disk drive is ready.

Syntax

```
DiskReady( [<cDrive>], [<lOSErrorMsg>] ) --> lDriveIsReady
```

Arguments

<cDrive>

This is a single character (A-Z) specifying the disk drive to test. It defaults to CurDrive().

<lOSErrorMsg>

This parameter defaults to .T. (true), causing the operating system to prompt to user for inserting a disk if no disk is inserted in a floppy drive. Passing .F. (false) for *<lOSErrorMsg>* suppresses the operating system error message.

Return

The return value is .T. (true) when the specified drive is ready, otherwise .F. (false) is returned.

Info

See also: DiskReadyW(), IsDisk()
Category: CT:DiskUtil, Disks and Drives
Source: ct\diskutil.prg
LIB: xhb.lib
DLL: xhbdll.dll

DiskReadyW()

Tests if a drive can be written to.

Syntax

```
DiskReadyW( [<cDrive>], [<lOSErrorMsg>] ) --> lIsWriteable
```

Arguments

`<cDrive>`

This is a single character (A-Z) specifying the disk drive to test. It defaults to CurDrive().

`<lOSErrorMsg>`

This parameter defaults to .T. (true), causing the operating system to prompt to user for inserting a disk if no disk is inserted in a floppy drive. Passing .F. (false) for *<lOSErrorMsg>* suppresses the operating system error message.

Return

The return value is .T. (true) when the specified drive can be written to, otherwise .F. (false) is returned.

Info

See also: DiskReady(), NetDisk()
Category: CT:DiskUtil, Disks and Drives
Source: ct\diskutil.prg
LIB: xhb.lib
DLL: xhbdll.dll

DiskSpace()

Returns the free storage space for a disk drive.

Syntax

```
DiskSpace( [<nDrive>], [<nType>] ) --> nBytes
```

Arguments

`<nDrive>`

The drive to query for free disk space is specified as a numeric value in the range of 1 (drive A:) to 26 (drive Z:). Omitting *<nDrive>* or passing 0 returns the disk space avalable on the current drive.

`<nType>`

Optionally, the type of storage space on a drive to query can be specified using #define constants from the FILEIO.CH file. Valid constants are listed below:

Storage space types for DiskSpace()

Constant	Value	Description
HB_DISK_AVAIL *)	0	Free disk space avialable to the application
HB_DISK_FREE	1	Total free disk space
HB_DISK_USED	2	Used disk space
HB_DISK_TOTAL	3	Total disk space

**) default*

Return

The function returns a numeric value indicating the storage space of a disk drive. If no parameters are passed, DiskSpace() returns the free disk space on the current drive that is available to the application.

Description

DiskSpace() determines free and used bytes on a disk drive. Some operating systems distinguish between total free bytes on disk and free bytes that are avalaible to the current user, or application. If the current operating system makes no such distinction, HB_DISK_AVAIL and HB_DISK_FREE yield the same results.

Note: if information is requested on a disk that is not available, a runtime error 2018 is generated.

Info

See also:	Directory(), DiskName(), File(), FOpen(), FSeek(), HB_DiskSpace(), IsDisk(), LastRec(), LUpdate(), RecCount(), RecSize()
Category:	Disks and Drives, File functions
Header:	fileio.ch
Source:	rtl\diskspac.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example lists two disk storage types for all drives taking
// advantage of the runtime error created for unaccessible drives.

   #include "Fileio.ch"
```

```
PROCEDURE Main
   LOCAL i, bError := ErrorBlock( {|e| Break(e) } )

   FOR i:=1 TO 26
      BEGIN SEQUENCE
        ? "Drive",Chr(64+i)+":", DiskSpace( i, HB_DISK_FREE ), ;
                                 DiskSpace( i, HB_DISK_TOTAL)
      RECOVER
        ?"Drive",Chr(64+i)+":", "not ready or not existent"
      END SEQUENCE
   NEXT

   ErrorBlock( bError )
RETURN
```

DiskTotal()

Returns the totl storage space of a disk drive in bytes.

Syntax

```
DiskTotal( [<cDrive>] ) --> nTotalBytes
```

Arguments

<cDrive>

This parameter defaults to the current drive letter. It can be specified as a drive letter from A to Z without a colon.

Return

The function returns a numeric value indicating the total storage space of the specified disk.

Info

See also:	DiskFree(), DiskSpace(), DiskUsed()
Category:	CT:DiskUtil, Disks and Drives
Source:	ct\diskutil.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

DiskUsed()

Returns the used storage space of a disk drive in bytes.

Syntax

```
DiskUsed( [<cDrive>] ) --> nUsedDiskSpace
```

Arguments

`<cDrive>`

This parameter defaults to the current drive letter. It can be specified as a drive letter from A to Z without a colon.

Return

The function returns a numeric value indicating the used storage space of the specified disk.

Info

See also:	CurDrive(), DiskFree(), DiskSpace(), DiskTotal()
Category:	CT:DiskUtil, Disks and Drives, xHarbour extensions
Source:	ct\diskutil.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

DispBegin()

Initiates buffering of console window output.

Syntax

```
DispBegin() --> NIL
```

Return

The return value is always NIL.

Description

The DispBegin() function is used to buffer a series of commands and/or functions that produce output on the the screen (console window). The display becomes visible when DispEnd() is called.

DispBegin() starts collecting screen output in an internal buffer so that complex output can be produced in a series of function calls. The result of individual function is not displayed, only the entire screen buffer is made visible when DispEnd() is called. Buffered screen output can result in a considerable performance increase.

Note that DispBegin() calls can be nested, i.e. DispBegin() can be called repeatedly without calling DispEnd(). To make the buffered display visible, however, DispEnd() must be called as many times as DispBegin() is called. Buffered screen output becomes only visible after a matching number of DispEnd() calls.

The number of pending DispEnd() calls can be obtained from function DispCount().

Info

See also:	DispCount(), DispEnd(), DispOut()
Category:	Screen functions
Source:	rtl\gt.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example uses buffered screen output to give the impression of
// a box moving behind another box that stays in front.

   #include "Box.ch"

   PROCEDURE Main
      LOCAL cFront, nT:= 0, nL:= 0, nB:= 3, nR:= 30

      CLS

      // display initial box in front
      DispBox( 8, 12, 18, 52, B_SINGLE+" ", "W+/B" )
      DispOut( "I am in front" )

      DO WHILE nT < 22

         // buffered screen output displays second box
         // behind box in front.
         DispBegin()

         cFront := SaveScreen(8,12,18,52)
         DispBox( nT, nL, nB, nR, B_DOUBLE+" ", "N/W*")
         DispOut( "I am moving behind a box" )
```

```
        RestScreen( 8, 12, 18, 52, cFront )

        DispEnd()

        Inkey(0.1)

        // buffered screen output erases moving box
        DispBegin()
        Scroll( nT, nL, nB, nR )
        RestScreen( 8, 12, 18, 52, cFront )
        DispEnd()

        // new coordinates for moving box
        nT++ ; nL++ ; nB++ ; nR++
     ENDDO

  RETURN
```

DispBox()

Displays a box on the screen.

Syntax

```
DispBox( <nTop>, <nLeft>, <nBottom>, <nRight>, ;
        [<cnBoxString>] , [<cColor>] ) --> NIL
```

Arguments

`<nTop> and <nLeft>`

Numeric values indicating the screen coordinates for the upper left corner of the DispBox() output.

`<nBottom> and <nRight>`

Numeric values indicating the screen coordinates for the lower right corner of the DispBox() output.

`<cnBoxString>`

The appearance of the box to display can either be specified as numeric 1 (single line box), numeric 2 (double line box), or as a character string holding up to nine characters. The first eight characters define the border of the box while the ninth character is used to fill the box. #define constants to be used for *<cnBoxString>* are available in the BOX.CH #include file.

Pre-defined box strings for DispBox()

Constant	Description
B_SINGLE	Single-line box
B_DOUBLE	Double-line box
B_SINGLE_DOUBLE	Single-line top, double-line sides
B_DOUBLE_SINGLE	Double-line top, single-line sides

If no *<cnBoxString>* is specified, a single-line box is drawn.

`<cColor>`

An optional SetColor() compliant color string can be specified to draw the box. It defaults to the standard color of SetColor().

Return

The return value is always NIL.

Description

The function DispBox() displays a box on the screen as specified with *<cnBoxString>*, using the standard color of SetColor() or *<cColor>*, if specified.

If a character string is used for *<cnBoxString>*, the first eight characters define the border of the box in clockwise direction, beginning with the upper left corner. An optional ninth character fills the area inside the box. Alternatively, a single character can be passed which is used to draw the entire box border.

When the box is completely drawn, the cursor is positioned at the coordinates *<nTop>*+1 and *<nLeft>*+1, so that a subsequent DispOut() call starts displaying in the upper left corner of the box area.

Info

See also: @...BOX, @...CLEAR, @...TO, Scroll(), SetColor()
Category: Screen functions
Header: box.ch
Source: rtl\box.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates how characters are used to draw a box.
// Alphabetic characters define <cnBoxString> instead of characters
// holding graphic signs.

   #include "Box.ch"

   PROCEDURE Main
      CLS
      DispBox( 10,10,20,50, "AbCdEfGhi", "W+/R" )

      Inkey(0)

      DispBox( 10,10,20,50, B_DOUBLE + Space(1) )
      DispOut( "Using #define constant" )

      @ MaxRow(),0
   RETURN
```

DispCount()

Retrieves the number of pending DispEnd() calls.

Syntax

```
DispCount() --> nDispCount
```

Return

The function returns a numeric value indicating how often DispEnd() must be called to make buffered screen output visible.

Description

DispCount() is used to determine the number of DispEnd() calls required with nested DispBegin() calls. Buffered screen output becomes only visible, when the number of DispEnd() calls matches exactly with the number of DispBegin() calls.

Info

See also:	DispBegin(), DispEnd(), DispOut()
Category:	Screen functions
Source:	rtl\gt.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example uses a typical coding pattern for DispCount() that
// forces buffered screen output to become visible.

   <code for buffered screen output with DispBegin()>

   DO WHILE DispCount() > 0
      DispEnd()
   ENDDO
```

DispEnd()

Displays buffered screen output.

Syntax

```
DispEnd() --> NIL
```

Return

The return value is always NIL.

Description

DispEnd() terminates buffered screen output initiated with a call to DispBegin() and makes the buffered output visible. DispEnd() must be called as many times as DispBegin(). This is important for nested DispBegin() calls.

Info

See also:	DispBegin(), DispCount(), DispOut()
Category:	Screen functions
Source:	rtl\gt.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// see the example for DispBegin()
```

DispOut()

Displays a value on the screen

Syntax

```
DispOut( <expression>, [<cColor>] ) --> NIL
```

Arguments

<expression>

Any expression whose value should be output to the screen.

<cColor>

An optional SetColor() compliant color string can be specified for display. It defaults to the standard color of SetColor().

Return

The return value is always NIL.

Description

The DispOut() function displays the value of *<expression>* at the current cursor position on the screen, using the color *<cColor>*, if specified. DispOut() ignores the SET DEVICE setting and sends output always to the screen.

Info

See also:	Col(), DevOut(), DispBegin(), DispCount(), DispEnd(), DispOutAt(), OutStd(), QOut() | QQOut(), Row(), SetPos()
Category:	Output functions, Screen functions
Source:	rtl\console.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example changes the current cursor position and displays
// two strings using different colors.

   PROCEDURE Main
      CLS

      SetPos( 10, 20 )

      DispOut( "xHarbour", "W+/R" )

      DispOut( "compiler", "W+/G" )
   RETURN
```

DispOutAt()

Displays a value on the screen at a certain position.

Syntax

```
DispOutAt( <nRow>       , ;
           <nCol>       , ;
           <expression>, ;
           [<cColor>]   , ;
           [<lSetPos>]    ) --> NIL
```

Arguments

<nRow>

This is a numeric value specifying the row on the screen where the value of *<expression>* is displayed.

<nCol>

This is a numeric value specifying the column on the screen where the value of *<expression>* is displayed.

<expression>

Any expression whose value should be output to the screen.

<cColor>

An optional SetColor() compliant color string can be specified for display. It defaults to the standard color of SetColor().

<lSetPos>

An optional logical value specifies whether or not to update the screen cursor position after display. .T. (true) updates the cursor position and .F. (false) leaves the cursor position unchanged. The default is the return value of function DispOutAtSetPos().

Return

The return value is always NIL.

Description

The DispOutAt() function displays the value of *<expression>* at the specified cursor position on the screen, using the color *<cColor>*. The position of the screen cursor after display depends on parameter *<lSetPos>*.

Info

See also:	Col(), DispBegin(), DispCount(), DispEnd(), DispOut(), DispOutAtSetPos(), Row(), SetPos()
Category:	Output functions, Screen functions, xHarbour extensions
Source:	rtl\console.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example uses background processing for displaying the time
// continuously without changing the cursor position while Browse()
// is active.

   PROCEDURE Main
```

```
   LOCAL nIdle, bTask, nTask

   SET BACKGROUND TASKS ON

   bTask := {|| DispoutAt( 0, MaxCol()-7, Time(), "W+/B", .F. ) }
   nTask := HB_BackGroundAdd( bTask, 1000 )
   nIdle := HB_IdleAdd( {|| HB_BackGroundRun() } )

   USE Customer
   Browse()
   USE

   HB_BackGroundDel( nTask )
   HB_IdleDel( nIdle )
RETURN
```

DispOutAtSetPos()

Toggles update of the screen cursor with DispOutAt().

Syntax

```
DispOutAtSetPos( [<lNewSetting>] ) --> lOldSetting
```

Arguments

`<lNewSetting>`

This is a logical value which can change the current setting.

Return

The function returns the old setting.

Description

DispOutAtSetPos() defines the default value for the fifth parameter of function DispoutAt().

Info

See also:	DispOutAt(), SetPos()
Category:	Output functions, Screen functions, xHarbour extensions
Source:	rtl\console.c
LIB:	xhb.lib
DLL:	xhbdll.dll

DllCall()

Executes a function located in a dynamically loaded external library.

Syntax

```
DllCall( <cDllFile>|<nDllHandle>, ;
         [<nCallingConvention>] , ;
         <cFuncName>|<nOrdinal> , ;
        [<xParams,...>]          ) --> nResult
```

Arguments

`<cDllFile>`

A character string holding the name of the DLL file where the function is located. This is an external DLL, not created by xHarbour. If a character string is passed for *<cDllFile>*, it must contain complete path information, unless the file is located in the current directory, or in the list of directories held in the SET PATH environment variable of the operating system.

`<nDllHandle>`

Alternatively, the first parameter can be a numeric DLL handle as returned by function LoadLibrary().

`<nCallingConvention>`

The calling convention to use for the DLL function can optionally be specified. Constants are available for this parameter.

Calling conventions

Constant	Value	Description
DC_CALL_CDECL	0x0010	C calling convention (__cdecl)
DC_CALL_STD *)	0x0020	Standard convention for WinAPI (__stdcall)

**) default*

`<cFuncName>`

This is a character string holding the symbolic name of the function to call. Unlike regular xHarbour functions, this function name is **case sensitive**.

`<nOrdinal>`

Instead of the symbolic function name, the numeric ordinal position of the function inside the DLL file can be passed. This is, however, not recommended, since ordinal positions of functions may change between DLL versions.

`<xParams,...>`

The values of all following parameters specified in a comma separated list are passed on to the DLL function.

Return

The function returns the result of the called DLL function as a numeric value.

Description

Function DllCall() can be used to execute functions located in DLL files which are not created by the xHarbour compiler and linker. This allows for executing functions residing in system DLLs of the operating system or Third Party producers, for example.

The DLL that contains a function is either specified by its numeric DLL handle, or by its symbolic file name. In the latter case, DllCall() loads the library, executes the function and frees the library when the function has returned. When a numeric DLL handle is specified, the DLL is already loaded by LoadLibrary(), and it remains loaded when the DLL function is complete. The DLL must then be released explicitely with FreeLibrary().

When more than three parameters are specified, all <*xParams,...*> are passed on to the DLL function. However, since xHarbour has its own data types and more of them are available than in the C language, only values of a restricted number of data types can be passed. The values are converted to corresponding C data types.

Data type conversion

PRG level	C level
Character	*char
C structure	*void
Date	DWORD
Logical	DWORD
NIL	(DWORD) NULL
Numeric	
- Integer	DWORD
- Decimal number	double
Pointer	*void

If other xHarbour data types are passed to DllCall(), a runtime error is generated.

Note: many Windows API functions are available as a Unicode and an Ansi version. If this is the case, DllCall() uses the Ansi version of the WinAPI function.

Info

See also:	CallDll(), DllExecuteCall(), DllPrepareCall(), FreeLibrary(), GetProcAddress(), GetLastError(), LoadLibrary()
Category:	DLL functions, xHarbour extensions
Source:	rtl\dllcall.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example outlines how a Windows API function can be wrapped
// within an xHarbour function without reverting to xHarbour's
// extend API. The GetVolumeInformation() wrapper obtains only
// the volume name and serial number. Other information available
// from the WinAPI function is ignored. Note that the "out"
// parameters of the API must be passed by reference to DllCall().

   #define DC_CALL_STD            0x0020
   #define MAX_PATH                  260

   PROCEDURE Main
      LOCAL cDrive := "D:\"
      LOCAL cVolName, cVolSerial

      cVolName := GetVolumeInformation( cDrive, @cVolSerial )

      ? cVolName                      // result: BACKUP

      ? cVolSerial                    // result: 584C:2AE1
   RETURN
```

```
FUNCTION GetVolumeInformation( cVolume, cSerial )
   LOCAL cVolumeName := Replicate( Chr(0), MAX_PATH+1 )
   LOCAL nNameSize   := Len( cVolumeName )
   LOCAL nResult

   cSerial := U2Bin(0)

   nResult :=                    ;  // * C prototype *
      DllCall(                   ;
        "Kernel32.dll"         , ;  // DLL to call
         DC_CALL_STD           , ;  // calling vonvention
        "GetVolumeInformation" , ;  // BOOL GetVolumeInformation(
         cVolume               , ;  //   LPCTSTR lpRootPathName ,
        @cVolumeName           , ;  //   LPTSTR lpVolumeNameBuffer ,
         nNameSize             , ;  //   DWORD nVolumeNameSize ,
        @cSerial               , ;  //   LPDWORD lpVolumeSerialNumber ,
         0                     , ;  //   LPDWORD lpMaximumComponentLength ,
         0                     , ;  //   LPDWORD lpFileSystemFlags ,
         0                     , ;  //   LPTSTR lpFileSystemNameBuffer ,
         0                       )  //   DWORD nFileSystemNameSize )

   // format serial number as FFFF:FFFF
   cSerial := NumToHex( Bin2U(cSerial), 8 )
   cSerial := Stuff( cSerial, 5, 0, ":" )

RETURN Left( cVolumeName, At( Chr(0), cVolumeName ) - 1 )
```

DllExecuteCall()

Executes a DLL function via call template.

Syntax

```
DllExecuteCall( <pCallTemplate>, [<xParams,...>] ) --> nResult
```

Arguments

<pCallTemplate>

This is a pointer to the template for calling the DLL function. It is returned from function DllPrepareCall().

<xParams,...>

This is a comma separated list of parameters which are passed to the DLL function to call.

Return

The function returns the result of the executed DLL function as a numeric value.

Description

Function DllExecuteCall() executes a DLL function using the call template as it is prepared by function DllPrepareCall(). The DllPrepareCall()/DllExecuteCall() way of calling a function in an external DLL is advantageous when the same DLL function must be executed many times. The call template is prepared once and contains all information required to execute the DLL function. If a DLL function needs to be called only once or a few times, it can be called directly via DllCall(). The overhead of preparing the call template can be avoided in such cases.

Info

See also:	DllCall(), DllPrepareCall(), FreeLibrary(), GetProcAddress(), GetLastError(), LoadLibrary()
Category:	DLL functions, xHarbour extensions
Source:	rtl\dllcall.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example implements two Windows API wrappers for conversion of
// Ansi strings to Unicode and vice versa. The call templates for
// DllExecuteCall() are created on the first function call and stored
// in STATIC variables.

   #define DC_CALL_STD            0x0020

   PROCEDURE Main
      LOCAL cString, cWideString

      cString     := "Hello World"
      cWideString := AnsiToWide( cString )

      ? Len( cString ), cString
      ? Len( cWideString ), cWideString

      ? WideToAnsi( cWideString )
   RETURN
```

```
FUNCTION AnsiToWide( cString )
   STATIC pCallTemplate

   LOCAL nWideLen  := 2 * ( Len( cString ) )
   LOCAL cWideChar := Replicate( Chr(0), nWideLen )
   LOCAL nRet

   IF pCallTemplate == NIL
      // create call template on first call
      pCallTemplate := DllPrepareCall(  ;
                        "Kernel32.dll", ;      // external DLL
                        DC_CALL_STD   , ;      // calling convention
                       "MultiByteToWideChar" ) // external function
   ENDIF

   nRet := DllExecuteCall(    ;
              pCallTemplate , ;
              0             , ;
              0             , ;
              cString       , ;
              -1            , ;
             @cWideChar     , ;
              nWideLen        )
RETURN cWideChar


FUNCTION WideToAnsi( cWideChar )
   STATIC pCallTemplate

   LOCAL nLen    := Int( Len( cWideChar ) / 2 )
   LOCAL cString := Replicate( Chr(0), nLen )
   LOCAL nRet

   IF pCallTemplate == NIL
      // create call template on first call
      pCallTemplate := DllPrepareCall(  ;
                        "Kernel32.dll", ;      // external DLL
                        DC_CALL_STD   , ;      // calling convention
                       "WideCharToMultiByte" ) // external function
   ENDIF

   nRet := DllExecuteCall(    ;
              pCallTemplate , ;
              0             , ;
              0             , ;
              cWideChar     , ;
              -1            , ;
             @cstring       , ;
              nLen          , ;
              0             , ;
              0               )
RETURN cString
```

DllLoad()

Loads a DLL file into memory.

Syntax

```
DllLoad( <cDLLFileName> ) --> nDllHandle
```

Arguments

<cDLLFileName>

This is a character string holding the name of the DLL file to load into memory.

Return

The function returns a numeric DLL handle > 0. If the DLL file cannot be loaded, the return value is zero.

Description

Function DllLoad() is a synonym for function LoadLibrary(). Refer to LoadLibrary().

Info

See also:	LoadLibrary()
Category:	DLL functions, xHarbour extensions
Source:	rtl\dllcall.c
LIB:	xhb.lib
DLL:	xhbdll.dll

DllPrepareCall()

Creates a call template for an external DLL function.

Syntax

```
DllPrepareCall( <cDllFile>|<nDllHandle>, ;
                [<nCallingConvention>]  , ;
                <cFuncName>|<nOrdinal>    ) --> pCallTemplate
```

Arguments

`<cDllFile>`

A character string holding the name of the DLL file where the function is located. This is an external DLL, not created by xHarbour. If a character string is passed for *<cDllFile>*, it must contain complete path information, unless the file is located in the current directory, or in the list of directories held in the SET PATH environment variable of the operating system.

`<nDllHandle>`

Alternatively, the first parameter can be a numeric DLL handle as returned by function LoadLibrary().

`<nCallingConvention>`

The calling convention to use for the DLL function can optionally be specified. Constants are available for this parameter.

Calling conventions

Constant	Value	Description
DC_CALL_CDECL	0x0010	C calling convention (__cdecl)
DC_CALL_STD *)	0x0020	Standard convention for WinAPI (__stdcall)

**) default*

`<cFuncName>`

This is a character string holding the symbolic name of the function to call. Unlike regular xHarbour functions, this function name is **case sensitive**.

`<nOrdinal>`

Instead of the symbolic function name, the numeric ordinal position of the function inside the DLL file can be passed. This is, however, not recommended, since ordinal positions of functions may change between DLL versions.

Return

The function returns a pointer to the call template for the specified DLL function.

Description

Function DllPrepareCall() prepares a call template which contains all information required to invoke a DLL function, except for the parameters to pass. The call template is then passed along with the parameters to function DllExecuteCall() which executes the DLL function (see DllExecuteCall() for a complete example).

The preparation of a call template is advantageous when a DLL function must be called many times, since the information assembled in the call template is the same for multiple DLL function calls. If a DLL function needs to be called only once or a few times, it can be executed via DllCall() since the extra overhead of the call template preparation is not justified in such case.

Important: the call template must not be changed.

Info

See also: DllExecuteCall(), FreeLibrary(), GetProcAddress(), GetLastError(), LoadLibrary()
Category: DLL functions, xHarbour extensions
Source: rtl\dllcall.c
LIB: xhb.lib
DLL: xhbdll.dll

DllUnload()

Releases a dynamically loaded external DLL from memory.

Syntax

```
DllUnload( <nDllHandle> ) --> lSuccess
```

Arguments

<nDllhandle>

This is a numeric handle of the DLL to release as returned from LoadLibrary().

Return

The function returns a logical value inidcating a successful operation.

Description

Function DllUnload() is a synonym for FreeLibrary(). Refer to FreeLibrary().

Info

See also:	FreeLibrary()
Category:	DLL functions, xHarbour extensions
Source:	rtl\dllcall.c
LIB:	xhb.lib
DLL:	xhbdll.dll

DMY()

Formats a date as "dd. Month yyyy"

Syntax

```
DMY( [<dDate>], [<lPeriod>] ) --> cDate
```

Arguments

<dDate>

An expression returning a Date value. It defaults to Date().

<lPeriod>

Optionally, a logical value can be passed. If *<lPeriod>* is .T. (true), a period is inserted after the day number in the returned string. The default value is .F. (false).

Return

The function returns the formatted date as a character string. The string contains the month name. A two digit year is inserted when SET CENTURY is OFF.

Info

See also:	MDY(), SET CENTURY, SET DATE
Category:	CT:DateTime, Date and time
Source:	ct\dattime2.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays formatted dates.

   PROCEDURE Main
      LOCAL dDate := StoD( "20061101" )

      ? DMY( dDate )        // result: 1 November 06
      ? DMY( dDate, .T. )   // result: 1. November 06

      SET CENTURY ON
      ? DMY( dDate, .T. )   // result: 1. November 2006
   RETURN
```

DosError()

Sets or retrieves the last DOS error code.

Syntax

```
DosError( [<nNewErrorCode>] ) --> nOsErrorCode
```

Arguments

`<nNewErrorCode>`

This numeric parameter defines the return value for subsequent DosError() calls.

Return

The function returns the last DOS error code as a numeric value.

Description

The DosError() function returns an error code of the Disk Operating System (DOS) that is set when a disk, directory or file operation fails. This leads normally to a runtime error, so that DosError() is usually called within error handling routines to test if a DOS error has occured. DosError() retains its value until a new DOS error occurs, or a new return value is defined explicitely with *<nNewErrorCode>*. The return value of DosError() is set to zero after each successful DOS operation. The following table gives an overview of some DOS error codes together with their meaning.

Disk Operating System (DOS) error codes

Error	Description
1	Invalid function number
2	File not found
3	Path not found
4	Too many open files (no handles left)
5	Access denied
6	Invalid handle
7	Memory control blocks destroyed
8	Insufficient memory
9	Invalid memory block address
10	Invalid environment
11	Invalid format
12	Invalid access code
13	Invalid data
15	Invalid drive was specified
16	Attempt to remove the current directory
17	Not same device
18	No more files
19	Attempt to write on write-protected media
20	Unknown unit
21	Drive not ready
22	Unknown command
23	Cyclic redundancy check (CRC) -- part of diskette is bad
24	Bad request structure length
25	Seek error
26	Unknown media type
27	Sector not found
28	Printer out of paper
29	Write fault
30	Read fault

31	General failure
32	Sharing violation
33	Lock violation
34	Invalid disk change
35	FCB unavailable
36	Sharing buffer overflow
38	Unable to complete the operation
50	Network request not supported
51	Remote computer not listening
52	Duplicate name on network
53	Network path not found
54	Network busy
55	Network device no longer exists
56	NETBIOS command limit exceeded
57	System error, NETBIOS error
58	Incorrect response from network
59	Unexpected network error
60	Incompatible remote adapter
61	Print queue full
62	Not enough space for print file
63	Print file was cancelled
64	Network name was denied
65	Access denied
66	Network device type incorrect
67	Network name not found
68	Network name limit exceeded
69	NETBIOS session limit exceeded
70	Sharing temporarily paused
71	Network request not accepted
72	Print or disk redirection is paused
80	File exists
82	Cannot make directory entry
83	Fail on INT 24
84	Too many redirections
85	Duplicate redirection
86	Invalid password
87	Invalid parameter
88	Network data fault
89	Function not supported by network
90	Required system component not installed

Info

See also: BEGIN SEQUENCE, ErrorBlock(), FError(), HB_OsError(), TRY...CATCH
Category: Error functions
Source: vm\errorapi.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example shows how to catch a "file not found" error
// using DosError() within a TRY...CATCH error handling structure.

   PROCEDURE Main
      LOCAL oError
```

```
   DosError(55)             // sets current DOS error code

   TRY
      ? DosError()          // result: 55

      USE xyz NEW           // create DOS error

   CATCH oError

      IF DosError() == 2
         ? "File not found:", oError:fileName
         QUIT
      ENDIF
   END

RETURN
```

DosParam()

Returns the command line parameters passed to an xHarbour application.

Syntax

```
DosParam() --> cCommandLine
```

Return

The function returns a character string holding a comma separated list of command line parameters passed to an xHarbour application.

Info

See also: ExeName(), HB_ArgV(), PCount(), PValue(), Token()
Category: CT:Miscellaneous, Environment functions
Source: ct\ctmisc.prg
LIB: xhb.lib
DLL: xhbdll.dll

DoW()

Determines the numeric day of the week from a date.

Syntax

```
DoW( <dDate> ) --> nDayOfWeek
```

Arguments

<dDate>

The parameter must be a value of data type Date.

Return

The function returns the numeric day of the week or 0 for an empty date.

Description

DoW() calculates the numeric day of the week from a Date value. Weekdays are numbered from One to Seven, with Sunday being 1 and Saturday being 7.

Info

See also:	CDow(), CtoD(), Date(), Day(), DtoC(), DtoS(), SET CENTURY, SET DATE, StoD(), Transform()
Category:	Conversion functions, Date and time
Source:	rtl\dateshb.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example illustrates CDoW() and Dow() return values.

   PROCEDURE Main
      LOCAL dDate := CtoD( "01/01/2006" )

      ? CDoW( dDate )                    // result: Sunday
      ? DoW ( dDate )                    // result: 1

      ? CDoW( dDate+4 )                  // result: Thursday
      ? DoW ( dDate+4 )                  // result: 5
   RETURN
```

DoY()

Returns the day number of a Date value in a year.

Syntax

```
DoY( [<dDate>] ) --> nDayOfYear
```

Arguments

<dDate>

An expression returning a Date value. It defaults to Date().

Return

The function returns a numeric value. It is the number of the day in a year that represents *<dDate>*.

Info

See also:	Day(), DaysToMonth()
Category:	CT:DateTime, Date and time
Source:	ct\dattime2.c
LIB:	xhb.lib
DLL:	xhbdll.dll

DriveType()

Determines the type of a drive.

Syntax

```
DriveType( [<cDrive>] ) --> nDriveType
```

Arguments

<cDrive>

This parameter defaults to the current drive letter. It can be specified as a drive letter from A to Z without a colon.

Return

The function returns a numeric value indicating the type of the specified drive. The following values are possible:

Codes for drive types

Return	Drive type
0	RAM drive
2	Floppy drive
3	Hard drive
4	CD-Rom drive
5	Network drive
9	Unknown drive

Info

See also: DiskName()
Category: CT:DiskUtil, Disks and Drives
Source: ct\disk.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example lists the types of 26 possible drives and their capacity

   PROCEDURE Main
     LOCAL cDrive, i

     CLS

     FOR i:=1 TO 26
        cDrive := Chr(64+i)
        ? cDrive, DriveType( cDrive), DiskTotal( cDrive )
     NEXT
   RETURN
```

DtoC()

Converts a Date value to a character string in SET DATE format.

Syntax

```
DtoC( <dDate>)  --> cDateString
```

Arguments

<dDate>

The parameter must be a value of data type Date.

Return

The return value is a character string formatted in the current SET DATE format.

Description

The function converts a Date value to a character string. The string is formatted according to the current SET DATE format. The default date format is "MM/DD/YY".

Important: use DtoC() and its counterpart CtoD() with extreme care. The result of both functions depends on the current SET DATE and SET EPOCH format.

Info

See also:	CtoD(), Date(), DtoS(), SET CENTURY, SET DATE, SET EPOCH, StoD(), Transform(), TtoC()
Category:	Conversion functions, Date and time
Source:	rtl\dateshb.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example creates a date format independent Date value and
// outlines a common problem with date to string conversions. The
// last CtoD() call yields a date in October instead of January.

   PROCEDURE Main
      LOCAL dDate := StoD( "20060110" ) //  10th of January, 2006
      LOCAL cDate
                                        // use default date format
      ? cDate := DtoC( dDate )          // result: 01/10/06

      SET DATE ITALIAN                  // select Italian date format

      ? DtoC( dDate )                   // result: 10-01-06

      SET CENTURY ON
      SET EPOCH TO 2000

      ? DtoC( dDate )                   // result: 10-01-2006

                                        // CAUTION: date format mismatch
      ? CtoD( cDate )                   // result: 01-10-2006
   RETURN
```

DtoR()

Converts an angle from degrees to radians.

Syntax

```
DtoR( <nDegrees> ) --> nRadians
```

Arguments

<nDegrees>

A numeric value in the range of 0 to 360 degrees

Return

The numeric return value is the angle in radians.

Info

See also: RtoD()
Category: CT:Math, Conversion functions, Mathematical functions, Trigonometric functions
Source: ct\trig.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays results of function DtoR().

   PROCEDURE Main
      ? DtoR(   0 )   // result: 0.00
      ? DtoR(  45 )   // result: 0.79
      ? DtoR(  60 )   // result: 1.05
      ? DtoR(  90 )   // result: 1.57 (qtr. circle = Pi()/2)
      ? DtoR( 120 )   // result: 2.09
      ? DtoR( 180 )   // result: 3.14 (half circle = Pi())
      ? DtoR( 240 )   // result: 4.19
      ? DtoR( 300 )   // result: 5.24
      ? DtoR( 360 )   // result: 6.28 (full circle=2*Pi())
   RETURN
```

DtoS()

Converts a Date value to a character string in YYYYMMDD format.

Syntax

```
DtoS( <dDate>)  --> cYYYYMMDD
```

Arguments

<dDate>

The parameter must be a value of data type Date.

Return

The return value is a character string in YYYYMMDD format. DtoS() returns Space(8) for an empty date.

Description

This function converts a Date value to a character string that is independent from the settings SET CENTURY, SET DATE and SET EPOCH. Therefore, DtoS() and its counterpart StoD() are the recommended Date to Character string conversion functions, unless a Date value must be output in a country specific format.

When Date fields are combined with Character fields in an index expression, use DtoS() to concatenate the Date field with the Character field. This ensures that the Date field is sorted in chronological order.

Info

See also:	CtoD(), Date(), DtoC(), INDEX, StoD(), TtoS()
Category:	Conversion functions, Date and time
Source:	rtl\dateshb.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example illustrates sorting of date formatted character strings
// and how it is influenced by the date format.

   PROCEDURE Main
      LOCAL aDate := { StoD( "20060209" ), ;
                       StoD( "20060303" ), ;
                       StoD( "20060106" )  }

      SET DATE TO ITALIAN   // dd-mm-yyyy

                            // using DtoC()
      ASort( aDate,,, {|d1,d2| DtoC( d1 ) < DtoC( d2 ) } )
      AEval( aDate, {|d| QOut(Day(d), CMonth(d) ) } )
                            /* result:
                             3 March
                             6 January
                             9 February
                             */

                            // using DtoS()
      ASort( aDate,,, {|d1,d2| DtoS( d1 ) < DtoS( d2 ) } )
      AEval( aDate, {|d| QOut(Day(d), CMonth(d) ) } )
                            /* result:
```

```
                                6 January
                                9 February
                                3 March
                                */
      RETURN
```

ElapTime()

Calculates the time elapsed between a start and an end time.

Syntax

```
ElapTime( <cStartTime>, <cEndTime> ) --> cElapsedTime
```

Arguments

<cStartTime>

This is a Time() formatted character string containing the start time to use for the calculation. The time string must be coded using a 24h time format.

<cEndTime>

A time string containing the end time.

Return

The function returns a time-formatted character string holding the duration of the time interval in the format hh:mm:ss.

Description

This function returns a string that shows the difference between the starting time represented as *<cStartTime>* and the ending time as *<cEndTime>*. If the starting time is later than the ending time, the function assumes that the end time occured on the next day.

Info

See also:	Days(), Time(), Secs(), Seconds()
Category:	Date and time
Source:	rtl\samples.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows return values of ElapTime()

   PROCEDURE Main
      LOCAL cStartTime := "17:30:15"

      ? ElapTime( cStartTime, "21:45:30" )  // result: 04:15:15

      ? ElapTime( cStartTime, "07:45:30" )  // result: 14:15:15
   RETURN
```

Empty()

Checks if the passed argument is empty.

Syntax

```
Empty( <expression> ) -->  lIsEmpty
```

Arguments

<expression>

An expression of any data type whose value is tested for being empty.

Return

The function returns .T. (true) when the passed value is empty, otherwise .F. (false).

Description

The function tests if an expression yields an empty value. The status "Empty" exists for all data types, except Code block Object and Pointer. It is commonly used to test if a user has entered data into an input field. The following table list empty values:

Data types and their empty values

Data type	Valtype()	Description
Array	A	Array with zero elements
Code block	B	No empty value
Character	C	Null string and White space characters (CR/LF, Space(), Tabs)
Date	D	Null date (CTOD(""))
Hash	H	Hash with zero elements
Logical	L	False (.F.)
Memo	M	Same as character
Numeric	N	The value zero
Object	O	No empty value
Pointer	P	No empty value
NIL	U	NIL

Info

See also: Len(), Valtype()
Category: Logical functions
Source: rtl\empty.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example lists common empty values and outlines few values
// that could be considered empty, but which are not.

   PROCEDURE Main
      ? Empty( NIL )              // result: .T.

      ? Empty( 0 )                // result: .T.

      ? Empty( .F. )              // result: .T.

      ? Empty( CtoD("") )         // result: .T.
```

```
    ? Empty( "" )               // result: .T.
    ? Empty( "     " )          // result: .T.

    ? Empty( Chr( 0) )          // result: .F.
    ? Empty( Chr( 9) )          // result: .T.
    ? Empty( Chr(10) )          // result: .T.
    ? Empty( Chr(13) )          // result: .T.
    ? Empty( Chr(32) )          // result: .T.

    ? Empty( {} )               // result: .T.
    ? Empty( {0} )              // result: .F.

    ? Empty( {|| NIL } )        // result: .F.

    ? Empty( {=>} )             // result: .T.
RETURN
```

Enhanced()

Selects the enhanced color of SetColor().

Syntax

```
Enhanced() --> cNull
```

Return

The function selects the 2nd color of the SetColor() setting for standard console output and returns a null string ("").

Info

See also:	ColorSelect(), Standard(), UnSelected()
Category:	CT:Video, Screen functions
Source:	ct\color.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Eof()

Determines when the end-of-file is reached.

Syntax

```
Eof() --> lIsEndOfFile
```

Return

The function returns .T. (true) when the record pointer in a work area has reached the end of file, otherwise .F. (false).

Description

The Eof() function indicates if the record pointer has reached the logical end-of-file mark during database navigation. The end-of-file mark is a state variable that exists for each work area. Eof() operates in the current work area unless it is used within an aliased expression.

The Eof() state of a work area is evaluated each time the record pointer is moved with commands like SKIP, SEEK or LOCATE or their functional equivalents. When Eof() returns .T. (true), this value is retained until the record pointer is moved to a different record, or a new record is appended.

Note that Eof() is always .T. (true) when a database has no records, and it returns always .F. (false) when a work area is not in use.

Info

See also:	Bof(), DbSeek(), DbSkip(), DO WHILE, Found(), GO, LastRec(), LOCATE, RecNo(), SEEK, SKIP
Category:	Database functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example illustrates the Eof() state of a work area during
// physical and logical record pointer movement.

   #include "Ord.ch"

   PROCEDURE Main
      LOCAL nCount := 0

      USE Customer
      INDEX ON Upper(LastName+FirstName) TO Cust01

      ? Eof(), LastName                // result: .F. Alberts

      // physical movement of record pointer
      GOTO 200
      ? Eof(), nCount, Recno()         // result: .F.   0   200

      // logical movement of record pointer
      DO WHILE .NOT.EoF()
         SKIP
         nCount ++
      ENDDO

      // Note: from record #10 we skipped 45 records
```

```
   // to hit end-of-file, and ended up at record #220
   // That's because the database is indexed.
   ? EoF(), nCount, Recno()         // result: .T.   220   225

   ? EoF(), Empty(LastName)         // result: .T.  .T.
   SKIP 0
   ? EoF(), Empty(LastName)         // result: .T.  .T.

   SKIP -1
   ? EoF(), LastName                // result: .F.  Waters

   SET SCOPETOP    TO "G"           // restrict logical navigation
   SET SCOPEBOTTOM TO "M"           // restrict logical navigation

   GO BOTTOM
   ? EoF(), LastName                // result: .F.  Miller

   SKIP                             // EoF reached due to scope
   ? EoF(), LastName                // result: .T. (empty)

   CLOSE Customer
RETURN
```

EoM()

Returns the date of the last day in a month.

Syntax

```
EoM( [<dDate>] ) --> dLastDayOfMonth
```

Arguments

<dDate>

Any Date value, except for an empty date, can be passed. The default is the return value of Date().

Return

The function returns the last day of the month that includes *<dDate>* as a Date value, or an empty date on error.

Info

See also:	BoM(), BoQ(), BoY(), EoQ(), EoY(), LastDayoM(), Month()
Category:	CT:DateTime, Date and time
Source:	ct\datetime.c
LIB:	xhb.lib
DLL:	xhbdll.dll

EoQ()

Returns the date of the last day in a quarter.

Syntax

```
EoQ( [<dDate>] ) --> dEndOfQuarter
```

Arguments

`<dDate>`

Any Date value, except for an empty date, can be passed. The default is the return value of Date().

Return

The function returns the last day of the quarter that includes *<dDate>* as a date value, or an empty date on error.

Info

See also:	BoM(), BoQ(), BoY(), EoM(), EoY(), Quarter()
Category:	CT:DateTime, Date and time
Source:	ct\datetime.c
LIB:	xhb.lib
DLL:	xhbdll.dll

EoY()

Returns the date for the last day of a year.

Syntax

```
EoY( [<dDate>] ) --> dLastDayOfYear
```

Arguments

<dDate>

Any Date value, except for an empty date, can be passed. The default is the return value of Date().

Return

Returns the 31st of December of the year specified with *<dDate>*.

Info

See also:	BoM(), BoQ(), BoY(), EoM(), EoQ(), Year()
Category:	CT:DateTime, Date and time
Source:	ct\datetime.c
LIB:	xhb.lib
DLL:	xhbdll.dll

ErrorBlock()

Sets or retrieves the error handling code block.

Syntax

```
ErrorBlock( [<bErrorBlock>] ) --> bLastErrorBlock
```

Arguments

<bErrorBlock>

An optional code block can be passed that replaces the current error handling code block. The code block receives as parameter an Error() object containing information about a runtime error.

Return

The return value is the currently installed error handling code block.

Description

The error code block is part of the runtime error handling system of xHarbour and can be retrieved or defined with the ErrorBlock() function. A default error code block is installed at program startup in the ErrorSys() routine.

By default, the error code block is evaluated when a runtime error occurs ouside a TRY...CATCH control structure. An error object is filled with error information by the xHarbour runtime kernel and passed on to the error code block. The error code block can then call a user defined error handling routine, or pass the Error object on to the Break() function, which, in turn, passes the Error object on to the RECOVER option of the BEGIN SEQUENCE statement.

If the error code block does not call Break(), its return value must be of logical data type to instruct the error handling system how to proceed with error handling. If .T. (true) is returned, the offending operation is retried, while .F. (false) instructs the error handling system to ignore the error and resume processing.

Note: the error code block is part of xHarbour's Clipper compatible error handling system and not recommended to use in new applications. Instead, xHarbour's moder TRY...CATCH error handling control structure provides a modern and more flexible way for programming runtime error recovery.

Info

See also:	BEGIN SEQUENCE, Break(), ErrorNew(), ErrorSys(), TRY...CATCH
Category:	Error functions, Debug functions
Source:	vm\errorapi.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Examples

```
// This example uses offensive error handling by assuming that
// index files required for a database exist. If they do not
// exist, they are created in the RECOVER section of the BEGIN
// SEQUENCE statement.

   PROCEDURE Main
      OpenCustomer()

      ? "Files are open"
      ? "Program can start"
   RETURN
```

```
   PROCEDURE OpenCustomer
      LOCAL oError, n
      LOCAL bError := ErrorBlock( {|e| Break(e) } )

      FOR n := 1 TO 2
         BEGIN SEQUENCE
            // regular code
            USE Customer NEW SHARED
            SET INDEX TO Cust01, Cust02
            ErrorBlock( bError )

         RECOVER USING oError
            // error handling code
            ErrorBlock( bError )

            USE Customer NEW EXCLUSIVE
            INDEX ON CustNo TO Cust01
            INDEX ON Upper(LastName+FirstName) TO Cust02
            USE

            // go back to open DBF in SHARED mode
            LOOP
        ENDSEQUENCE

        EXIT
     NEXT

   RETURN

// This example does the same as the previous one, except that
// it uses TRY...CATCH instead of an error code block.

   PROCEDURE Main
      OpenCustomer()

      ? "Files are open"
      ? "Program can start"
   RETURN


   PROCEDURE OpenCustomer
      LOCAL n

      FOR n:=1 TO 2
         TRY
            // regular code
            USE Customer NEW SHARED
            SET INDEX TO Cust01, Cust02
         CATCH
            // error recovery
            USE Customer NEW EXCLUSIVE
            INDEX ON CustNo TO Cust01
            INDEX ON Upper(LastName+FirstName) TO Cust02
            USE
            LOOP
         END

         EXIT
      NEXT

   RETURN
```

ErrorLevel()

Sets the exit code of the xHarbour application.

Syntax

```
ErrorLevel( [<nNewExitCode>] ) --> nCurrentExitCode
```

Arguments

<nNewExitCode>

An optional numeric value in the range of 0 to 255 can be passed that replaces the current exit code of the xHarbour application.

Return

The function returns a numeric value which is the current exit code of the xHarbour application.

Description

The ErrorLevel() function defines or retrieves the exit code of the xHarbour application. The exit code can be viewed as the return value of the application that is passed to the calling process upon termination. Normally, the calling process is the operating system. When the xHarbour application ends regularly, the exit code is set to 0. If the application quits due to a runtime error, the default exit code is set to 1. The exit code can be queried in a batch file using the ERRORLEVEL command of the operating system. This way, regular and irregular program termination can be detected.

ErrorLevel() allows for user-defined exit codes since numeric values in the range of 0 to 255 can be passed to the function. A user-defined exit code is retained until a new value is passed to ErrorLevel().

Info

See also:	ErrorBlock(), ErrorSys(), Getenv(), QUIT
Category:	Error functions, Debug functions
Source:	vm\hvm.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// This example consists of three files: MYSTART.BAT, a batch processing
// file executed by the operating system, and two PRG files containing
// the code for two xHarbour applications.
// The ErrorLevel() exit code of both applications is queried in the
// batch file to run both xHarbour applications as alternating processes
// when the user presses the Y key.

   @ECHO OFF
   REM This is the command file MYSTART.BAT
   :Start
   ECHO Main application module
   MyProgram.exe

   IF ERRORLEVEL 25 GOTO Reorg
   GOTO Exit

   :Reorg
   ECHO Data reorganization module
   Reorganize.exe
   IF ERRORLEVEL 26 GOTO Start
```

```
:Exit
ECHO Application has ended
REM *EOF*

/*
 * Program: MyProgram.exe
 */
PROCEDURE Main
   ? "press <Y> to re-organize data"
   ?
   Inkey(0)

   IF Chr( LastKey() ) $ "yY"
      ErrorLevel( 25 )
   ENDIF
RETURN


/*
 * Program: Reorganize.exe
 */
PROCEDURE Main
   ? "Data re-organization is running."
   ?
   ? "press <Y> to re-start main program"
   ?
   Inkey(0)

   IF Chr( LastKey() ) $ "yY"
      ErrorLevel( 26 )
   ENDIF
RETURN
```

ErrorNew()

Creates a new Error object and optionally fills it with data.

Syntax

```
ErrorNew( [<cSubSystem>  ], ;
          [<nGenCode>    ], ;
          [<nSubCode>    ], ;
          [<cOperation>  ], ;
          [<cDescription>], ;
          {<aArgs>       ], ;
          [<cModuleName> ], ;
          [<cProcName>   ], ;
          [<nProcLine>   ]  ) --> oError
```

Arguments

<cSubSystem>

A character string that is assigned to oError:subSystem. Defaults to NIL.

<nGenCode>

A numeric value that is assigned to oError:genCode. Defaults to NIL.

<nSubCode>

A numeric value that is assigned to oError:subCode. Defaults to NIL.

<cOperation>

A character string that is assigned to oError:operation. Defaults to NIL.

<cDescription>

A character string that is assigned to oError:description. Defaults to NIL.

<aArgs>

An array that is assigned to oError:args. Defaults to NIL.

<ModuleName>

A character string that is assigned to oError:modulName. Defaults to ProcFile(1).

<cProcName>

A character string that is assigned to oError:procName. Defaults to ProcName(1).

<nProcLine>

A numeric value that is assigned to oError:procLine. Defaults to ProcLine(1).

Return

The function returns the new Error object with the passed parameters assigned to instance variables.

Description

ErrorNew() creates a new Error object and optionally assigns the parameters passed to instance variables of the object. The function is automatically called by the xHarbour runtime system when a runtime error occurs. User defined routines can create their own Error objects and fill them with error information. The Error object is then passed to the error code block (see ErrorBlock()) or to the Throw() function.

Error objects are instances of the Error() class. Refer to that class for more information on creating user defined Error objects.

Info

See also: Error(), ErrorBlock(), Throw(), TRY...CATCH
Category: Error functions, Object functions
Header: error.ch
Source: rtl\terror.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// refer to the Error() class for an Error object example
```

ErrorSys()

Installs the default error code block.

Syntax

```
ErrorSys( <bDefaultErrorBlock> ) --> NIL
```

Arguments

<bDefaultErrorBlock>

A code block accepting one parameter must be passed. The code block must call the default error handling routine and pass the received parameter on to it.

Return

The return value is always NIL.

Description

ErrorSys() is an implicit INIT PROCEDURE called at startup of an xHarbour application. It installs the default error code block used for standard error handling. ErrorSys() is programmed in the ERRORSYS.PRG module and should not be called at runtime of an xHarbour application. To replace the default error code block, call the ErrorBlock() function.

Since ErrorSys() is implicitely called it must be present in an xHarbour application. The only reason to program a user defined ErrorSys() routine is when the entire ERRORSYS.PRG module is replaced by a user-defined module.

Info

See also:	BEGIN SEQUENCE, Break(), Error(), ErrorBlock(), ErrorLevel(), Throw(), TRY...CATCH
Category:	Error functions
Header:	error.ch
Source:	rtl\ errorsys.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Eval()

Evaluates a code block.

Syntax

```
Eval( <bBlock> [, <xValue,...> ] )   --> xReturn
```

Arguments

<bBlock>

A code block to be evaluated

<xValue>

Any number of values to be passed as arguments to the code block.

Return

The function returns the value of the last expression inside the code block.

Description

This function evaluates the code block *<bBlock>* and passes on to it all parameters specified as the comma separated list *<xValues,...>*. A code block can contain multiple, comma separated expressions that are executed from left to right. The value of the last expression executed in the code block is used as return value of Eval().

Code blocks are values that contain executable code. Normally, the xHarbour compiler creates the code associated with code blocks at compile time. It is possible, however, to create code blocks at runtime of an application. This is accomplished by the macro operator (&) which compiles a character string holding the syntax for a code block at runtime.

Info

See also:	AEval(), AScan(), ASort(), DbEval(), FieldBlock()
Category:	Code block functions, Indirect execution
Source:	vm\evalhb.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example outlines basic rules for code block usage.

   PROCEDURE Main
      LOCAL bBlock

      bBlock := {|x| QOut(x) }

      Eval( bBlock, Date() )           //   result: NIL (of QOut())
                                       // displays: 02/08/06
      bBlock := {|x,y| x + y }

      ? Eval( bBlock, 5, 17 )          // result: 22

      ? Eval( bBlock, "A", "string" )  // result: Astring
   RETURN
```

ExeName()

Returns the EXE file name of an xHarbour application.

Syntax

```
ExeName() --> cExeFileName
```

Return

The function returns a character string holding the full qualified file name of the EXE.

Info

See also:	DosParam(), HB_CmdArgArgV()
Category:	CT:Miscellaneous, Environment functions
Source:	ct\ctmisc.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Exp()

Calculates the value of e raised by an exponent.

Syntax

```
Exp( <nExponent> ) --> nAntiLogarithm
```

Arguments

`<nExponent>`

A numeric expression used as exponent for e (2.71828...)

Return

The function returns the numeric anti-logarithm of *<nExponent>*.

Description

This function returns the value of e raised to the power of *<nExponent>*. It is the inverse function of Log(). Note that *<nExponent>* cannot be larger than 46. Otherwise, a numeric overflow is created.

Info

See also: Log(), SET DECIMALS, SET FIXED
Category: Mathematical functions, Numeric functions
Source: rtl\math.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example shows return values of Exp()

   PROCEDURE Main

      SET DECIMALS TO 15

      ? Exp(1)        // result:          2.718281828459045
      ? Exp(-1)       // result:          0.367879441171442
      ? Exp(46)       // result:  94961194206024470000.000000000000000
      ? Exp(46.1)     // result:  ************************************
   RETURN
```

Expand()

Inserts characters between all characters in a string.

Syntax

```
Expand( <cString>, ;
       [<nCount>], ;
       [<xChar>]   ) --> cResult
```

Arguments

`<cString>`

This is the character string to process.

`<nCount>`

Optionally, the number of characters to insert between each character can be specified. The default value is 1.

`<xChar>`

This is a single character or its numeric ASCII code that is inserted <*nCount*> times between each character of <*cString*>. It defaults to a blank space (Chr(32)).

Return

The function iterates <*cString*>, inserts the character sequence Replicate(<*xChar*>, <*nCount*>) between characters of <*cString*>, and returns the expanded result.

Info

See also:	CharSpread(), PadLeft(), PadRight()
Category:	CT:String manipulation, Character functions
Source:	ct\spread.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays results of Expand()

   PROCEDURE Main
      LOCAL cStr1 := "xHarbour"
      LOCAL cStr2 := "ABCD"

      ? Expand( cStr1 )           // result: x H a r b o u r
      ? Expand( cStr2, 3, "." )   // result: A...B...C...D
   RETURN
```

Exponent()

Calculates the exponent of a floating point number.

Syntax

```
Exponent( <nFloat> ) --> nExponent
```

Arguments

<nFloat>

Any numeric value can be passed.

Return

The function returns the exponent base 2 of a floating point number. It is used together with the Mantissa() for the binary representation of floating point numbers.

Info

See also: Mantissa()
Category: CT:NumBits, Numbers and Bits
Source: ct\exponent.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays results of Exponent() and shows
// how a number is calculated from its mantissa and exponent.

   PROCEDURE Main
      LOCAL nE, nM

      ? Exponent( -10 )       // result:    3
      ? Exponent(  -1 )       // result:    0
      ? Exponent(-0.1 )       // result:   -4
      ? Exponent(   0 )       // result:    0
      ? Exponent( 0.1 )       // result:   -4
      ? Exponent(   1 )       // result:    0
      ? Exponent(  10 )       // result:    3

      nE := Exponent( 10 )
      nM := Mantissa( 10 )
      ? nM * 2^nE             // result:   10.0000
   RETURN
```

Fact()

Calculates the factorial of a number.

Syntax

```
Fact( <nInteger> ) --> nFactorial
```

Arguments

<nInteger>

A numeric integer value in the range of 0 to 20. If the parameter has a decimal fraction, it is truncated.

Return

The function returns the factorial of *<nInteger>*, or -1 when the value is out of range.

Info

See also: Ceiling(), Floor()
Category: CT:Math, Mathematical functions
Source: ct\ctmath2.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays results of function Fact().

   PROCEDURE Main
      LOCAL i

      ? Fact(-3)            // result:  -1 (error value)

      ? Fact(3)             // result:   6

      ? Fact(5.5)           // result: 120 (factorial of 5)

      FOR I:=1 TO 40
         IF Fact(i) < 0
            ? i-1           // result: 20 (max. value)
            EXIT
         ENDIF
      NEXT

   RETURN
```

Fahrenheit()

Converts degrees Celsius to Fahrenheit.

Syntax

```
Fahrenheit( <nCelsius> ) --> nFahrenheit
```

Arguments

<nCelsius>

This is a numeric value specifying degrees Celsius.

Return

The function returns degrees Fahrenheit as a numeric value.

Info

See also: Celsius()
Category: CT:NumBits, Numbers and Bits
Source: ct\num1.c
LIB: xhb.lib
DLL: xhbdll.dll

FCharCount()

Counts the number of characters in a text file ignoring white space characters.

Syntax

```
FCharCount( <cFileName> ) --> nCharCount
```

Arguments

<cFileName>

This is a character string holding the name of the text file whose characters to count. It must include path and file extension. If the path is omitted from *<cFileName>*, the file is searched in the current directory.

Return

The function returns the number of non white space characters contained in a text file.

Description

The function counts the characters in an ASCII text file but ignores white space characters. These are Tab (Chr(9)), Line Feed (Chr(10)), Carriage return (Chr(13)) and Blank space (Chr(32)).

Info

See also:	FCreate(), FOpen(), FLineCount(), FParse(), FWordCount(), MemoLine(), MlCount()
Category:	File functions, xHarbour extensions
Source:	rtl\fparse.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays the number of characters of
// the example.

   PROCEDURE Main
      LOCAL cFileName := "FCharCount.prg"

      ? FCharCount( cFileName )        // result: 141

   RETURN
```

FClose()

Closes a binary file.

Syntax

```
FClose( <nFileHandle> ) --> lSuccess
```

Arguments

<nFileHandle>

The numeric file handle returned from a previous call to FOpen() or FCreate().

Return

The return value is .T. (true) when the file is successfully closed, otherwise .F. (false).

Description

FClose() closes a file that is either newly created with FCreate() or opened with FOpen(). Both functions return a numeric file handle that must be passed to FClose() in order to close the open file. When the file is successfuly closed, all internal file buffers are permanently written to disk, and the function returns .T. (true).

If the close operation fails, the return value is .F. (false). The reason for failure can then be queried by calling the FError() function.

Info

See also: FCreate(), FError(), FOpen(), FRead(), FSeek(), FWrite()
Category: File functions, Low level file functions
Source: rtl\philes.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates how to create a new file, write
// some data to it and close it. If any operation fails, an
// appropriate error message is displayed along with the error
// code obtained from the operating system.

   #include "Fileio.ch"

   PROCEDURE Main
      LOCAL nFileHandle, cText

      nFileHandle := FCreate( "NewFile.txt", FC_NORMAL )

      IF FError() <> 0
         ? "Error while creatingg a file: ", FError()
         QUIT
      ENDIF

      cText  := "New text for file"
      IF FWrite( nFileHandle, cText ) <> Len( cText )
         ? "Error while writing a file: ", FError()
      ENDIF

      IF .NOT. FClose( nFileHandle )
         ? "Error while closing a file: ", FError()
```

```
   ELSE
      ? "File successfully created and closed"
   ENDIF
RETURN
```

FCount()

Returns the number of fields in the current work area.

Syntax

```
FCount() --> nFieldCount
```

Return

The function returns a numeric value which is the number of field variables available in a work area. If a work area is not used, the return value is zero.

Description

FCount() is used to determine how many fields, or field variables, exist in a work area. By default, the function operates in the current work area. Use an aliased expression to determine the number of fields in a different work area.

Info

See also:	FIELD, FieldGet(), FieldName(), FieldType(), Type()
Category:	File functions, Field functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows a typical programming pattern for FCount() where
// an array is filled with the names and contents of all fields.

   PROCEDURE Main
      LOCAL aArray, n

      USE Customer NEW
      aArray := Array( FCount() )

      FOR n := 1 TO FCount()
         aArray[n] := { FieldName(n), FieldGet(n) }
      NEXT

      AEval( aArray, {|a| QOut( Padr(a[1],10)+":",a[2] ) } )

      USE
   RETURN
```

FCreate()

Creates an empty binary file.

Syntax

```
FCreate( <cFileName>, [<nFileAttr>] ) --> nFileHandle
```

Arguments

`<cFilename>`

This is a character string holding the name of the file to create. It must include path and file extension. If the path is omitted from *<cFileName>*, the file is created in the current directory.

`<nFileAttr>`

A numeric value specifying one or more attributes associated with the new file. #define constants from the FILEIO.CH file can be used for *<nFileAttr>* as listed in the table below:

Attributes for binary file creation

Constant	Value	Attribute	Description
FC_NORMAL *)	0	Normal	Creates a normal read/write file
FC_READONLY	1	Read-only	Creates a read-only file
FC_HIDDEN	2	Hidden	Creates a hidden file
FC_SYSTEM	4	System	Creates a system file

**) default attribute*

Return

The function returns a numeric file handle to be used later for writing data to the file using FWrite(). The return value is -1 when the operation fails. Use FError() to determine the cause of failure.

Description

The low-level file function FCreate() creates a new file, or truncates an existing file to contain zero bytes. If the operation is successful, the return value must be preserved in a variable for accessing the file lateron. The file is left open in compatibility sharing mode and read/write access mode so that the creating application can write data to the file. The file attribute *<nFileAttr>* is set when the file is closed again. Any attribute restricting file access, like FC_READONLY, for example, becomes effective after the newly created file is closed.

Info

See also:	FClose(), FErase(), FError(), FOpen(), FRead(), FSeek(), FWrite(), HB_FCommit(), HB_FCreate(), HB_FTempCreate()
Category:	File functions, Low level file functions
Header:	FileIO.ch
Source:	rtl\philes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example creates a new file, writes some data to it and closes it.
// If file creation or data write fails, an error message is displayed
// along with the error code obtained from the operating system.

   #include "Fileio.ch"
```

```
PROCEDURE Main
   LOCAL nFileHandle, cText

   nFileHandle := FCreate( "NewFile.txt", FC_NORMAL )

   IF FError() <> 0
      ? "Error while creatingg a file:", FError()
      QUIT
   ENDIF

   cText := "Text for new file"

   IF FWrite( nFileHandle, cText ) <> Len( cText )
      ? "Error while writing a file:", FError()
   ENDIF

   IF .NOT. FClose( nFileHandle )
      ? "Error while closing a file:", FError()
   ELSE
      ? "File successfully created and closed"
   ENDIF

RETURN
```

FErase()

Deletes a file from disk.

Syntax

```
FErase( <cFileName> ) --> nSuccess
```

Arguments

<cFilename>

This is a character string holding the name of the file to delete. It must include path and file extension. If the path is omitted from *<cFileName>*, the file is searched and deleted in the current directory.

Return

The return value is 0 when the file is successfully deleted, or -1 on failure. Use function FError() to determine the cause of failure.

Description

The low-level file function FErase() deletes a file from disk. The file must be closed before it can be deleted.

Note that *<cFileName>* must include the entire path and file name. If no path is specified, the function searches only the current directory to delete the file. The setting SET DEFAULT and SET PATH are ignored by FErase().

Info

See also:	CLOSE, ERASE, FClose(), FCreate(), FError(), FOpen(), FRead(), FRename(), FSeek(), FWrite(), RENAME
Category:	File functions, Low level file functions
Source:	rtl\philes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example deletes all CDX index files found in the current
// directory.

   #include "Directry.ch"

   PROCEDURE Main
      LOCAL aCdxFiles := Directory( "*.CDX" )

      AEval( aCdxFiles, { |aFile| FErase( aFile[ F_NAME ] ) } )
   RETURN
```

FError()

Retrieves the error code of the last low-level file operation.

Syntax

```
FError() --> nErrorCode
```

Return

The function returns the error code of the last low-level file operation as a numeric value. If the last low-level file operation is successful, the return value is zero.

Description

Low-level file operations work with file handles and include the functions FClose(), FCreate() , FErase(), FOpen(), FRead(), FReadStr(), FRename(), FSeek() and FWrite(). If any file operation performed by these functions fails, the error code obtained from the disk operating system is returned from function FError(). The function retains this value until a successful low-level file operation completes.

Note: error codes for low-level file functions are a subset of error codes listed under DosError(). See DosError() for a description of the meaning of error codes.

Info

See also:	DosError(), FClose(), FCreate(), FErase(), FOpen(), FRead(), FReadStr(), FRename(), FSeek(), FWrite(), HB_OsError()
Category:	File functions, Low level file functions
Source:	rtl\philes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// See the example for FCreate()
```

FieldBlock()

Creates a set/get code block for a field variable

Syntax

```
FieldBlock( <cFieldName> ) --> bFieldBlock
```

Arguments

`<cFieldName>`

A character string holding the name of the database field to be accessed by the returned code block.

Return

The function returns a code block accessing the field variable *<cFieldName>*, or NIL when an illegal parameter is passed. The code block accepts one parameter used to assign a value to the field variable.

Description

The code block function FieldBlock() creates a code block which accesses a field variable in the current work area. When evaluated with no parameter, the code block retrieves (gets) the value of *<cFieldName>*. If one parameter is passed, the corresponding value is assigned (is set) to the field variable.

It is not necessary that a database is open when FieldBlock() is called. When the returned code block is passed to the Eval() function, however, the field variable *<cFieldName>* must exist in the current work area.

Note: The FieldBlock() code block can be evaluated in any work area that has the specified field variable, as long as the work area is current. That is, the code block is not bound to any particular work area but is evaluated always in the context of the current work area.

Use function FieldWBlock() to create a set/get code block bound to a particular work area.

Info

See also:	FieldGet(), FieldPut(), FieldWBlock(), MemvarBlock()
Category:	Code block functions, Database functions
Source:	rtl\fieldbl.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example creates a set/get block for the field variable
// LASTNAME in the CUSTOMER database. The field variable is
// first get and then set.

   PROCEDURE Main
      LOCAL bBlock := FieldBlock( "LastName" )

      USE Customer

      // get field variable
      ? Eval( bBlock )                // result: Miller

      // set field variable
      EVal( bBlock, "MILLER" )
      ? LastName                      // result: MILLER
```

```
   USE

   // unused work area
   ? Eval( bBlock )                 // result: runtime error
RETURN
```

FieldDec()

Retrieves the number of decimal places of a field variable.

Syntax

```
FieldDec( <nFieldPos> ) --> nDecimalPlaces
```

Arguments

<nFieldPos>

A numeric value specifying the ordinal position of the field variable to query. Valid values are in the range from 1 to FCount().

Return

The function returns a numeric value indicating the number of decimal places of the specified field variable.

Description

FieldDec() retrieves a particular information available for field variables: the number of decimal places. This is only relevant for field variables of Numeric data type. All other field variables have no decimal places.

Info

See also:	DbFieldInfo(), DbStruct(), FieldLen(), FieldName(), FieldType()
Category:	Field functions, xHarbour extensions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

FieldDeci()

Returns the decimal places of a database field.

Syntax

```
FieldDeci( <nFieldPos> ) --> nDecimals
```

Arguments

<nFieldPos>

A numeric value specifying the ordinal position of a database field.

Return

The function is a synonym for FieldDec() and returns the decimal places of a database field as a numeric value.

Info

See also:	FieldDec(), FieldLen(), FieldName(), FieldType()
Category:	CT:Database, Field functions
Source:	ct\dbftools.c
LIB:	xhb.lib
DLL:	xhbdll.dll

FieldGet()

Retrieves the value of a field variable by its ordinal position.

Syntax

```
FieldGet( <nFieldPos> ) --> xFieldValue
```

Arguments

<nFieldPos>

A numeric value specifying the ordinal position of the field variable to query. Valid values are in the range from 1 to FCount().

Return

The function returns the value of the field variable at the specified position, or NIL if *<nFieldPos>* does not fall into the valid range.

Description

FieldGet() retrieves the value of a field variable by its ordinal position, rather than by its field name. The ordinal position is determined by the sequence of field declarations when a database file is created. Ordinal positions begin with 1 and end with FCount().

Info

See also:	FieldBlock(), FieldName(), FieldPut(), FieldWBlock()
Category:	Database functions, Field functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example calls FieldGet() in the current work area
// and in a different work area using an aliased expression.

   PROCEDURE Main

      USE Customer ALIAS Cust
                                 // current work area
      ? FieldPos( "Lastname" )   // result: 3
      ? FieldGet(3)              // result: Miller

      SELECT 0                   // new work area
      ? Used()                   // result: .F.
      ? FieldGet(3)              // result: NIL

                                 // aliased expression
      ? Cust->(FieldGet(3))      // result: Miller

      CLOSE Cust
   RETURN
```

FieldLen()

Retrieves the number of bytes occupied by a field variable.

Syntax

```
FieldLen( <nFieldPos> ) --> nFieldLength
```

Arguments

<nFieldPos>

A numeric value specifying the ordinal position of the field variable to query. Valid values are in the range from 1 to FCount().

Return

The function returns a numeric value indicating the number of bytes the specified field variable occupies in one record of the database file.

Description

FieldLen() retrieves a particular information available for field variables: the number of bytes required to store the field variable in a database record. Note that the value of memo fields is stored in a separate file. A memo field variable occupies 4 or 10 bytes in a database record, depending on the RDD used.

Info

See also:	DbFieldInfo(), DbStruct(), FieldDec(), FieldName(), FieldType()
Category:	Field functions, xHarbour extensions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

FieldName()

Retrieves the name of a field variable by its ordinal position.

Syntax

```
FieldName( <nFieldPos> ) --> cFieldName
```

Arguments

<nFieldPos>

A numeric value specifying the ordinal position of the field variable to query. Valid values are in the range from 1 to FCount().

Return

The function returns the name of the specified field variable as a character string, or a null string ("") if *<nFieldPos>* does not fall into the valid range.

Description

FieldName() retrieves the name of a field variable by its ordinal position. The ordinal position is determined by the sequence of field declarations when a database file is created. Ordinal positions begin with 1 and end with FCount().

Info

See also:	DbFieldInfo(), DbStruct(), FCount(), FieldDec(), FieldGet(), FieldLen(), FieldPos(), FieldPut(), FieldType(), Type(), Valtype()
Category:	Database functions, Field functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example lists the names and values of all field variables
// in the current work area.

   PROCEDURE Main
      LOCAL n
      USE Customer ALIAS Cust

      FOR n:=1 TO FCount()
         ? Padr(FieldName(n),10),":",FieldGet(n)
      NEXT

      CLOSE Cust
   RETURN
```

FieldNum()

Returns the ordinal position of a field in a database.

Syntax

```
FieldNum( <cFieldName> ) --> nFieldPos
```

Arguments

<cFieldName>

A character string holding the name of a database field.

Return

The function is a synonym for FieldPos() and returns the ordinal position of a database field as a numeric value.

Description

Info

See also: FieldPos()
Category: CT:Database, Field functions
Source: ct\dbftools.c
LIB: xhb.lib
DLL: xhbdll.dll

FieldPos()

Retrieves the ordinal position of a field variable by its name.

Syntax

```
FieldPos( <cFieldName> ) --> nFieldPos
```

Arguments

<cFieldName>

A character string holding the name of the field variable.

Return

The function returns a numeric value indicating the ordinal position of the field variable *<cFieldName>* in the work area. If the work area has no field variable with this name, the return value is zero.

Description

FieldPos() accepts a field name as character string and retrieves from it the ordinal position of the field variable. This is the reverse functionality of function FieldName(). Both FieldPos() and FieldName() are used for programming generic database routines that work with unknown database structures.

Info

See also:	DbFieldInfo(), FCount(), FieldDec(), FieldGet(), FieldLen(), FieldName(), FieldPut(), FieldType()
Category:	Database functions, Field functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates a typical use for FieldPos().

   PROCEDURE Main
      LOCAL cName

      USE Customer NEW

      ? cLast := FieldGet(FieldPos("LASTNAME")) // result: Miller

      ? FieldPos("LASTNAME")                    // result: 3

      USE
   RETURN
```

FieldPut()

Assigns a value to a field variable by its ordinal position.

Syntax

```
FieldPut( <nFieldPos>, <xValue> ) --> xAssigneValue
```

Arguments

`<nFieldPos>`

A numeric value specifying the ordinal position of the field variable to assign a value to. *<nFieldPos>* must be in the range from 1 to FCount().

`<xValue>`

An expression whose value is assigned to the field variable. The data type of *<xValue>* must match the data type of the field variable. Note that memo fields must be assigned character strings.

Return

The function returns the value assigned to the specified field variable, or NIL if *<nFieldPos>* is outside the valid range.

Description

FieldPut() assigns a value to a field variable by using its ordinal position, rather than its field name. The ordinal position is determined by the sequence of field declarations when a database file is created. Ordinal positions begin with 1 and end with FCount().

Info

See also:	DbFieldInfo(), FieldDec(), FieldGet(), FieldLen(), FieldName(), FieldPos(), FieldType(), REPLACE
Category:	Database functions, Field functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates various possibilities of
// assigning a value to a field variable.

   PROCEDURE Main
      LOCAL cField := "LASTNAME" , cLastName := "Miller"

      USE Customer NEW ALIAS Cust

      Cust->LASTNAME := cLastName       // hard-coded assignment

      ? FieldPos( cField )              // result: 3

      ? FieldPut( 3, cLastName )        // result: Miller

                                        // aliased expression
      Cust-> ( FieldPut( FieldPos(cField), cLastName ) )

      USE
   RETURN
```

FieldSize()

Returns the length of a database field.

Syntax

```
FieldSize( <nFieldPos> ) --> nFieldLength
```

Arguments

`<nFieldPos>`

A numeric value specifying the ordinal position of a database field.

Return

The function is a synonym for FieldLen() and returns the length of a database field as a numeric value.

Info

See also:	FieldDec(), FieldLen(), FieldName(), FieldType()
Category:	CT:Database, Field functions
Source:	ct\dbftools.c
LIB:	xhb.lib
DLL:	xhbdll.dll

FieldType()

Retrieves the data type of a field variable.

Syntax

```
FieldType( <nFieldPos> ) --> cFieldType
```

Arguments

<nFieldPos>

A numeric value specifying the ordinal position of the field variable to query. Valid values are in the range from 1 to FCount().

Return

The function returns a single character indicating the data type of the specified field variable, or a null string ("") if *<nFieldPos>* is outside the valid range.

Description

FieldType() retrieves a particular information available for field variables: its data type.

Info

See also:	DbFieldInfo(), DbStruct(), FieldDec(), FieldLen(), FieldName()
Category:	Field functions, xHarbour extensions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

FieldWBlock()

Creates a set/get code block for a field variable in a particular work area.

Syntax

```
FieldWBlock( <cFieldName>, <nWorkArea> ) --> bFieldWBlock
```

Arguments

<cFieldName>

A character string holding the name of the database field to be accessed by the returned code block.

<nWorkArea>

<nWorkArea> is the numeric ordinal position of a work area to bind the code block to. Work areas are numbered from 1 to 65535.

Return

The function returns a code block accessing the field variable *<cFieldName>* in the work area *<nWorkArea>*, or NIL when an illegal parameter is passed. The code block accepts one parameter used to assign a value to the field variable.

Description

The code block function FieldWBlock() creates a code block which accesses a field variable in the work area specified with *<nWorkArea>*. By this, the code block is bound to the work area. When evaluated with no parameter, the code block retrieves (gets) the value of *<cFieldName>* in the particular work area. If one parameter is passed, the corresponding value is assigned (is set) to the field variable.

It is not necessary that a database is open when FieldWBlock() is called. When the returned code block is passed to the Eval() function, however, the field variable *<cFieldName>* must exist in the work area *<nWorkArea>*.

Info

See also:	FieldBlock(), FieldGet(), FieldPut(), MemvarBlock()
Category:	Code block functions, Database functions
Source:	rtl\fieldbl.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example creates a set/get code block accessing work area #10.
// A database is opened in this work area, and an unused work area
// is selected as current to demonstrate that the code block is bound
// to work area #10.

   PROCEDURE Main1
      LOCAL bBlock := FieldWBlock( "LastName", 10 )

      SELECT 10
      USE Customer ALIAS Cust

      SELECT 0
      ? Used()                      // result: .F.
```

```
   // get field variable
   ? Eval( bBlock )              // result: Miller

   // set field variable
   EVal( bBlock, "Smith" )
   ? Cust->LastName              // result: Smith

   ? Eval( bBlock )              // result: Smith

   CLOSE ALL
RETURN
```

File()

Checks for the existence of a file in the default directory or path

Syntax

```
File( <cFileSpec> ) --> lExists
```

Arguments

<cFileSpec>

This is a character string containing the file specification to search for. It can include the wildcard characters (* and ?) and/or drive and directory information. If a complete file name isgiven, it must include a file extension.

Return

The function returns .T. (true) if a match is found for *<cFileSpec>*, otherwise .F. (false).

Description

The File() function is used to check if a file exists that matches the file specification *<cFileSpec>*. When *<cFileSpec>* does not contain directory information, the function searches directories in the following order:

1. the current directory.
2. the directory set with SET DEFAULT.
3. the directories listed in the SET PATH setting.

Directories defined with environment variables of the operating system are not searched for *<cFileSpec>*. In addition, hidden or system files are not recognized by File().

Info

See also:	CurDir(), Directory(), IsDirectory(), SET DEFAULT, SET PATH
Category:	File functions
Source:	rtl\filehb.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows the dependency between File() and the
// SET PATH and SET DEFAULT settings.

   PROCEDURE Main

      ? File( "Customer.dbf" )                 // result: .F.
      ? File( "\xhb\apps\data\Customerdbf")    // result: .T.

      SET PATH TO \xhb\apps\data

      ? File( "Customer.dbf" )                 // result: .T.

      SET PATH TO
      SET DEFAULT TO \xhb\apps\data

      ? File( "Customer.dbf" )                 // result: .T.
      ? File( "*.cdx")                         // result: .T.
```

```
RETURN
```

FileAppend()

Concatenates two files.

Syntax

```
FileAppend( <cSourceFile>, <cTargetFile> ) --> nBytesAdded
```

Arguments

`<cSourceFile>`

This is a character string containing the name of the file to append to the target file. It must be specified with a file extension. If *<cSourceFile>* contains no drive and/or directory information, the current directory is used. SET DEFAULT and SET PATH settings are ignored.

`<cTargetFile>`

This is a character string holding the name of the target file to append the contents of *<cSourceFile>* to. The file name may include drive and directory and must include an extension. The file is created if it does not exist. Otherwise, the function attempts to open the file for write access. When this fails, the source of error can be determined with FError().

Return

The function returns the number of bytes appended to *<cTargetFile>*. This is equivalent to the file size of *<cSourceFile>*. If the operation fails, the return value is zero.

Info

See also:	CSetSafety(), FileCopy(), FileSeek()
Category:	CT:DiskUtil, File functions, Low level file functions
Source:	ct\fcopy.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

FileAttr()

Returns the attributes of a file.

Syntax

```
FileAttr( [<cFileName>] ) --> nAttributes
```

Arguments

<cFileName>

This is an optional character string containing the name of the file to query. It defaults to the last file found with FileSeek(). If specified, it must contain a file extension. If *<cFileName>* contains no drive and/or directory information, the current directory is used. SET DEFAULT and SET PATH settings are ignored.

Return

The function returns the attributes of a file as a numeric value, or -1 on error. See the example for decoding the return value.

Info

See also:	FileDate(), FileSeek(), FileSize(), FileTime(), SetFAttr()
Category:	CT:DiskUtil, File functions, Low level file functions
Source:	ct\files.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example lists #define constants that can be used for
// decoding the return value of FileAttr()

   #define FA_NORMAL           0
   #define FA_RDONLY           1   /* R */
   #define FA_HIDDEN           2   /* H */
   #define FA_SYSTEM           4   /* S */
   #define FA_LABEL            8   /* V */
   #define FA_DIREC           16   /* D */
   #define FA_ARCH            32   /* A */
   #define FA_DEVICE          64   /* I */
   #define FA_TEMPORARY      256   /* T */
   #define FA_SPARSE         512   /* P */
   #define FA_REPARSE       1024   /* L */
   #define FA_COMPRESSED    2048   /* C */
   #define FA_OFFLINE       4096   /* O */
   #define FA_NOTINDEXED    8192   /* X */
   #define FA_ENCRYPTED    16384   /* E */
   #define FA_VOLCOMP      32768   /* M */


   PROCEDURE Main
      LOCAL aFiles := Directory( "\xhb\lib\*.*" )
      LOCAL aFile

      FOR EACH aFile IN aFiles
         ShowAttr( "\xhb\lib\" + aFile[1] )
      NEXT
   RETURN
```

```
PROCEDURE ShowAttr( cFile )
   LOCAL nAttr := FileAttr( cFile )

   IF ( nAttr == -1 )
      ? "File not found ", cFile
      RETURN
   ENDIF

   ? "Attrib: "

   IF ( nAttr & FA_RDONLY ) > 0
      ?? "Readonly "
   ENDIF

   IF ( nAttr & FA_HIDDEN ) > 0
      ?? "Hidden "
   ENDIF

   IF ( nAttr & FA_SYSTEM ) > 0
      ?? "System "
   ENDIF

   IF ( nAttr & FA_DIREC ) > 0
      ?? "Directory "
   ENDIF

   IF ( nAttr & FA_ARCH ) > 0
      ?? "Archive "
   ENDIF

   IF ( nAttr & FA_NORMAL ) > 0
      ?? "Normal "
   ENDIF

   IF ( nAttr & FA_TEMPORARY ) > 0
      ?? "Temp "
   ENDIF

RETURN
```

FileCClose()

Closes the file opened in backup mode with FileCopy().

Syntax

```
FileCClose() --> lSuccess
```

Return

The return value is .T. (true) when the file is successfully closed, otherwise .F. (false) is returned.

Description

The function is used to close the file opened in backup mode of FileCopy().

Info

See also:	FileCCont(), FileCOpen(), FileCopy()
Category:	CT:DiskUtil, File functions, Low level file functions
Source:	ct\fcopy.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

FileCCont()

Continues file copying in backup mode of FileCopy().

Syntax

```
FileCCont( <cFileName> ) --> nBytes
```

Arguments

<cFileName>

This is a character string containing the name of the file to receive the next stream of data from the file opened in backup mode of FileCopy(). It must be specified with a file extension. If *<cFileName>* contains no drive and/or directory information, the current directory is used. SET DEFAULT and SET PATH settings are ignored.

Return

The function returns the number of bytes copied to *<cFileName>*

Info

See also:	CSetSafety(), FileCClose(), FileCOpen(), FileCopy()
Category:	CT:DiskUtil, File functions, Low level file functions
Source:	ct\fcopy.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

FileCOpen()

Tests if the file opened in backup mode with FileCopy() is still open.

Syntax

```
FileCOpen() --> lFileIsOpen
```

Return

The function returns .T. (true) if the source file opened in backup mode of FileCopy() is still open, otherwise .F. (false).

Info

See also:	FileCClose(), FileCCont(), FileCopy()
Category:	CT:DiskUtil, File functions, Low level file functions
Source:	ct\fcopy.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

FileCopy()

Copies files normally or in backup mode.

Syntax

```
FileCopy( <cSourceFile>, ;
          <cTargetFile>, ;
         [<lBackup>]     ) --> nBytesCopied
```

Arguments

`<cSourceFile>`

This is a character string containing the name of the file to copy to the target file. It must be specified with a file extension. If *<cSourceFile>* contains no drive and/or directory information, the current directory is used. SET DEFAULT and SET PATH settings are ignored.

`<cTargetFile>`

This is a character string holding the name of the target file to create. The file name may include drive and directory and must include an extension. If the file exists already, it is overwritten.

`<lBackup>`

This parameter defaults to .F. (false). When set to .T. (true) the function operates in backup mode. This allows for copying large files to several diskettes.

Return

The funtion returns the number of bytes copied from *<cSourceFile>* to *<cTargetFile>*.

Description

The FileCopy() function copies a source file to a target file. The *<lBackup>* parameter can be used to copy large files to several diskettes. If *<lBackup>* set to .T. (true), FileCopy() copies the maximum possible number of bytes to the target file (diskette). The function FileCOpen() is then used to check if *<cSourceFile>* is still open. If this is the case the user can be prompted to insert a new diskette and FileCCont() continues the copying operation.

Note: when a large file is copied to multiple diskettes, it can be re-assembled using the FileAppend() function.

Info

See also:	CSetSafety(), FileAppend(), FileCClose(), FileCCont()
Category:	CT:DiskUtil, File functions, Low level file functions
Source:	ct\fcopy.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates the two modes of FileCopy()

   PROCEDURE Main
      LOCAL cSourceFile := "LargeFile.txt"
      LOCAL cTargetFile := "NewLargeFile.txt"
      LOCAL nCounter    := 1

      CLS

      ? "Regular copy "
```

```
    ?? FileCopy( cSourceFile, cTargetFile ), "bytes"

    ? "Backup copy "
    cTargetFile := "A:\backup.001"

    ?? cTargetFile, FileCopy( cSourceFile, cTargetFile ), "bytes"

    DO WHILE FileCOpen()
       WAIT "Please insert a new diskette in drive A:"

       nCounter ++
       cTargetFile := "A:\backup." + StrZero( nCounter, 3 )
       ? cTargetFile, FileCCont( cTargetFile ), "bytes"
    ENDDO

 RETURN
```

FileDate()

Returns the date of a file.

Syntax

```
FileDate( [<cFileName>] ) --> dDate
```

Arguments

<cFileName>

This is an optional character string containing the name of the file to query. It defaults to the last file found with FileSeek(). If specified, it must contain a file extension. If *<cFileName>* contains no drive and/or directory information, the current directory is used. SET DEFAULT and SET PATH settings are ignored.

Return

The function returns the date of the file specified or an empty date in case of an error.

Info

See also:	FileAttr(), FileSeek(), FileSize(), FileTime()
Category:	CT:DiskUtil, File functions, Low level file functions
Source:	ct\files.c
LIB:	xhb.lib
DLL:	xhbdll.dll

FileDelete()

Deletes one or more files specified by a file mask and file attributes.

Syntax

```
FileDelete( <cFileMask>, [<nAttributes>] ) --> lDeleted
```

Arguments

`<cFileMask>`

This is a character string holding the drive, directory and/or file specification of the file(s) to delete. It may contain "wild card characters" (eg: "c:\xhb\data*.cdx").

`<nAttributes>`

This is a numeric value specifying the file attributes of the files to query. Values of the following list are used for file attributes. To specifiy multiple attributes, pass the sum of the corresponding values:

Values for file attributes

Value	Attribute
0	Normal
1	Read only
2	Hidden
4	System
8	Volume
16	Directory
32	Archived

Return

The return value is .T. (true) if at least one file was deleted, otherwise .F. (false).

Info

See also: DeleteFile(), FErase()
Category: CT:DiskUtil, File functions, Low level file functions
Source: ct\files.c
LIB: xhb.lib
DLL: xhbdll.dll

FileMove()

Moves a file to another directory.

Syntax

```
FileMove( <cSourceFile>, <cTargetFile> ) --> nErrorCode
```

Arguments

<cSourceFile>

This is a character string holding the name of the file to move. It must include path and file extension. The path can be omitted from <*cSourceFile*> when the file resides in the current directory.

<cTargetFile>

This is a character string with the new file name including file extension. Drive and/or path are optional.

Return

The function returns zero on success or a numeric error code on failure.

Description

The function moves <*cSourceFile*> to a file specified with <*cTargetFile*>. The target file must be on the same drive as the source file.

Info

See also:	COPY FILE, FileCopy()
Category:	CT:DiskUtil, File functions, Low level file functions
Source:	ct\disk.c
LIB:	xhb.lib
DLL:	xhbdll.dll

FileReader()

Returns a new TStreamFileReader object.

Syntax

```
FileReader( <cFileName>, [<nMode>] ) --> oTStreamFileReader
```

Arguments

<cFileName>

This is a character string holding the name of the file to open for reading. It must include path and file extension. If the path is omitted from *<cFileName>*, the file is searched in the current directory.

<nMode>

A numeric value specifying the open mode and access rights for the file. #define constants from the FILEIO.CH file can be used for *<nMode>* as listed in the table below:

File open modes

Constant	Value	Description
FO_READ	0	Open file for reading
FO_WRITE	1	Open file for writing
FO_READWRITE *)	2	Open file for reading and writing

**) default*

Constants that define the access or file sharing rights can be added to an FO_* constant. They specify how file access is granted to other applications in a network environment.

File sharing modes

Constant	Value	Description
FO_COMPAT *)	0	Compatibility mode
FO_EXCLUSIVE	16	Exclusive use
FO_DENYWRITE	32	Prevent other applications from writing
FO_DENYREAD	48	Prevent other applications from reading
FO_DENYNONE	64	Allow others to read or write
FO_SHARED	64	Same as FO_DENYNONE

**) default*

Return

The function returns a new, initialized TStreamFileReader object.

Description

Function TStreamFileReader() is the functional equivalent of TStreamFileReader():new(). Refer to the description of the TStreamFileReader object.

Info

See also:	TStreamFileReader()
Category:	Object functions, xHarbour extensions
Header:	fileio.ch
Source:	rtl\stream.prg
LIB:	lib\xhb.lib
DLL:	dll\xhbdll.dll

FileScreen()

Reads the contents of a screen from a file.

Syntax

```
FileScreen( <cFileName>, [<nOffSet>]) --> nBytesRead
```

Arguments

<cFilename>

This is a character string holding the name of the file to load a screen from, previously saved with ScreenFile(). It must include path and file extension. If the path is omitted from *<cFileName>*, the file is searched in the current directory.

<nOffSet>

If multiple screens are saved to *<cFileName>* an offset of (nScreenNumber-1) * nScreenSize bytes must be specified for *<nOffSet>* to read consecutive screens from the file. The screen size in bytes can be calulated as follows:

```
nScreenSize := 2 * (MaxRow()+1) * (MaxCol()+1)
```

Return

The function loads a saved screen from *<cFileName>*, displays it and returns the number of bytes read as a numeric value.

Info

See also:	FOpen(), FRead(), SaveScreen(), ScreenFile()
Category:	CT:Video, Screen functions
Source:	ct\strfile.c
LIB:	xhb.lib
DLL:	xhbdll.dll

FileSeek()

Seeks files specified by a file mask and file attributes.

Syntax

```
FileSeek( [<cFileMask>]  , ;
          [<nAttributes>]  ) --> cFileName
```

Arguments

`<cFileMask>`

This is a character string holding the drive, directory and/or file specification of the file(s) to seek. It may contain "wild card characters" (eg: "c:\xhb\data*.cdx").

`<nAttributes>`

This is a numeric value specifying the file attributes of the files to query. Values of the following list are used for file attributes. To specifiy multiple attributes, pass the sum of the corresponding values:

Values for file attributes

Value	Attribute
0	Normal
1	Read only
2	Hidden
4	System
8	Volume
16	Directory
32	Archived

Return

Returns a filename matching the specified parameters or a null string ("") if no further file is found.

Description

The parameters for FileSeek() must be specified for an initial call of the function. Subsequent calls can be performed without parameters. FileSeek() returns the next file matching the initial parameters until no further file is found.

Info

See also: FileAttr(), FileDate(), FileSize(), FileTime()
Category: CT:DiskUtil, File functions, Low level file functions
Source: ct\files.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays all PRG files in the current directory

   PROCEDURE Main
      LOCAL cFileName

      ? cFileName := FileSeek( "*.prg" )

      DO WHILE .NOT. Empty( cFileName )
```

```
      ? cFileName := FileSeek()
   ENDDO
RETURN
```

FileSize()

Returns the size of a file.

Syntax

```
FileSize( [<cFileName>], [<nAttributes>] ) --> nFileSize
```

Arguments

`<cFileName>`

This is an optional character string containing the name of the file to query. It defaults to the last file found with FileSeek(). If specified, it must contain a file extension. If *<cFileName>* contains no drive and/or directory information, the current directory is used. SET DEFAULT and SET PATH settings are ignored.

`<nAttributes>`

This is a numeric value specifying the file attributes of the files to query. Values of the following list are used for file attributes. To specifiy multiple attributes, pass the sum of the corresponding values:

Values for file attributes

Value	Attribute
0	Normal
1	Read only
2	Hidden
4	System
8	Volume
16	Directory
32	Archived

Return

The function returns the size of the specified file in bytes as a numeric value, or -1 in case of an error.

Info

See also: FileAttr(), FileDate(), FileSeek(), FileTime()
Category: CT:DiskUtil, File functions, Low level file functions
Source: ct\files.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example lists file name and size of all PRG files
// in the current directory.

   PROCEDURE Main
      LOCAL cFileName

      cFileName := FileSeek( "*.prg" )

      DO WHILE .NOT. Empty( cFileName )
         ? cFileName, FileSize()
         cFileName := FileSeek()
      ENDDO
   RETURN
```

FileStats()

Retrieves file information for a single file.

Syntax

```
FileStats( <cFileName>    , ;
          [@<cFileAttr>]  , ;
          [@<nFileSize>]  , ;
          [@<dCreateDate>], ;
          [@<nCreateTime>], ;
          [@<dChangeDate>], ;
          [@<nChangeTime>]  ) --> lSuccess
```

Arguments

`<cFileName>`

This is a character string holding the name of the file to query. It must include path and file extension. If the path is omitted from *<cFileName>*, the file is searched in the current directory.

`@<cFileAttr>`

If specified, *<cFileAttr>* must be passed by reference. It receives the file attributes of the file as a character string.

`@<nFileSize>`

If specified, *<cFileSize>* must be passed by reference. It receives the file size of the file as a numeric value.

`@<dCreateDate>`

If specified, *<dCreateDate>* must be passed by reference. It receives the creation date of the file as a date value.

`@<nCreateTime>`

If specified, *<nCreateTime>* must be passed by reference. It receives the creation time of the file as a numeric value. The unit is "seconds elapsed since midnight". Use function TString() to convert this value to a "hh:mm:ss" formatted time string.

`@<dChangeDate>`

If specified, *<dChangeDate>* must be passed by reference. It receives the last change date of the file as a date value.

`@<nChangeTime>`

If specified, *<nChangeTime>* must be passed by reference. It receives the creation time of the file as a numeric value. The unit is "seconds elapsed since midnight".

Return

The return value is .T. (true) when the information on the file *<cFileName>* could be retrieved, otherwise .F. (false) is returned.

Description

Function FileStats() retrieves statistical information about a single file. It is more efficient than the Directory() function which retrieves the same information for a group of files and stores them in an array. FileStats(), in contrast, allows for "picking" desired information about a single file by passing the according parameters by reference to the function.

Info

See also: Directory(), FCreate(), HB_FSize()
Category: File functions, Low level file functions, xHarbour extensions
Source: rtl\filestat.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example shows how to retrieve statistical information about
// a single file, and how to convert the time values from Seconds
// to a hh:mm:ss time formatted string.

   PROCEDURE Main
      LOCAL cFileName := "FILESTATS.PRG"
      LOCAL cFileAttr  , nFileSize
      LOCAL dCreateDate, nCreateTime
      LOCAL dChangeDate, nChangeTime

      ? FileStats( cFileName, @cFileAttr  , @nFileSize  , ;
                              @dCreateDate, @nCreateTime, ;
                              @dChangeDate, @nChangeTime  )

      ? "File statistiscs"
      ? "File Name :", cFileName
      ? "Attributes:", cFileAttr
      ? "File Size :", nFileSize
      ? "Created   :", dCreateDate, TString( nCreateTime )
      ? "Changed   :", dChangeDate, TSTring( nChangeTime )
   RETURN
```

FileStr()

Reads a string from a file beginning at a specified offset.

Syntax

```
FileStr( <cFileName>, ;
        [<nBytes>]  , ;
        [<nStart>]  , ;
        [<lCtrl_Z>]   ) --> cString
```

Arguments

<cFileName>

This is a character string holding the name of the file to read from. It must include path and file extension. If the path is omitted from *<cFileName>*, the file is searched in the current directory.

<nBytes>

This is a numeric value specifying the number of bytes to read from the file. If omitted, the entire file is read.

<nStart>

This optional numeric parameter specifies the number of bytes to skip before reading from the file. It defaults to 0 bytes, which reads from the beginning of the file.

<lCtrl_Z>

This parameter defaults to .F. (false). When .T. (true) is passed, only the characters up to the first Ctrl-Z character (Chr(26)) are returned. This is useful for reading Chr(26) terminated DBT memo fields directly from a DBT file.

Return

The function returns a character string holding the data read from *<cFileName>*, or a null string in case of an error or when the end-of-file is reached.

Info

See also: FRead(), FReadStr(), StrFile()
Category: CT:DiskUtil, File functions, Low level file functions
Source: ct\strfile.c
LIB: xhb.lib
DLL: xhbdll.dll

FileTime()

Returns the change time of a file.

Syntax

```
FileTime( [<cFileName>], [<nAttributes>] ) --> cFileTime
```

Arguments

`<cFileName>`

This is an optional character string containing the name of the file to query. It defaults to the last file found with FileSeek(). If specified, it must contain a file extension. If *<cFileName>* contains no drive and/or directory information, the current directory is used. SET DEFAULT and SET PATH settings are ignored.

`<nAttributes>`

This is a numeric value specifying the file attributes of the files to query. Values of the following list are used for file attributes. To specifiy multiple attributes, pass the sum of the corresponding values:

Values for file attributes

Value	Attribute
0	Normal
1	Read only
2	Hidden
4	System
8	Volume
16	Directory
32	Archived

Return

The function returns the time of the specified file in bytes as a Time() formatted character string, or a null string ("") in case of an error.

Info

See also: FileAttr(), FileDate(), FileSeek(), FileSize()
Category: CT:DiskUtil, File functions, Low level file functions
Source: ct\files.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example lists file name and time of all PRG files
// in the current directory.

   PROCEDURE Main
      LOCAL cFileName

      cFileName := FileSeek( "*.prg" )

      DO WHILE .NOT. Empty( cFileName )
         ? cFileName, FileTime()
         cFileName := FileSeek()
```

```
   ENDDO
RETURN
```

FileValid()

Tests if a string contains a valid file name.

Syntax

```
FileValid( <cFileName>     , ;
          [<nMaxName>]     , ;
          [<nMaxExtension>], ;
          [<lNoExtension>] , ;
          [<lSpaces>]        ) --> lIsValid
```

Arguments

<cFileName>

This is a character string holding a file name. By default, only FAT compatible 8.3 file names are checked (DOS file system).

<nMaxName>

This optional numeric parameter defaults to 8, which is the maximum length of file names for the FAT file system. Pass the value 255 for file systems supporting long file names.

<nMaxExtension>

This optional numeric parameter defaults to 3, which is the maximum length of file extensions for the FAT file system. Pass the value 255 for file systems supporting longer file extensions.

<lNoExtension>

This optional parameter defaults to .T. (true), indicating that the file extension should not be validated. Passing .F. (false) includes the file extension in the validation test.

<lSpaces>

This optional parameter defaults to .F. (false), indicating that blank spaces are illegal in a file name. Passing .F. (false) accepts blank spaces as valid characters of the file name.

Return

The function returns .T. (true) if *<cFileName>* contains a valid filename, otherwise .F. (false) is returned.

Info

See also:	HB_FNameSplit(), Token(), TrueName()
Category:	CT:DiskUtil, File functions, Low level file functions
Source:	ct\diskutil.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example implements the function LongFileValid() which tests
// if a long file name is valid

   PROCEDURE Main

      ? FileValid( "MyApp.prg" )                  // result: .T.

      ? FileValid( "My New Application.prg" )     // result: .F.

      ? LongFileValid( "My New Application.prg" )  // result: .T.
```

```
RETURN

FUNCTION LongFileValid( cFileName, lExtension )

   IF Valtype( lExtension ) <> "L"
      lExtension := .F.
   ENDIF

RETURN FileValid( cFileName, 255, 255, lExtension, .T. )
```

FileWriter()

Returns a new TStreamFileWriter object.

Syntax

```
FileWriter( <cFileName>, [<nFileAttr>] ) --> oTStreamFileWriter
```

Arguments

<cFilename>

This is a character string holding the name of the file to create and/or write to. It must include path and file extension. If the path is omitted from *<cFileName>*, the file is created or opened in the current directory.

<nFileAttr>

A numeric value specifying one or more attributes associated with the file to create or open. #define constants from the FILEIO.CH file can be used for *<nFileAttr>* as listed in the table below:

Attributes for binary file creation

Constant	Value	Attribute	Description
FC_NORMAL *)	0	Normal	Creates a normal read/write file
FC_READONLY	1	Read-only	Creates a read-only file
FC_HIDDEN	2	Hidden	Creates a hidden file
FC_SYSTEM	4	System	Creates a system file

**) default attribute*

Return

The function returns a new, initialized TStreamFileWriter object.

Description

Function TStreamFileWriter() is the functional equivalent of TStreamFileWriter():new(). Refer to the description of the TStreamFileWriter object.

Info

See also:	TStreamFileWriter()
Category:	Object functions, xHarbour extensions
Header:	fileio.ch
Source:	rtl\stream.prg
LIB:	lib\xhb.lib
DLL:	dll\xhbdll.dll

FkLabel()

Returns a function key name.

Syntax

```
FKLABEL( <nFunctionKey> ) --> cKeyLabel
```

Arguments

<nFunctionKey>

A numeric value identifying a function key. It must be in the range of 1 to 40.

Return

The function returns a character string containing the label of a function key, or a null string ("") when an invalid parameter is passed.

Description

This function exists for compatibility reasons only and is not recommended to be used in new xHarbour programs.

Info

See also:	FkMax()
Category:	Keyboard functions
Source:	rtl\fkmax.c
LIB:	xhb.lib
DLL:	xhbdll.dll

FkMax()

Returns the number of available function keys.

Syntax

```
FkMax() --> nFKeyCount
```

Return

The function returns the number of available function keys.

Description

This function exists for compatibility reasons only and is not recommended to be used in new xHarbour programs.

Info

See also: FkLabel()
Category: Keyboard functions
Source: rtl\fkmax.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example lists the names of all function keys

   PROCEDURE Main
      LOCAL n
      ?
      FOR n:=1 TO FkMax()
         ?? "", FkLabel(n)
         IF n % 10 == 0
            ?
         ENDIF
      NEXT
   RETURN
```

FLineCount()

Counts the lines in an ASCII text file.

Syntax

```
FLineCount( <cFileName> ) --> nLineCount
```

Arguments

<cFileName>

This is a character string holding the name of the text file whose lines to count. It must include path and file extension. If the path is omitted from *<cFileName>*, the file is searched in the current directory.

Return

The function returns the number of lines contained in *<cFileName>* as a numeric value. If the file *<cFileName>* does not exist or cannot be opened, the return value is zero.

Description

FLineCount() is an optimized function for counting the lines in an ASCII text file fast and efficiently. It can be used in conjunction with HB_FReadLine() to extract single lines from an ASCII text file.

Info

See also:	FileStats(), FCharCount(), FCreate(), FOpen(), FParse(), FWordCount(), HB_FReadLine(), MemoLine(), MlCount()
Category:	File functions, xHarbour extensions
Source:	rtl\fparse.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example uses FLineCount() to determine the size of an
// array used to collect all lines of a text file.

   PROCEDURE Main
      LOCAL cFileName   := "Test.txt"
      LOCAL nCount      := FLineCount( cFileName )
      LOCAL aLines      := Array( nCount )
      LOCAL nFileHandle := FOpen( cFileName )
      LOCAL nLine       := 0
      LOCAL cLine

      FOR EACH cLine IN aLines
         HB_FReadLine( nFileHandle, @cLine )
         aLines[ ++nLine ] := cLine
      END

      FClose( nFileHandle )

      ? Len( aLines )
      AEval( aLines, {|c| QOut( c ) } )
   RETURN
```

FLock()

Applies a file lock to an open, shared database.

Syntax

```
FLock() --> lSuccess
```

Return

The function returns .T. (true) when the entire file open in a work area is successfully locked, otherwise .F. (false).

Description

The function FLock() is used when a database open in SHARED mode in a work area must be protected entirely from concurrent updates by other applications in a multi-user scenario. As long as an application has obtained a file lock with FLock(), other applications can read records from the file, but are excluded from updating any record in the file with new data. Therefore, file locks should be used with special care and should be released as soon as possible with DbUnlock().

Before a file lock is placed, it is recommended to consider whether or not the same protection could be achieved with one or multiple record locks. They are created with the Rlock() or DbRLock() functions and are by far less restrictive for network operations.

Certain database updates, however, require an application to obtain a file lock, otherwise a safe operation is not guaranteed to complete. As a general rule, database operations that involve many records require a file lock, while those updating only one or few records can be achieved using record locks.

Commands requiring an FLock()

Command	Protection
APPEND FROM	FLock() or USE...EXCLUSIVE
DELETE (multiple records)	FLock() or USE...EXCLUSIVE
RECALL (multiple records)	FLock() or USE...EXCLUSIVE
REPLACE (multiple records)	FLock() or USE...EXCLUSIVE
UPDATE ON	FLock() or USE...EXCLUSIVE

FLock() attempts to lock the database once only. If this attempt fails, the function returns .F. (false). Common reasons for failure are that another application has currently obtained a record lock or a file lock on the same database file.

Info

See also:	DbRLock(), DbUnlock(), DbUnlockAll(), DbUseArea(), NetFileLock(), RLock(), SET EXCLUSIVE, UNLOCK, USE
Category:	Database functions, Network functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// This example uses FLock() to delete many records at once.

   PROCEDURE Main

      USE Customer SHARED NEW

      IF FLock()
```

```
      DELETE FOR Year(Date())-Year(Cust->LastDate) > 5
      DbCommit()
      DbUnlock()
   ELSE
      ? "Error: unable to lock file."
   ENDIF

   USE
RETURN
```

Floor()

Rounds a decimal number to the next lower integer.

Syntax

```
Floor( <nValue> ) --> nInteger
```

Arguments

`<nValue>`

Any numeric value can be passed.

Return

The return value is the next lower integer value of *<nValue>*. If *<nValue>* is an integer already, it is returned unchanged.

Info

See also:	Ceiling(), Round()
Category:	CT:Math, Mathematical functions
Source:	ct\ctmath2.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows results of function Floor().

   PROCEDURE Main

      ? Floor( -2.0 )       // result: -2

      ? Floor( -2.00001 )   // result: -3

      ? Floor(  2.0 )       // result: 2

      ? Floor(  2.00001 )   // result: 2

   RETURN
```

FloppyType()

Determines the type of a floppy drive.

Syntax

```
FloppyType( [<cDrive>] ) --> nFloppyType
```

Arguments

<cDrive>

This parameter specifies the floppy drive to query. It can be either an upper case *A* or *B*, indicating the drive letter. The default value is *A*.

Return

The function returns a numeric value indicating the floppy drive type. The following values are possible:

Return values of FloppyType()

Value	Description
0	Drive is not a floppy drive
1	Drive is a 360 KB floppy
2	Drive is a 720 KB floppy
3	Drive is a 1.2 MB floppy
4	Drive is a 1.44 MB floppy

Info

See also: DriveType()
Category: CT:DiskUtil, Disks and Drives
Source: ct\diskutil.prg
LIB: xhb.lib
DLL: xhbdll.dll

FOpen()

Opens a file on the operating system level.

Syntax

```
FOpen( <cFileName>, [<nMode>]) --> nFileHandle
```

Arguments

`<cFilename>`

This is a character string holding the name of the file to open. It must include path and file extension. If the path is omitted from *<cFileName>*, the file is searched in the current directory.

`<nMode>`

A numeric value specifying the open mode and access rights for the file. #define constants from the FILEIO.CH file can be used for *<nMode>* as listed in the table below:

File open modes

Constant	Value	Description
FO_READ *)	0	Open file for reading
FO_WRITE	1	Open file for writing
FO_READWRITE	2	Open file for reading and writing

**) default*

Constants that define the access or file sharing rights can be added to an FO_* constant. They specify how file access is granted to other applications in a network environment.

File sharing modes

Constant	Value	Description
FO_COMPAT *)	0	Compatibility mode
FO_EXCLUSIVE	16	Exclusive use
FO_DENYWRITE	32	Prevent other applications from writing
FO_DENYREAD	48	Prevent other applications from reading
FO_DENYNONE	64	Allow others to read or write
FO_SHARED	64	Same as FO_DENYNONE

**) default*

Return

The function returns a numeric file handle to be used later for accessing the file using FRead() or FWrite(). The return value is -1 when the operation fails. Use FError() to determine the cause of failure.

Description

The low-level file function FOpen() opens a file on the operating system level. The returned file handle must be preserved in a variable to be passed to other low-level file functions until the file is closed. Without the file handle, data cannot be written to or read from the file.

Open mode and sharing rights can be specified with *<nMode>*. The default open and sharing mode grants shared read access to the application.

FOpen() does not search the directories specified with SET PATH or SET DEFAULT. Therefore, the file name must be specified including directory information and extension. If no directory information is provided, FOpen() searches the file only in the current directory.

Info

See also: DisableWaitLocks(), FClose(), FCreate(), FError(), FRead(), FSeek(), FWrite()
Category: File functions, Low level file functions
Header: FileIO.ch
Source: rtl\philes.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example implements a user-defined function that reads the
// entire contents of a file into a memory variable using
// low-level file functions:

   #include "FileIO.ch"

   PROCEDURE Main
      LOCAL cFile := "Fopen.prg" //"MyFile.txt"
      LOCAL cStream

      IF .NOT. ReadStream( cFile, @cStream )
         IF FError() <> 0
            ? "Error reading file:", FError()
         ELSE
            ? "File is empty"
         ENDIF
      ELSE
         ? "Successfully read", Len(cStream), "bytes"
      ENDIF
   RETURN

   FUNCTION ReadStream( cFile, cStream )
      LOCAL nFileHandle := FOpen( cFile )
      LOCAL nFileSize

      IF FError() <> 0
         RETURN .F.
      ENDIF

      nFileSize := FSeek( nFileHandle, 0, FS_END )
      cStream   := Space( nFileSize )
      FSeek( nFileHandle, 0, FS_SET )
      FRead( nFileHandle, @cStream, nFileSize )
      FClose( nFileHandle )
   RETURN ( FError() == 0 .AND. .NOT. Empty( cStream ) )
```

Found()

Checks if the last database search operation was successful

Syntax

```
Found() --> lSuccess
```

Return

The function returns .T. (true) when the last search operation in a work area was successful, otherwise .F. (false) is returned. The return value is always .F. (false), when a work area is unused.

Description

The Found() function returns a status of a work area that indicates success of a search operation. Each work area has a Found status which is set when a search function or command is executed in a work area. The Found status is set to .T. (true) when a record matching a search condition is found. If the Found status is .T. (true), it is reset to .F. (false) when the record pointer is moved again.

This most commonly used search function is DbSeek() which searches a value in an index. This search sets the Found status explicitely. If a work area is related to a parent work area via SET RELATION, the Found status of the child work area is implicitly set to .T. (true) when a record matches the relation expression.

Info

See also: DbSeek(), DbSetRelation(), Eof(), LOCATE, SEEK, SET RELATION, SET SOFTSEEK
Category: Database functions, Index functions
Source: rdd\dbcmd.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example illustrates return values of Found()
// in different work areas.

   PROCEDURE Main

      USE Invoice NEW ALIAS Inv
      INDEX ON CustNo TO InvA
      SET INDEX TO InvA

      USE Customer ALIAS Cust
      INDEX ON Upper(LastName+FirstName) TO Cust01
      SET INDEX TO Cust01

      ? Alias()                      // result: CUST
      DbSeek( "BECKER" )
      ? Found()                      // result: .T.

      ? Inv->( Found() )             // result: .F.

      Inv->( DbSeek( Cust->CustNo ) )
      ? Inv->( Found() )             // result: .T.

      ? Alias()                      // result: CUST
      DbSkip()
      ? Found()                      // result: .F.
```

```
   ? Inv->( Found() )             // result: .T.

   CLOSE DATABASES
RETURN
```

FParse()

Parses a delimited text file and loads it into an array.

Syntax

```
FParse( <cFileName>, <cDelimiter> ) --> aTextArray
```

Arguments

`<cFileName>`

This is a character string holding the name of the text file to load into an array. It must include path and file extension. If the path is omitted from *<cFileName>*, the file is searched in the current directory.

`<cDelimiter>`

This is a single character used to parse a single line of text. It defaults to the comma.

Return

The function returns a two dimensional array, or an empty array when the file cannot be opened.

Description

Function FParse() reads a delimited text file and parses each line of the file at *<cDelimiter>*. The result of line parsing is stored in an array. This array, again, is collected in the returned array, making it a two dimensional array

FParse() is mainly designed to read the comma-separated values (or CSV) file format, were fields are separated with commas and records with new-line character(s).

Info

See also: FCreate(), FOpen(), FCharCount(), FLineCount(), FParseEx(), FParseLine(), MemoLine(), MlCount()
Category: File functions, xHarbour extensions
Source: rtl\fparse.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example reads a CSV file and displays its records
// and field data

   PROCEDURE Main()
      LOCAL aText := FPparse( "Testdata.csv" )
      LOCAL aRecord, cData

      FOR EACH aRecord IN aText     //
         ? "Record #" + CStr( HB_EnumIndex() ), ": "

         FOR EACH cData IN aRecord
            ?? cData + " "
         NEXT
      NEXT

   RETURN
```

FParseEx()

Parses a delimited text file and loads it into an array (optimized).

Syntax

```
FParseEx( <cFileName>, [<cDelimiter>] ) --> aTextArray
```

Arguments

`<cFileName>`

This is a character string holding the name of the text file to load into an array. It must include path and file extension. If the path is omitted from *<cFileName>*, the file is searched in the current directory.

`<cDelimiter>`

This is a single character used to parse a single line of text. It defaults to the comma.

Return

The function returns a two dimensional array, or an empty array when the file cannot be opened.

Description

Function FParseEx() is a speed optimized version of FParse(). Refer to FParse() for more information.

Info

See also:	FParse()
Category:	File functions, xHarbour extensions
Source:	rtl\fparse.c
LIB:	xhb.lib
DLL:	xhbdll.dll

FParseLine()

Parses one line of a delimited text and loads it into an array.

Syntax

```
FParseLine( <cTextLine>, [<cDelimiter>]  ) --> aTextFields
```

Arguments

`<cTextLine>`

This is a character string holding on line of delimited text.

`<cDelimiter>`

This is a single character used to parse <*cTextLine*>. It defaults to the comma.

Return

The function returns a one dimensional array, or an empty array when an error occurs.

Description

Function FParseLine() parses one line of delimited text at <*cDelimiter*>. The result is stored in a one dimensional array.

FParseLine() is mainly designed to process the comma-separated values (or CSV) file format, were fields are separated with commas and records with new-line character(s).

Note: function FParse() is available to process an entire CSV file.

Info

See also:	FParse(), HB_FReadLine(), MemoLine()
Category:	File functions, xHarbour extensions
Source:	rtl\fparse.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example parses one line of delimited text and displays
// the resukt. Note that commas exist in the first and sixth
// text field of the CSV formatted string.

   PROCEDURE Main()
      LOCAL cText, aData

      cText := '"Jones, Mr",Male,"Oklahoma","IL",20041231,"Director, President"'

      aData := FParseLine( cText )

      FOR EACH cText IN aData
         ? LTrim( Str( HB_EnumIndex() ) ), cText
      NEXT

      ** Output
      // 1 Jones, Mr
      // 2 Male
      // 3 Oklahoma
      // 4 IL
      // 5 20041231
```

```
   // 6 Director, President
RETURN
```

FRead()

Reads characters from a binary file into a memory variable.

Syntax

```
FRead( <nFileHandle>, ;
      @<cBuffer>    , ;
       <nBytes>     , ;
      [<nOffset>]     ) --> nBytesRead
```

Arguments

<nFileHandle>

This is a numeric file handle returned from function FOpen() or FCreate().

@<cBuffer>

A memory variable holding a character string must be passed by reference to FRead(). It must have at least <*nBytes*> characters.

<nBytes>

This is a numeric value specifying the number of bytes to transfer from the file into the memory variable <*cBuffer*>, beginning at the current file pointer position.

<nOffset>

This is a numeric value specifying the number of bytes to skip at the beginning of <*cBuffer*> where the result is copied to. This allows to copy the data from the file into the middle of a string buffer. The default value is zero. Note that the sum of <*nBytes*>+<*nOffset*> must be smaller than or equal Len(<*cBuffer*>).

Return

The function returns a numeric value indicating the number of bytes that are transfered from the file into the memory variable. If the return value is smaller than Len(<*cBuffer*>), either the end of file is reached, or a file read error ocurred. This can be identified using function FError().

Description

The low-level file function FRead() reads bytes from a file and copies them into a memory variable. To accomplish this, FRead() must receive a string buffer <*cBuffer*>, large enough to receive the number of bytes specified with <*nBytes*>. Since the bytes are copied into a memory variable, the buffer variable must be passed by reference to FRead().

FRead() advances the file pointer by the number of bytes read and returns this number. When the end of file is reached, the file pointer cannot be advanced further, so that the return value is usually smaller than <*nBytes*>. Use FSeek() to reposition the file pointer without reading the file.

If an error occurs during the operation, function FError() returns an error code indicating the cause of failure.

Info

See also: Bin2I(), Bin2L(), Bin2U(), Bin2W(), FClose(), FCreate(), FError(), FOpen(), FReadStr(), FSeek(), FWrite(), HB_FReadLine()
Category: File functions, Low level file functions
Source: rtl\philes.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example reads an entire file in blocks of 4096 bytes and fills
// them into an array

   #define BLOCK_SIZE     4096

   PROCEDURE Main
      LOCAL cBuffer     := Space( BLOCK_SIZE )
      LOCAL nFileHandle := FOpen( "MyFile.log")
      LOCAL aBlocks     := {}
      LOCAL nRead

      IF FError() <> 0
         ? "Error opening file:", FERROR()
         QUIT
      ENDIF

      DO WHILE .T.
         nRead := FRead( nFileHandle, @cBuffer, BLOCK_SIZE )
         IF nRead == BLOCK_SIZE
            AAdd( aBlocks, cBuffer )
         ELSE
            // end of file reached
            AAdd( aBlocks, SubStr(cBuffer,nRead) )
            EXIT
         ENDIF
      ENDDO

      FClose( nFileHandle )
      ? Len( aBlocks ), "Blocks of", BLOCK_SIZE, "bytes read"
   RETURN
```

FReadStr()

Reads characters up to Chr(0) from a binary file.

Syntax

```
FreadStr( <nFileHandle>, <nBytes> ) --> cString
```

Arguments

<nFileHandle>

This is a numeric file handle returned from function FOpen() or FCreate().

<nBytes>

This is a numeric value specifying the number of bytes to read from the file, beginning at the current file pointer position.

Return

The function returns the bytes read from the file as a character string. If an error occurs, a null string ("") is returned.

Description

The low-level file function FReadStr() reads *<nBytes>* bytes from an open file and returns them as a character string. The file pointer is advanced by the number of bytes read. The function reads all bytes up to the end of file, except Chr(0). The bytes up to the first Chr(0), but not including Chr(0), are returned.

Info

See also:	Bin2I(), Bin2L(), Bin2U(), Bin2W(), FClose(), FCreate(), FError(), FOpen(), FRead(), FSeek(), FWrite(), HB_FReadLine()
Category:	File functions, Low level file functions
Source:	rtl\philes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example reads a file using FReadStr() and FRead() and
// compares the results.

   #include "Fileio.ch"

   PROCEDURE Main
      LOCAL cBuffer
      LOCAL nHandle

      nHandle := FCreate( "Testfile.txt", FC_NORMAL )
      FWrite( nHandle , "xHarbour" + Chr(0) + "compiler" )
      FClose( nHandle )

      nHandle  := FOpen( "Testfile.txt" , FO_READ )

      ? cBuffer := FReadStr( nHandle, 20 ) // result: xHarbour
      ? Len( cBuffer )                     // result: 8

      FSeek(nHandle, 0, FS_SET )           // go to begin of file

      cBuffer := Space( 20 )
```

```
   ? FRead( nHandle, @cBuffer, 20 )      // result: 17
   ? Trim( cBuffer )                     // result: xHarbour compiler
   ? Asc( cBuffer[9] )                   // result: 0

   FClose( nHandle )
   FErase( "Testfile.txt" )
RETURN
```

FreeLibrary()

Releases a dynamically loaded external DLL from memory.

Syntax

```
FreeLibrary( <nDllHandle> ) --> lSuccess
```

Arguments

<nDllHandle>

This is a numeric handle of the DLL to release as returned from LoadLibrary().

Return

The function returns a logical value inidcating a successful operation.

Description

Function FreeLibrary() releases a DLL previously loaded with LoadLibrary() from memory. Internally, the function decrements the load counter of the DLL with the handle *<nDllHandle>* to signal the operating system that the DLL is no longer required by the xHarbour application. Whether or not the DLL is actually removed from memory is controlled by the operating system, since the same DLL could be in use by multiple applications.

Info

See also: DllCall(), LibFree(), LoadLibrary()
Category: DLL functions, xHarbour extensions
Source: rtl\dllcall.c
LIB: xhb.lib
DLL: xhbdll.dll

FRename()

Renames a file.

Syntax

```
FRename( <cOldFile>, <cNewFile> ) --> nSuccess
```

Arguments

<cOldFile>

This is a character string holding the name of the file to rename. It must include path and file extension. The path can be omitted from *<cOldFile>* when the file resides in the current directory.

<cNewFile>

This is a character string with the new file name including file extension. Drive and/or path are optional.

Return

The function returns a numeric value. 0 indicates success and -1 is returned for failure. The cause of a failure can be determined using function FError()

Description

The FRename() function changes the name of a file. The file *<cOldFile>* is searched in the current directory only, unless a full qualified file name including drive and path is specified. Directories specified with SET DEFAULT and SET PATH are ignored by FRename().

If *<cNewFile>* is specified as a full qualified file name and its directory differs from the one of *<cOldFile>*, the source file is moved to the new directory and stored under the new file name.

When the new file either exists or is currently open, FRename() aborts the operation. Use the File() function to test for the existence of *<cNewFile>*.

A file must be closed before attempting to rename it.

Info

See also:	CLOSE, ERASE, FCReate(), FErase(), FError(), File(), RENAME
Category:	File functions, Low level file functions
Source:	rtl\philes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example changes the name of an existing Log file
// for archival purposes and creates a new Log file

   PROCEDURE Main
      LOCAL nLogs, nHandle

      IF File( "Logfile.txt" )
         nLogs := 0
         DO WHILE File( "Logfile." + Padl( nLogs, 3, "0" ) )
            nLogs ++
            IF nLogs > 999
               nLogs := 0
               EXIT
```

```
            ENDIF
         ENDDO

         IF FRename( "Logfile.txt", ;
                     "Logfile." + Padl( nLogs, 3, "0" ) ) == -1
            ? "Error renaming file:", FError()
         ENDIF
      ENDIF

      nHandle := FCreate( "Logfile.txt" )
      IF FError() == 0
         FWrite( nHandle, "Created: " + DtoS(Date()), 17 )
         FClose( nHandle )
      ENDIF
   RETURN
```

FSeek()

Changes the position of the file pointer.

Syntax

```
FSeek( <nFileHandle>, <nBytes>, [<nOrigin>] ) --> nPosition
```

Arguments

`<nFileHandle>`

This is a numeric file handle returned from function FOpen(), FCreate(), or HB_FCreate().

`<nBytes>`

This is a numeric value specifying the number of bytes to move the file pointer. It can be a positive or negative number. Negative numbers move the file pointer backwards (towards the beginning of the file), positive values move it forwards. The value zero is used to position the file pointer exactly at the location specified with <*nOrigin*>.

`<nOrigin>`

Optionally, the starting position from where to move the file pointer can be specified. #define constants are available in the FILEIO.CH file that can be used for <*nOrigin*>.

Start positions for moving the file pointer

Constant	Value	Description
FS_SET *)	0	Start at the beginning of the file
FS_RELATIVE	1	Start at the current file pointer position
FS_END	2	Start at the end of the file

**) default*

Return

The function returns a numeric value. It is the new position of the file pointer, counted from the beginning of the file (position 0).

Description

The low-level file function FSeek() moves the file pointer of an open file to a new position, without reading the contents of the file. The file pointer can be moved forwards (positive <*nBytes*>) and backwards (negative <*nBytes*>). It is not possible, however, to move the file pointer outside the boundaries of a file.

Info

See also: FClose(), FCreate(), FError(), FOpen(), FRead(), FReadStr(), FWrite(), HB_F_Eof()
Category: File functions, Low level file functions
Header: FileIO.ch
Source: rtl\philes.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example implements the user-defined function FSize()
// which determines the size of a file using FSeek()

   #include "FileIO.ch"
```

```
PROCEDURE Main
   LOCAL cFile := "MyFile.txt"

   ? FSize( cFile )
RETURN

FUNCTION FSize( cFile )
   LOCAL nHandle := FOpen( cFile )
   LOCAL nSize   := -1

   IF FError() == 0
      nSize := FSeek( nHandle, 0, FS_END )
      FClose( nHandle )
   ENDIF
RETURN nSize
```

FtoC()

Converts a floating point number to an 8 byte binary string.

Syntax

```
FtoC( <nFloat> ) --> cFloat
```

Arguments

<nFloat>

Any numeric value can be passed.

Return

The function returns an 8 byte character string holding the binary representation of a floating point number. It can be converted back tu numeric using CtoF().

Info

See also:	CtoF(), XtoC()
Category:	CT:NumBits, Numbers and Bits
Source:	ct\ftoc.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Fv()

Calculates the future value of constant, periodic investments.

Syntax

```
Fv( <nInvestment>   , ;
    <nInterestRate> , ;
    <nPeriods>        ) --> nFutureValue
```

Arguments

<nInvestment>

A numeric value defining the constant, periodic investment.

<nInterestRate>

This parameter defines a constant interest rate per period during the periods of the investment in the range of 0 to 1 (1 equals 100% interest rate).

<nPeriods>

This is a numeric value defining the number of periods to include in the calculation.

Return

The function returns the future value of a periodic, constant investments at a constant interest rate over a period of time as a numeric value.

Info

See also:	Payment(), Periods(), Pv(), Rate()
Category:	CT:Math, Financial functions, Mathematical functions
Source:	ct\finan.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example calculates the future value of a monthly investment
// of 100 units of money over a period of 66 months at a savings
// interest rate of 4% per year.

   PROCEDURE Main
      LOCAL nInvestment := 100        // monthly payment
      LOCAL nInterest   := 0.04/12    // monthly interest rate
      LOCAL nMonths     := 66         // total investment period

      ? FV( nInvestment, nInterest, nMonths )  // result: 7368.63

   RETURN
```

FWordCount()

Counts the words in a text file.

Syntax

```
FWordCount( <cFileName> ) --> nWordCount
```

Arguments

<cFileName>

This is a character string holding the name of the text file whose words to count. It must include path and file extension. If the path is omitted from *<cFileName>*, the file is searched in the current directory.

Return

The function returns the number of words contained in a text file.

Description

The function counts the words in an ASCII text file. Words are recognized as sequences of characters delimited by blank spaces (Chr(32)).

Info

See also:	FCharCount(), FLineCount()
Category:	File functions, xHarbour extensions
Source:	rtl\fparse.c
LIB:	xhb.lib
DLL:	xhbdll.dll

FWrite()

Writes data to an open binary file.

Syntax

```
FWrite( <nFileHandle>, ;
         <cBuffer>    , ;
        [<nBytes>]    , ;
        [<nOffset>]    ) --> nBytesWritten
```

Arguments

<nFileHandle>

This is a numeric file handle returned from function FOpen() or FCreate().

<cBuffer>

This is a character string to be written to the open file.

<nBytes>

A numeric value specifying the number of bytes to write to the file, beginning with the first character of *<cBuffer>*. It defaults to Len(*<cBuffer>*), i.e. all bytes of *<cBuffer>*.

<nOffset>

This is a numeric value specifying the number of bytes to skip at the beginning of *<cBuffer>*. This allows to write data from the middle of a string buffer into a file. The default value is zero. Note that the sum of *<nBytes>*+*<nOffset>* must be smaller than or equal Len(*<cBuffer>*).

Return

The function returns a numeric value which is the number of bytes written to the file. If the return value equals *<nBytes>*, the operation was succesful. Any value differeing from *<nBytes>* indicates failure, which can be identified using function FError().

Description

The low-level file function FWrite() writes data provided in form of a character string into an open file. Data is written starting at the current position of the file pointer. The file pointer is advanced to a new position by the number of written bytes.

Info

See also:	FClose(), FCreate(), FError(), FOpen(), FRead(), FReadStr(), FWrite()
Category:	File functions, Low level file functions
Source:	rtl\philes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example implements the user defined function WriteStream()
// which writes an entire character string into a newly created file

   PROCEDURE Main
      LOCAL aDir   := Directory()
      LOCAL cFiles := ""
      AEval( aDir, {|a| cFiles += a[1] + Chr(13)+Chr(10) } )

      IF .NOT. WriteStream( "Files.txt", cFiles )
         ? "Error writing file", FError()
```

```
   ENDIF
RETURN

FUNCTION WriteStream( cFile, cStream )
   LOCAL nHandle := FCreate( cFile )

   IF FError() <> 0
      RETURN .F.
   ENDIF

   FWrite( nHandle, cStream, Len(cStream) )
   FClose( nHandle )
RETURN ( FError() == 0 )
```

GetActive()

Returns or assigns the current Get object during READ.

Syntax

```
GetActive( [<oGet>] ) --> oGet | NIL
```

Arguments

<oGet>

If a Get object is passed, it becomes the currently active Get entry field during READ.

Return

The function returns a reference to the currently active Get object while READ is being executed. If there is no pending READ, the return value is NIL.

Description

GetActive() is a utility function of the Get system used to query the Get object currently being edited while READ is being executed. The function also allows for changing the input focus to another Get object, since *<oGet>* becomes the new active Get object.

Info

See also:	@...GET, CurrentGet(), Get(), READ
Category:	Get system
Source:	rtl\getsys.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

GetActiveObject()

Retturns an instantiated OLE Automation object.

Syntax

```
GetActiveObject( <cProgID> ) --> oOleAuto
```

Arguments

<cProgID>

This is a character string holding the OLE ProgID of the application instance to use for automation. They consist of a program name, usually followed by ".Application". For example:

```
cProgID := "Word.Application"               // MS Office Word
cProgID := "Excel.Application"              // MS Office Excel
cProgID := "InternetExplorer.Application"   // Internet Explorer
```

Return

The function returns the TOleAuto object of an existing OLE automation instance. When the the requested OLE automation object is not running, a runtime error is raised.

Description

The function GetActiveObject() is similar to CreateObject(). Hower, it searches a running instance of the desired OLE application, rather than loading a new instance into memory.

Info

See also: CreateObject()
Category: OLE Automation, xHarbour extensions
Source: rtl\win32ole.prg
LIB: xhb.lib, ole.lib
DLL: xhbdll.dll

Example

```
// The example shows the typyical programming pattern to request a running
// OLE application. GetActiveObject() first requests an instantiated OLE
// object. When this fails, the CreateObject() functions tries to instantiate
// the OLE application. If this fails again, the requested OLE object is not
// (properly) installed on the computer and the xHarbour application quits.

   PROCEDURE Main
      LOCAL oOLE

      TRY
         oOLE := GetActiveObject( "InternetExplorer.Application" )
      CATCH
         TRY
            oOLE := CreateObject( "InternetExplorer.Application" )
         CATCH
            Alert( "ERROR! IE not avialable. [" + Ole2TxtError()+ "]" )
            QUIT
         END
      END

      <... code for OLE automation ...>
      oOle:quit()
   RETURN
```

GetApplyKey()

Sends an Inkey() code to the currently active Get object.

Syntax

```
GetApplyKey( <oGet>, <nKey> ) --> NIL
```

Arguments

<oGet>

This must be the Get object that has input focus. It is determined with GetActive().

<nKey>

This is the numeric Inkey() code of the key to send to the Get object. Refer to the file Inkey.ch for #define constants that can be used for *<nKey>*.

Return

The return value is always NIL.

Description

GetApplyKey() is a utility function of the Get system used for default processing of key strokes. It is automatically called within the Get system.

Info

See also:	@...GET, Get(), GetActive(), GetDoSetkey(), READ
Category:	Get system
Header:	Inkey.ch
Source:	rtl\getsys.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

GetClearA()

Returns the default color attribute for clearing the screen.

Syntax

```
GetClearA() --> nClearAttribute
```

Return

The function returns the default color attribute for clearing the screen as a numeric value.

Info

See also:	NToColor(), SetClearA()
Category:	CT:Video, Screen functions
Source:	ct\setclear.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays the default color attribute and its
// corresponding color value.

   PROCEDURE Main
      CLS

      ? GetClearA()                     // result:   7
      ? NToColor( GetClearA(), .T. )    // result: W/N
   RETURN
```

GetClearB()

Returns the default character for clearing the screen.

Syntax

```
GetClearB() --> nClearCharacter
```

Return

The function returns the ASCII code of the default character used for clearing the screen as a numeric value. If it is not defined with SetClearB(), the return value is 32.

Info

See also: SetClearB()
Category: CT:Video, Screen functions
Source: ct\setclear.c
LIB: xhb.lib
DLL: xhbdll.dll

GetClrBack()

Returns the background color of a color value.

Syntax

```
GetClrBack( <cColor> ) --> cBackgroundColor
```

Arguments

<cColor>

A character string holding a color value consisting of foreground and background color.

Return

The function returns the background color of the passed color value as a character string.

Note: refer to function SetColor() for an explanation how colors are encoded in a character string.

Info

See also:	GetClrFore(), HB_ColorToN(), SetColor()
Category:	Screen functions
Source:	rtl\color53.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays results of GetClrBack()

   PROCEDURE Main
      LOCAL cColor1 := "W+/B"
      LOCAL cColor2 := "N/BG"

      ? GetClrBack( cColor1 )   // result: B

      ? GetClrBack( cColor2 )   // result: BG

   RETURN
```

GetClrFore()

Returns the foreground color of a color value.

Syntax

```
GetClrFore( <cColor> ) --> cForegroundColor
```

Arguments

<cColor>

A character string holding a color value consisting of foreground and background color.

Return

The function returns the foreground color of the passed color value as a character string.

Note: refer to function SetColor() for an explanation how colors are encoded in a character string.

Info

See also:	GetClrBack(), HB_ColorToN(), SetColor()
Category:	Screen functions
Source:	rtl\color53.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays results of GetClrFore()

   PROCEDURE Main
      LOCAL cColor1 := "W+/B"
      LOCAL cColor2 := "N/BG"

      ? GetClrFore( cColor1 )   // result: W+

      ? GetClrFore( cColor2 )   // result: N
   RETURN
```

GetClrPair()

Extracts a color value from a color string.

Syntax

```
GetClrPair( <cColorString>, <nPos> ) --> cColorValue
```

Arguments

<cColorString>

This is a SetColor() compliant character string holding color values as a comma separated list.

<nPos>

This numeric value specifies the ordinal position of the color value to extract from *<cColorString>*. The first color value in the color string has the ordinal position 1.

Return

The function returns the color value at position *<nPos>* in the passed color string as a character string. If *<cColorString>* has less than *<nPos>* colors, the return value is a null string ("").

Info

See also:	GetPairLen(), GetPairPos(), SetClrPair(), SetColor()
Category:	Screen functions
Source:	rtl\color53.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example extracts all color values from a color
// string holding eight color values.

   PROCEDURE Main
      LOCAL i, cColor := "W/R,W/G,W/B,W+/R,W+/G,W+/B,W/BG,W+/BG"

      FOR i:=1 TO 8
         ? GetClrPair( cColor, i )
      NEXT

      ** output
      // W/R
      // W/G
      // W/B
      // W+/R
      // W+/G
      // W+/B
      // W/BG
      // W+/BG
   RETURN
```

GetCurrentThread()

Retrieves the handle of the current thread.

Syntax

```
GetCurrentThread() --> pThreadHandle
```

Return

The function returns the handle of the thread that executes GetCurrentThread().

Description

Function GetCurrentThread() is used to retrieve the thread handle of the current thread. Thread handles are created by function StartThread(). They are required for other multi-threading functions like JoinThread() or StopThread().

Info

See also:	GetThreadID(), GetSystemThreadID(), HB_MutexCreate(), IsSameThread(), JoinThread(), StopThread(), ThreadSleep()
Category:	Multi-threading functions, xHarbour extensions
Source:	vm\thread.c
LIB:	xhbmt.lib
DLL:	xhbmtdll.dll

GetDefaultPrinter()

Retrieves the name of a computer's default printer.

Syntax

```
GetDefaultPrinter() --> cPrinterName
```

Return

The function returns a character string holding the name of the default printer.

Description

This function serves informational purposes. It retrieves the name of the printer selected as default printer for the current user in the system configuration.

Info

See also:	GetPrinters(), PrintFileRaw(), PrinterExists(), PrinterPortToName()
Category:	Printer functions, xHarbour extensions
Source:	rtl\tprinter.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
PROCEDURE Main()
   LOCAL cPrinter := GetDefaultPrinter()
       ? cPrinter
RETURN
```

GetDoSetKey()

Evaluates a SetKey() code block during READ.

Syntax

```
GetDoSetKey( <bBlock>, <oGet> ) --> NIL
```

Arguments

<bBlock>

This is a SetKey() code block associated with a key pressed during READ.

<oGet>

This must be the Get object that has input focus. It is determined with GetActive().

Return

The return value is always NIL.

Description

GetDoSetKey() is a utility function of the Get system used for default processing of code blocks associated with a key. It is automatically called within the Get system.

Info

See also:	@...GET, Get(), GetApplyKey(), GetActive(), READ, SetKey()
Category:	Get system
Source:	rtl\getsys.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

GetE()

Retrieves an operating system environment variable.

Syntax

```
GetE( <cEnvVar> ) --> cString
```

Arguments

<cEnvVar>

A character string with the name of the environment variable to retrieve. The name is not case-sensitive.

Return

GetE() is a short form of GetEnv() and returns the same information.

Info

See also:	CurDir(), CurDrive(), File(), GetEnv(), SET DEFAULT, SET PATH
Category:	Environment functions
Source:	rtl\gete.c
LIB:	xhb.lib
DLL:	xhbdll.dll

GetEnv()

Retrieves an operating system environment variable.

Syntax

```
GetEnv( <cEnvVar> ) --> cString
```

Arguments

<cEnvVar>

A character string with the name of the environment variable to retrieve. The name is not case-sensitive.

Return

The function returns the contents of the environment variable *<cEnvVar>* as a character string, or a null string ("") when the environment variable does not exist.

Description

The GetEnv() function queries the environment variables defined for the operating system command shell that has started the xHarbour application. Environment variables are defined using the SET command of the command shell.

Info

See also: CurDir(), CurDrive(), File(), SET DEFAULT, SET PATH
Category: Environment functions
Source: rtl\gete.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example retrieves some environment variables common for
// a programming environment:

   PROCEDURE Main
      ? GetEnv( "INCLUDE" )          // SET INCLUDE=path

      ? GetEnv( "LIB" )              // SET LIB=path

      ? GetEnv( "COMSPEC" )          // Command shell
   RETURN
```

GetFldCol()

Returns the screen column position of a Get field.

Syntax

```
GetFldCol( [<nGetPos>] ) --> nScreenColumn
```

Arguments

<nGetPos>

This numeric parameter specifies the ordinal position of a Get field in the PUBLIC *Getlist* array. It defaults to the return value of CurrentGet().

Return

The function returns the screen column position of the Get field stored in the element *<nGetPos>* of the *Getlist* array as a numeric value. If *<nGetPos>* does not identify an element of the *Getlist* array, the return value is -1.

Info

See also:	@...GET, CountGets(), CurrentGet(), GetFldRow()
Category:	CT:GetSys, Get system
Source:	ct\getinfo.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

GetFldRow()

Returns the screen row position of a Get field.

Syntax

```
GetFldRow( [<nGetPos>] ) --> nScreenRow
```

Arguments

`<nGetPos>`

This numeric parameter specifies the ordinal position of a Get field in the PUBLIC *Getlist* array. It defaults to the return value of CurrentGet().

Return

The function returns the screen row position of the Get field stored in the element *<nGetPos>* of the *Getlist* array as a numeric value. If *<nGetPos>* does not identify an element of the *Getlist* array, the return value is -1.

Info

See also:	@...GET, CountGets(), CurrentGet(), GetFldCol()
Category:	CT:GetSys, Get system
Source:	ct\getinfo.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

GetFldVar()

Returns the name of a Get variable.

Syntax

```
GetFldVar( [<nGetPos>] ) --> cGetVarName
```

Arguments

<nGetPos>

This numeric parameter specifies the ordinal position of a Get field in the PUBLIC *Getlist* array. It defaults to the return value of CurrentGet().

Return

The function returns the name of the edited variable of the Get field stored in the element *<nGetPos>* of the *Getlist* array as a numeric value. If *<nGetPos>* does not identify an element of the *Getlist* array, the return value is -1.

Info

See also:	@...GET, CountGets(), CurrentGet(), GetFldRow()
Category:	CT:GetSys, Get system
Source:	ct\getinfo.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

GetLastError()

Retrieves the error code of the last dynamically called DLL function.

Syntax

```
GetLastError() --> nErrorCode
```

Return

The function returns the last error code that occurred with a DllCall() invoked DLL function.

Description

GetLastError() is used when a Windows API function is invoked via DllCall(). Typically, a WinAPI function declared as BOOL returns 0 on failure. In this case, GetLastError() can be called to retrieve the error code for extended error information.

The last error code can be set with function SetLastError().

Info

See also:	CallDll(), DllCall(), LoadLibrary(), SetLastError()
Category:	DLL functions, xHarbour extensions
Source:	rtl\dllcall.c
LIB:	xhb.lib
DLL:	xhbdll.dll

GetNew()

Creates a new Get object.

Syntax

```
GetNew( [<nRow>]       , ;
        [<nCol>]       , ;
        <bVarBlock>    , ;
        [<cVarName>]   , ;
        [<cPicture>]   , ;
        [<cColorSpec>]  ) --> oGet
```

Arguments

<nRow>

This is the numeric row coordinate on the screen where the Get object is displayed. It defaults to the return value of function Row(). *<nRow>* is assigned to oGet:row.

<nCol>

This is the numeric column coordinate on the screen where the Get object is displayed. It defaults to the return value of function Col(). *<nCol>* is assigned to oGet:col.

<bVarBlock>

This is the data code block connecting a Get object with a variable holding the edited value (see description below). *<bVarBlock>* is assigned to oGet:block.

<cVarName>

This is a character string holding the symbolic name of a memory variable of PRIVATE or PUBLIC scope. If the Get object should edit such a variable, *<bVarBlock>* is optional. *<cVarName>* defaults to an empty string ("") and is assigned to oGet:name.

<cPicture>

This is a character string holding a PICTURE format. *<cPicture>* defaults to an empty string ("") and is assigned to oGet:picture.

<cColorSpec>

This is a character string holding a color string of up to four color values. (refer to function SetColor() for color values). *<cColorSpec>* is assigned to oGet:colorSpec.

Return

Function GetNew() returns a new, initialized Get object.

Description

Function GetNew() is the functional equivalent of Get():new(). Refer to the description of the Get object.

Info

See also:	Get()
Category:	Object functions, UI functions
Source:	rtl\tgetint.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

GetPairLen()

Returns the length of a color value within a color string.

Syntax

```
GetPairLen( <cColorString>, <nPos> ) --> nLength
```

Arguments

`<cColorString>`

This is a SetColor() compliant character string holding color values as a comma separated list.

`<nPos>`

This numeric value specifies the ordinal position of the color value to find in *<cColorString>*. The first color value in the color string has the ordinal position 1.

Return

The function returns the length of th color value at position *<nPos>* in the passed color string as a numeric value.

Info

See also: GetClrPair(), GetPairPos(), SetColor()
Category: Screen functions
Source: rtl\color53.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example lists the length of all color values
// in a color string of eight color values.

   PROCEDURE Main
      LOCAL i, cColor := "W/R,W/G,W/B,W+/R,W+/G,W+/B,W/BG,W+/BG"

      FOR i:=1 TO 8
         ? GetPairLen( cColor, i )
      NEXT

      ** output
      //   3
      //   3
      //   3
      //   4
      //   4
      //   4
      //   4
      //   5
   RETURN
```

GetPairPos()

Returns the absolute position of a color value in a color string.

Syntax

```
GetPairPos( <cColorString>, <nPos> ) --> nAbsPos
```

Arguments

<cColorString>

This is a SetColor() compliant character string holding color values as a comma separated list.

<nPos>

This numeric value specifies the ordinal position of the color value to find in *<cColorString>*. The first color value in the color string has the ordinal position 1.

Return

The function returns the absolute position of the color value at the ordinal position *<nPos>* in the passed color string as a numeric value.

Info

Category:	Screen functions
Source:	rtl\color53.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example uses function SubStr() to extract a color
// value from a color string

   PROCEDURE Main
      LOCAL cColor := "W/R,W/G,W/B,W+/R,W+/G,W+/B,W/BG,W+/BG"
      LOCAL nAbsPos, nLength

      nAbsPos := GetPairPos( cColor, 6 )
      nLength := GetPairLen( cColor, 6 )

      ? SubStr( cColor, nAbsPos, nLength )  // result: W+/B

      ? GetClrPair( cColor, 6 )             // result: W+/B
   RETURN
```

GetPostValidate()

Validates data after editing.

Syntax

```
GetPostValidate( <oGet> ) --> lDataIsValid
```

Arguments

<oGet>

This must be the Get object that has input focus. It is determined with GetActive().

Return

The function returns .T. (true) if the edited value is valid before the currently active Get looses input focus, otherwise .F. (false) is returned.

Description

GetPostValidate() is a utility function of the Get system used for validating edited data. It is automatically called within the Get system while READ is active. The rules for data validation are defined with the VALID or RANGE option of the @...GET command, or via a code block assigned to oGet:postBlock. When data validation fails, the input focus remains with the current Get entry field.

Info

See also:	@...GET, Get(), GetPreValidate(), READ
Category:	Get system
Source:	rtl\getsys.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

GetPrec()

Retrieves computing precision for trigonometric functions.

Syntax

```
GetPrec() --> nDecimalPlaces
```

Return

The function returns a numeric value. It identifies the accuracy of trigonometric functions as the number of decimal places that must be calculated before a trigonometric function returns its result. It can be defined with SetPrec().

Info

See also:	SetPrec()
Category:	CT:Math, Mathematical functions, Trigonometric functions
Source:	ct\ctmath.c
LIB:	xhb.lib
DLL:	xhbdll.dll

GetPreValidate()

Validates data before editing.

Syntax

```
GetPreValidate( <oGet> ) --> lCanEdit
```

Arguments

<oGet>

This is a Get object that is about to receive input focus.

Return

The function returns .T. (true) if *<oGet>* may receive input focus, otherwise .F. (false) is returned.

Description

GetPreValidate() is a utility function of the Get system used for validating if a Get object may receive input focus. It is automatically called within the Get system while READ is active. The rules for pre validation are defined with the WHEN option of the @...GET command, or via a code block assigned to oGet:preBlock. When pre validation fails, the next Get is searched that may receive input focus, and *<oGet>* does not receive focus.

Info

See also:	@...GET, Get(), GetPostValidate(), READ
Category:	Get system
Source:	rtl\getsys.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

GetPrinters()

Retrieves information about available printers.

Syntax

```
GetPrinters( [<lPortInfo>], [<lLocalPrinters>] ) --> aPrinterInfo
```

Arguments

`<lPortInfo>`

The default value for *<lPortInfo>* is .F. (false). If set to .T. (true), the function includes port information in the returned array.

`<lLocalPrinters>`

The default value for *<lLocalPrinters>* is .F. (false). If set to .T. (true), the function returns only information for local printers.

Return

The function returns a one- or two-dimensional array. If *<lPortInfo>* is .F. (false), a one-dimensional array is returned. Each element contains a character string with the name of an available printer. If *<lPortInfo>* is .T., each element of the returned array is an array of four elements, holding character strings with additional printer information:

Array elements for additional printer information

Element	Description
1	Printer name
2	Port name
3	Printer type (e.g. Local/Network)
4	Printer driver

Description

The GetPrinters() function obtains information about printers that are currently available. If it is called without arguments, the function returns the names of all available printers. The list can be reduced to local printers only by specifying .T. (true) for *<lLocalPrinters>*. Additional information about port and printer driver can be retrieved when *<lPortInfo>* is set to .T. (true)

Info

See also: GetDefaultPrinter(), PrintFileRaw(), PrinterExists(), PrinterPortToName()
Category: Printer functions, xHarbour extensions
Source: rtl\tprinter.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example is a test program for Windows printing

   PROCEDURE Main
      LOCAL aPrinter, i
      CLS
      ?
      ? Set(_SET_DEVICE)

      aPrinter := GetPrinters()
```

```
   IF Empty( aPrinter )
      ? '----- No Printers installed'
      QUIT
   ENDIF

   SET PRINTER TO ( GetDefaultPrinter() )
   ? Set( _SET_PRINTER   )
   ? Set( _SET_PRINTFILE )
   SET CONSOLE OFF
   SET PRINTER ON
   ? 'Default Printer'
   ?
   ? GetDefaultPrinter()
   ?
   ? 'Printers available'
   ?'------------------'
   FOR i:= 1 TO Len( aPrinter )
     ? aPrinter[i]
   NEXT i

   aPrinter:= GetPrinters( .T. )
   ?
   ? 'Printers and Ports'
   ? '------------------'
   FOR i:= 1 TO Len( aPrinter )
      ? aPrinter[i,1],'on', aPrinter[i,2]
   NEXT

   EJECT
   SET PRINTER OFF
   SET CONSOLE ON
   SET PRINTER TO

   ? Set( _SET_PRINTER )
   ? Set( _SET_DEVICE  )
   WAIT
RETURN
```

GetProcAddress()

Retrieves the memory address of a function in a dynamically loaded DLL.

Syntax

```
GetProcAddress( <nDllHandle>, ;
                <cFuncName>|<nOrdinal> ) --> pAddress
```

Arguments

<nDllHandle>

This is a numeric handle of the DLL that contains the DLL function. It is the return value of LoadLibrary().

<cFuncName>

This is a character string holding the symbolic name of the function to obtain the memory address for. Unlike regular xHarbour functions, this function name is **case sensitive**.

<nOrdinal>

Instead of the symbolic function name, the numeric ordinal position of the function inside the DLL file can be passed.

Return

The function returns the memory address of *<cFuncName>* as a pointer. When the function cannot be found in the DLL, a null pointer is returned.

Description

Function GetProcAddress() obtains the memory address of a DLL function after the DLL is loaded into memory with LoadLibrary(). The memory address of a function, also called function pointer, is sometimes required by API functions when they need to call other functions.

Note: the pointer returned by GetProcAddress() can be passed to function CallDll() which executes the function.

Info

See also: CallDll(), DllCall(), DllPrepareCall(), LoadLibrary()
Category: DLL functions, xHarbour extensions
Source: rtl\dllcall.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates case sensitivity of function names
// with DLL functions.

   PROCEDURE Main
      LOCAL nDll := LoadLibrary( "Kernel32.dll" )

      ? GetProcAddress( nDll, "MultiByteToWideChar" )  // result: 7c809cad

      ? GetProcAddress( nDll, "MultibyteToWideChar" )  // result: 00000000

      FreeLibrary( nDll )
   RETURN
```

GetRegistry()

Retrieves the value of a registry entry

Syntax

```
GetRegistry( <nHKEY>, <cRegPath>, <cRegKey> ) --> xRegValue
```

Arguments

`<nHKEY>`

This numeric parameter specifies the root entry in the Windows registry to search for a registry key. #define constants are available in the Winreg.ch file that can be used for *<nHKEY>*. They begin the the prefix HKEY_.

`<cRegPath>`

This is a character string holding the search path in the registry to locate *<cRegKey>*. The path must include a backslash as delimiter, if required, but may neither begin or end with a backslash.

`<cRegKey>`

This is a character string holding the name of the registry key to retrieve the value for. *<cRegKey>* must be located underneath *<cRegPath>*.

Return

The function returns the value of the specified registry key or NIL if it does not exist within the given root entry and search path.

Description

The function searches a key in the Windows registry beginning with the root *<nHKEY>*, following the search path *<cRegPath>*. If the key *<cRegKey>* exists in *<cRegPath>*, the value of *<cRegKey>* is returned. If the key cannot be found, the return value is NIL.

Winreg.ch

Winreg.ch adds quite some overhead to an application program by adding structure definitions. If this is not required, Winreg.ch does not need to be #included. GetRegistry() recognizes the following values for *<nHKEY>* in addition to the HKEY_* #define constants:

Registry keys

Registry Key	Equivalent value
HKEY_LOCAL_MACHINE	0
HKEY_CLASSES_ROOT	1
HKEY_CURRENT_USER	2
HKEY_CURRENT_CONFIG	3
HKEY_LOCAL_MACHINE	4
HKEY_USERS	5

Note: on Windows NT, 2000, XP or later, the user may need certain security rights in order to be able to read the registry.

Info

See also: QueryRegistry(), SetRegistry()
Category: Registry functions, xHarbour extensions
Header: Winreg.ch
Source: rtl\winreg.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example reads values from the registry as they exist
// after an xHarbour Builder installation

   * #include "Winreg.ch"           // not needed for this example

   #define HKEY_CURRENT_USER  0     // use alternative #define constant

   PROCEDURE Main
      LOCAL nHKey    := HKEY_CURRENT_USER
      LOCAL cRegPath := "SOFTWARE\xHarbour.com\xHarbour Builder"

      ? GetRegistry( nHKey, cRegPath, "Edition" )   // result: Enterprise

      ? GetRegistry( nHKey, cRegPath, "rootdir" )   // result: C:\xhb

      ? GetRegistry( nHKey, cRegPath, "xhb build" ) // result: October 2006
   RETURN
```

GetSecret()

Allows for hidden Get input for character strings.

Syntax

```
GetSecret( <cInit> , ;
          [<nRow>] , ;
          [<nCol>] , ;
          [<lSay>] , ;
          [<cSay>]   ) --> cString
```

Arguments

<cInit>

This is a character string serving as the initial value for editing.

<nRow>

This is a numeric value in the range of 0 to MaxRow(). It specifies the screen row position for the Get. The default value is Row().

<nCol>

This is a numeric value in the range of 0 to MaxCol(). It specifies the screen column position for the Get. The default value is Col().

<lSay>

This logical parameter defaults to .F. (false). When set to .T. (true), the edited value is displayed when the function returns.

<cSay>

This is an optional character string displayed as prompt in front of the Get field.

Return

The function returns the edited value as a character string.

Description

GetSecret() is a simplified Get routine for hidden input. The user can edit a character string as in a regular Get field, but the characters are not displayed on the screen. Instead, each character is represented by an aserisk (*). This allows for comfortable input of a password, for example.

Info

See also: @...GET
Category: CT:GetSys, Get system
Source: ct\getsecrt.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example shows a typical scenario for GetSecret().
// The user must enter a password which is not displayed on
// the screen

   PROCEDURE Main
      LOCAL cPassword := Space(20)
      CLS
      SET CONFIRM ON
```

```
   cPassWord := ;
     GetSecret( cPassword, 10, 10, .T., "Enter your password: ")

   IF cPassWord <> "xHarbour"
      Alert( "Access denied" )
      QUIT
   ENDIF

   Alert( "Access granted" )
RETURN
```

GetSystemThreadID()

Retrieves the numeric system Thread ID of a thread.

Syntax

```
GetSystemThreadID( [<pThreadHandle>] ) --> nSystemTID
```

Arguments

<pThreadHandle>

This the a handle of the thread to retrieve the system ID for. If omitted, the system thread ID of the current thread is retrieved.

Return

The function returns the numeric ID of a thread as it is used by the operating system.

Description

When a new thread is started, it gets assigned a unique numeric Thread ID (TID) by the operating system. The system TID of the current or a particular thread is returned by GetSystemThreadID(). It is mainly of informational use, since system TIDs are random generated. xHarbour's virtual machine maintains an additional TID which is sequentially generated. See function GetThreadID() for more information.

Info

See also:	GetCurrentThread(), GetThreadID(), HB_MutexCreate(), StartThread(), ThreadSleep()
Category:	Multi-threading functions, xHarbour extensions
Source:	vm\thread.c
LIB:	xhbmt.lib
DLL:	xhbmtdll.dll

GetThreadID()

Retrieves the numeric application Thread ID of a thread.

Syntax

```
GetThreadID( [<pThreadHandle>] ) --> nApplicationTID
```

Arguments

<pThreadHandle>

This the a handle of the thread to retrieve the ID for. If omitted, the thread ID of the current thread is retrieved.

Return

The function returns the numeric ID of a thread as it is used by the xHarbour application.

Description

Function GetThreadID() returns the numeric Thread ID (TID) as it is used by xHarbour's virtual machine. TIDs are unique and are generated sequentially. This guarantees the following conditions:

1. the TID of the Main thread is always 1.
2. the TID of the next thread created is larger than the TID of a previously created thread.
3. if GetThreadID() returns the same TID for two *<pThreadHandle>*s, both *<pThreadHandle>*s refer to the same thread.

Note: if *<pThreadHandle>* does not contain a valid thread handle, a runtime error is raised. Refer to function IsValidThread() to test the validity of a thread handle.

Info

See also:	GetCurrentThread(), GetSystemThreadID(), HB_MutexCreate(), StartThread(), ThreadSleep()
Category:	Multi-threading functions, xHarbour extensions
Source:	vm\thread.c
LIB:	xhbmt.lib
DLL:	xhbmtdll.dll

Example

```
// The example displays the application TID and system TID of
// ten threads.

   PROCEDURE Main
      LOCAL i, aTID := {}

      DisplayTIDs()

      FOR i:=1 TO 9
         StartThread( "GetTIDs", aTID )
      NEXT

      WaitForThreads()

      AEval( aTID, {|c| QOut(c) } )
   RETURN
```

```
PROCEDURE GetTIDs( aTID )
   LOCAL th := GetCurrentThread()
   LOCAL cTID

   IF GetThreadID( th ) == 1
      cTID := "Main thread : "
   ELSE
      cTID := "Child thread: "
   ENDIF

   cTID += "  APP TID:" + Str( GetThreadID( th ), 5 )
   cTID += "  SYS TID:" + Str( GetSystemThreadID( th ), 5 )

   AAdd( aTID, cTID )
RETURN
```

GetVolInfo()

Retrieves the volume label of a disk.

Syntax

```
GetVolInfo( <cDrive> ) --> lSuccess
```

Arguments

<cDrive>

A single character, followed by a colon and backslash, specifies the disk drive to query. If omitted, the current drive is queried.

Return

The function returns the volume label of a disk as a character string, or a null string ("") in case of an error.

Info

See also: FileSeek(), VolSerial(), Volume()
Category: CT:DiskUtil, Disks and Drives, xHarbour extensions
Source: ct\disk.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example lists the volume labels of all drives and displays them.

   PROCEDURE Main
      LOCAL cDrive, i, cLabel

      FOR i:=1 TO 26
         cDrive := Chr(64+i) + ":\"
         cLabel := GetVolInfo( cDrive )

         IF .NOT. Empty( cLabel )
            ? cDrive, cLabel
         ENDIF
      NEXT
   RETURN
```

GuiApplyKey()

Sends an Inkey() code to the associated control of the currently active Get object.

Syntax

```
GuiApplyKey( <oGet>, <nKey>, <oMenu>, <oGetMsg> ) --> NIL
```

Arguments

`<oGet>`

This must be the Get object that has input focus. It is determined with GetActive().

`<nKey>`

This is the numeric Inkey() code of the key to send to the Get object. Refer to the file Inkey.ch for #define constants that can be used for *<nKey>*.

`<oMenu>`

Optionally, a TopBarMenu() object can be specified. In this case, the menu is activated when *<nKey>* is a menu hot key.

`<oGetMsg>`

Optionally, a GetMssgLine() object can be specified. It displays a message associated with *<oGet>* in the status line.

Return

The return value is always NIL.

Description

GuiApplyKey() is a utility function of the Get system used for default processing of key strokes. It is automatically called within the Get system.

Note: the prefix **Gui** comes from Clipper 5.3 and may be misleading. It has nothing to do with a Graphical User Interface. All **Gui** functions of the Get system operate with Get objects associated with text mode controls that mimick true GUI controls, such as menus, check boxes or radio buttons, for example.

Info

See also: @...GET, Get(), GetActive(), GuiReader(), READ
Category: Get system
Header: Inkey.ch
Source: rtl\getsys.prg
LIB: xhb.lib
DLL: xhbdll.dll

GuiGetPostValidate()

Validates data after editing.

Syntax

```
GuiGetPostValidate( <oGet>, <oControl>, [<oGetMsg>] ) --> lDataIsValid
```

Arguments

`<oGet>`

This must be the Get object that has input focus. It is determined with GetActive().

`<oControl>`

This ia an additional text mode Control object associated with the Get object, such as a HBCheckBox() or HBListBox() object.

`<oGetMsg>`

Optionally, a GetMssgLine() object can be specified. It displays a message associated with *<oGet>* in the status line.

Return

The function returns .T. (true) if the edited value is valid before the currently active Get looses input focus, otherwise .F. (false) is returned.

Description

GuiGetPostValidate() is a utility function of the Get system used for validating edited data. It is automatically called within the Get system while READ is active. The rules for data validation are defined with the VALID or RANGE option of the @...GET command, or via a code block assigned to oGet:postBlock. When data validation fails, the input focus remains with the current Get entry field, or associated control, respectively.

Note: the prefix **Gui** comes from Clipper 5.3 and may be misleading. It has nothing to do with a Graphical User Interface. All **Gui** functions of the Get system operate with Get objects associated with text mode controls that mimick true GUI controls, such as menus, check boxes or radio buttons, for example.

Info

See also:	@...GET, Get(), GuiGetPreValidate(), READ
Category:	Get system
Source:	rtl\getsys.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

GuiGetPrevalidate()

Validates data before editing.

Syntax

```
GuiGetPreValidate( <oGet>, <oControl>, [<oGetMsg>] ) --> lCanEdit
```

Arguments

`<oGet>`

This is a Get object that is about to receive input focus.

`<oControl>`

This ia an additional text mode Control object associated with the Get object, such as a HBCheckBox() or HBListBox() object.

`<oGetMsg>`

Optionally, a GetMssgLine() object can be specified. It displays a message associated with *<oGet>* in the status line.

Return

The function returns .T. (true) if *<oGet>* may receive input focus, otherwise .F. (false) is returned.

Description

GuiGetPreValidate() is a utility function of the Get system used for validating if a Get object may receive input focus. It is automatically called within the Get system while READ is active. The rules for pre validation are defined with the WHEN option of the @...GET command, or via a code block assigned to oGet:preBlock. When pre validation fails, the next Get is searched that may receive input focus, and *<oGet>*, or its associated control, does not receive focus.

Note: the prefix **Gui** comes from Clipper 5.3 and may be misleading. It has nothing to do with a Graphical User Interface. All **Gui** functions of the Get system operate with Get objects associated with text mode controls that mimick true GUI controls, such as menus, check boxes or radio buttons, for example.

Info

See also:	@...GET, Get(), GuiGetPostValidate(), READ
Category:	Get system
Source:	rtl\getsys.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

GuiReader()

Processes user input.

Syntax

```
GuiReader( <oGet>, <oGetlist>, [<oMenu>], [<oGetMsg>] ) --> NIL
```

Arguments

<oGet>

This is the Get object that receives input focus first.

<oGetList>

This is the HBGetList() object maintaining the Get entry fields. It is created in ReadModal().

<oMenu>

Optionally, a TopBarMenu() object can be specified. In this case, the menu can be activated by the user.

Return

The return value is always NIL

Description

GuiReader() is a utility function of the Get system. It implements the standard READ behavior of individual Get objects and associated controls. User input is obtained with the Inkey() function.

Note: the prefix **Gui** comes from Clipper 5.3 and may be misleading. It has nothing to do with a Graphical User Interface. All **Gui** functions of the Get system operate with Get objects associated with text mode controls that mimick true GUI controls, such as menus, check boxes or radio buttons, for example.

Info

See also: @...GET, Get(), GuiApplyKey(), READ
Category: Get system
Source: rtl\getsys.prg
LIB: xhb.lib
DLL: xhbdll.dll

HaaDelAt()

Removes a key/value pair from an associative array.

Syntax

```
HaaDelAt( <hArray>, <nPos> ) --> NIL
```

Arguments

<hHash>

A variable referencing the associative array to remove a key/value pair from.

<nPos>

A numeric value specifying the ordinal position of the key/value pair to remove. It must be in the range between 1 and Len(*<hArray>*).

Return

The function returns always NIL.

Description

The function removes a key/value pair from the associative array *<hArray>* by its ordinal position. If *<nPos>* is outside the valid range, a runtime error is raised. Use function HaaGetPos() to determine the ordinal position of a key.

Info

See also:	Array(), HaaGetKeyAt(), HaaGetValueAt(), HaaSetValueAt(), Hash(), HSetAACompatibility()
Category:	Associative arrays, Hash functions, xHarbour extensions
Source:	vm\hash.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example creates an associative array and deletes
// the third key/value pair.

   PROCEDURE Main
      LOCAL hArray := Hash()

      HSetAACompatibility( hArray, .T. )

      hArray[ "One"   ] := 10
      hArray[ "Two"   ] := 20
      hArray[ "Three" ] := 30
      hArray[ "Four"  ] := 40
      hArray[ "Five"  ] := 50

      ? hArray[ 3 ]               // result: 30
      ? hArray["Four"]            // result: 40

      HaaDelAt( hArray, 3 )

      ? hArray[ 3 ]               // result: 40
      ? hArray["Four"]            // result: 40
   RETURN
```

HaaGetKeyAt()

Retrieves the key from an associative array by its ordinal position.

Syntax

```
HaaGetKeyAt( <hArray>, <nPos> ) --> xKey
```

Arguments

<hArray>

A variable referencing the associative array to retrieve a key from.

<nPos>

A numeric value specifying the ordinal position of the key/value pair to query. It must be in the range between 1 and Len(*<hArray>*).

Return

The function returns the key at position *<nPos>* in the associative array *<hArray>*.

Description

This function retrieves the key from the associative array *<hArray>* at position *<nPos>*. If *<nPos>* is outside the valid range, a runtime error is raised. Use function HaaGetPos() to determine the ordinal position of a key.

The keys are inserted into the associative array by their sorting order and cannot be moved or changed.

Info

See also:	Array(), HaaGetValueAt(), HaaSetValueAt(), Hash(), HSetAACompatibility()
Category:	Associative arrays, Hash functions, xHarbour extensions
Source:	vm\hash.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates that keys are collected in an
// associative array by their sorting order.

   PROCEDURE Main
      LOCAL hArray := Hash()

      HSetAACompatibility( hArray, .T. )

      hArray[ "One"   ] := 10
      hArray[ "Two"   ] := 20
      hArray[ "Three"] := 30
      hArray[ "Four" ] := 40
      hArray[ "Five" ] := 50

      ? HGetKeyAt( hArray, 1 )    // result: Five
      ? HGetKeyAt( hArray, 5 )    // result: Two

      ? hArray[1]                 // result: 10
      ? hArray[5]                 // result: 50
   RETURN
```

HaaGetPos()

Retrieves the ordinal position of a key in an associative array.

Syntax

```
HaaGetPos( <hArray>, <xKey> ) --> nPos
```

Arguments

<hArray>

A variable referencing the associative array to search a key in.

<xKey>

This is the key to search for in *<hArray>*.

Return

The function returns the ordinal position of *<xKey>* in the associative array *<hArray>*, or 0 if the key is not present.

Description

This function is mostly used to check if a key is present in an associative array. If the return value is greater than zero, the key exists. The return value can then be passed on to function HaaGetValueAt() to retrieve the associated value. This is more efficient than using the key a second time for retrieving the value.

Info

See also:	Array(), HaaGetKeyAt(), HaaGetRealPos(), HaaGetValueAt(), HaaSetValueAt(), Hash(), HSetAACompatibility()
Category:	Associative arrays, Hash functions, xHarbour extensions
Source:	vm\hash.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example creates an associative array and retrieves values
// from it using a function and array notation.

   PROCEDURE Main
      LOCAL hArray := Hash()

      HSetAACompatibility( hArray, .T. )

      hArray[ "One"   ] := 10
      hArray[ "Two"   ] := 20
      hArray[ "Three"] := 30
      hArray[ "Four" ] := 40
      hArray[ "Five" ] := 50

      ? HaaGetPos( hArray, "Four" )    // result: 4
      ? HaaGetPos( hArray, "Five" )    // result: 5
      ? HaaGetPos( hArray, "Six"  )    // result: 0

      ? HaaGetValueAt( hArray, 4 )     // result: 40
      ? HaaGetValueAt( hArray, 5 )     // result: 50

      ? hArray[4]                      // result: 40
```

```
   ? hArray[5]                    // result: 50
RETURN
```

HaaGetRealPos()

Retrieves the sort order of a key in an associative array.

Syntax

```
HaaGetRealPos( <hArray>, <nPos> ) --> nSortOrder
```

Arguments

<hArray>

A variable referencing the associative array to search a key in.

<nPos>

A numeric value specifying the ordinal position of the key/value pair to query. It must be in the range between 1 and Len(*<hArray>*).

Return

The function returns a numeric value representing the sort order of *<xKey>* in the associative array *<hArray>*, or 0 if the key does not exist.

Description

Keys are internally held in an associative array by their sorting order. Function HaaGetRealPos() is used to determine this sorting order of a key in an associative array. The ordinal position of a key is returned by HaaGetPos().

Info

See also:	Array(), HaaGetKeyAt(), HaaGetPos(), HaaGetValueAt(), HaaSetValueAt(), Hash(), HGetVaaPos(), HSetAACompatibility()
Category:	Associative arrays, Hash functions, xHarbour extensions
Source:	vm\hash.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example outlines the difference of the ordinal position
// of a key and its sorting order in an associative array

   PROCEDURE Main
      LOCAL hArray := Hash(), i, nReal

      HSetAACompatibility( hArray, .T. )

      hArray[ "One"   ] := 10
      hArray[ "Two"   ] := 20
      hArray[ "Three"] := 30
      hArray[ "Four" ] := 40
      hArray[ "Five" ] := 50

      FOR i:=1 TO Len( hArray )
         ? i, HGetKeyAt( hArray, i )
      NEXT
      ** Output:
      // 1 Five
      // 2 Four
      // 3 One
      // 4 Three
```

```
   // 5 Two

   FOR i:=1 TO Len( hArray )
      nReal := HaaGetRealPos( hArray, i )
      ? nReal, HGetKeyAt( hArray, nReal )
   NEXT
   ** Output:
   // 3 One
   // 5 Two
   // 4 Three
   // 2 Four
   // 1 Five
RETURN
```

HaaGetValueAt()

Retrieves the value from an associative array by its ordinal position.

Syntax

```
HaaGetValueAt( <hArray>, <nPos> ) --> xValue
```

Arguments

<hArray>

A variable referencing the associative array to retrieve a value from.

<nPos>

A numeric value specifying the ordinal position of the value to retrieve. It must be in the range between 1 and Len(<*hArray*>).

Return

The function returns the value at position <*nPos*> in the associative array <*hArray*>.

Description

This function retrieves the value from the associative array <*hArray*> at position <*nPos*>. If <*nPos*> is outside the valid range, a runtime error is raised. Use function HaaGetPos() to determine the ordinal position of a key/value pair.

Values of key/value pairs can be changed by specifying the key for an 4associative array and assigning a new value, or by using function HaaSetValueAt() which accepts the ordinal position of the value to change.

Info

See also:	Array(), HaaGetKeyAt(), HaaGetPos(), HaaSetValueAt(), Hash(), HSetAACompatibility()
Category:	Associative arrays, Hash functions, xHarbour extensions
Source:	vm\hash.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example outlines the possibilities of retrieving
// values from an associative array.

   PROCEDURE Main
      LOCAL hArray := Hash()

      HSetAACompatibility( hArray, .T. )

      hArray[ "One"   ] := 10
      hArray[ "Two"   ] := 20
      hArray[ "Three"] := 30
      hArray[ "Four" ] := 40
      hArray[ "Five" ] := 50

      ? HaaGetValueAt( hArray, 1 )   // result: 10
      ? HaaGetValueAt( hArray, 5 )   // result: 50

      ? hArray[1]                    // result: 10
      ? hArray[5]                    // result: 50
```

```
   ? hArray["One"]               // result: 10
   ? hArray["Five"]              // result: 50
RETURN
```

HaaSetValueAt()

Changes the value in an associative array by its ordinal position.

Syntax

```
HaaSetValueAt( <hArray>, <nPos>, <xValue> ) --> NIL
```

Arguments

<hArray>

A variable referencing the associative array to change a value in.

<nPos>

A numeric value specifying the ordinal position of the value to change. It must be in the range between 1 and Len(*<hArray>*).

<xValue>

<xValue> is the new value to assign at position *<nPos>* in *<hArray>*.

Return

The function returns always NIL.

Description

This function changes the value of the associative array *<hArray>* at position *<nPos>*. If *<nPos>* is outside the valid range, a runtime error is raised. Use function HaaGetPos() to determine the ordinal position of a key/value pair.

Info

See also:	Array(), HaaGetKeyAt(), HaaGetPos(), HaaGetValueAt(), Hash(), HSetAACompatibility()
Category:	Associative arrays, Hash functions, xHarbour extensions
Source:	vm\hash.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example outlines the possibilities of changing
// values in an associative array.

   PROCEDURE Main
      LOCAL hArray := Hash()

      HSetAACompatibility( hArray, .T. )

      hArray[ "One"   ] := 10
      hArray[ "Two"   ] := 20
      hArray[ "Three"] := 30
      hArray[ "Four" ] := 40
      hArray[ "Five" ] := 50

      HaaSetValueAt( hArray, 1, 100 )
      hArray["Two"] := 200
      hArray[3]     := 300

      ? hArray["One"]                  // result: 100
      ? HaaGetValueAt( hArray, 2 )     // result: 200
```

```
   ? hArray[3]                        // result: 300
RETURN
```

HAllocate()

Pre-allocates memory for a large hash.

Syntax

```
HAllocate( <hHash>, <nCount> ) --> NIL
```

Arguments

<hHash>

A variable referencing an empty hash to pre-allocate memory for.

<nCount>

A numeric value specifying the number of key/value pairs to reserve memory for.

Return

The function returns always NIL.

Description

The function is used to pre-allocate memory for an empty hash or to reduce its pre-allocated size. When the number of key/value pairs to collect in a hash is approximately known in advance, it can be useful to pre-allocate memory for it so that the time to grow a hash and allocate memory for it is reduced to a minimum. To reduce memory pre-allocated in excess, call HAllocate() with a small value for *<nCount>*.

The function is provided in case huge hashes must be built and performance is at stake. Normally, the partitioning of hashes using HSetPartition() is more effective in creating large hashes.

Info

See also: Hash(), HSetPartition()
Category: Hash functions, xHarbour extensions
Source: vm\hash.c
LIB: xhb.lib
DLL: xhbdll.dll

HardCR()

Replaces soft carriage returns with hard CRs in a character string.

Syntax

```
HardCR( <cString> ) --> cConvertedString
```

Arguments

`<cString>`

A character string to be converted.

Return

The function returns *<cString>* with all soft carriage returns converted to hard carriage returns.

Description

Soft carriage returns (Chr(141)+Chr(10)) are inserted into a string by MemoEdit() when a line wraps during editing. The string returned from MemoEdit() retains soft carriage returns and is usually stored in a memo field. When such a string must be printed or displayed with another function than MemoEdit(), it is necessary to replace soft carriage returns with hard carriage returns (Chr(13)+Chr(10)) since soft carriage returns are not interpreted as end of line characters.

Note: when a memo field is output using a proportional font, use MemoTran() to replace soft carriage returns with a space character.

Info

See also:	?|??, @...SAY, MemoTran(), StrTran()
Category:	Character functions, Memo functions
Source:	rtl\hardcr.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays a memo field previously formatted with the
// automatic word wrapping of MemoEdit().

   PROCEDURE Main
      USE Customer NEW
      ? HardCR( Customer->Notes )
      USE
   RETURN
```

Hash()

Creates a new hash.

Syntax

```
Hash( [<xKey1>, <xValue1> [, <xKeyN>, <xValueN>] ] ) --> hHash
```

Arguments

`<xKey1> .. <xKeyN>`

<xKey1> is the key value associated with *<xValue1>* in the hash. The key value is usually of data type Character, but may be a Date or Numeric value.

`<xValue1> .. <xValueN>`

<xValue1> is a value in the hash which can be accessed through its key value *<xKey1>*.

Return

The function returns a new hash, populated with the specified key/value pairs. If no parameter is passed, an empty hash is returned.

If the function is called with an odd number of arguments, or key values are specified which are not of data type Character, Date or Numeric, the return value is NIL.

Description

Hash variables are usually initialized within a variable declaration using the literal Hash operator \{=>\}. The Hash() function is equivalent to this operator and allows for creating hashes programmatically outside a variable declaration.

Hashes can be populated with key/value pairs by passing an even number of parameters to the function. Data representing the key values must be of orderable data types which restricts them to the data types C, D and N.

Info

See also:	{=>}, HAllocate(), HClone(), HCopy(), HGet(), HDel(), HEval(), HSet(), HSetCaseMatch(), HGetPartition()
Category:	Hash functions, xHarbour extensions
Source:	vm\hash.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example creates hashes using the literal hash operator
// and the Hash() function.

   PROCEDURE Main
      LOCAL hHash1 := { "OPT1" => 10, "OPT2" => 20 }
      LOCAL hHash2, hHash3

      hHash2 := Hash( "OPT2", 200, "OPT3", 300, "OPT4", 400 )

      hHash3 := hHash1 + hHash2

      ? ValToPrg( hHash3 )
      // { "OPT1" => 10, "OPT2" => 200, "OPT3" => 300, "OPT4" => 400 }
   RETURN
```

HbConsoleLock()

Locks the console for the current thread

Syntax

```
HbConsoleLock() --> NIL
```

Return

The return value is always NIL.

Description

The function locks the console for the current thread. While the lock is active, no other thread can produce output on the screen (text mode). The lock must be released with HbConsoleUnlock().

Info

See also: DispBegin(), HbConsoleUnlock()
Category: Multi-threading functions, xHarbour extensions
Source: rtl\console.c
LIB: xhbmt.lib
DLL: xhbdllmt.dll

HbConsoleUnlock()

Releases the console lock.

Syntax

```
HbConsoleUnlock() --> NIL
```

Return

The return value is always NIL.

Description

The function releases a lock of the console previously obtained with HbConsoleLock(). This is required when muliple threads produce screen output simultaneously. A screen lock guarantees that only one thread has access to the console.

Info

See also:	DispBegin(), HbConsoleLock()
Category:	Multi-threading functions, xHarbour extensions
Source:	rtl\console.c
LIB:	xhbmt.lib
DLL:	xhbdllmt.dll

HB_AExpressions()

Parses a character string into an array of macro expressions.

Syntax

```
HB_AExpressions( <cMacro> ) --> aMacroExpressions
```

Arguments

<cMacro>

This is a character string holding a list of macro expressions.

Return

The function returns a one dimensional array holding in its elements individual macro expressions.

Description

Function HB_AExpressions() parses a character string, extracts from it single macro expressions and collects them in the returned array. If the string *<cMacro>* contains multiple expressions, they must be comma separated.

Info

See also:	& (macro operator), HB_MacroCompile(), HB_VmExecute()
Category:	Indirect execution, xHarbour extensions
Source:	vm\arrayshb.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example parses a string holding three macro expressions. The
// the individual macro expressions are displayed and then compiled.

   PROCEDURE Main( ... )
      LOCAL cMacro, aExpr

      PRIVATE nNum  := 10
      PRIVATE aName := { "xHarbour" }

      cMacro := "PCount(),(Valtype(nNum)=='N' .AND. nNum>0), aName[1]"

      aExpr  := HB_AExpressions( cMacro )

      AEval( aExpr, {|c| QOut(c) } )
      ** Output:
      // PCount()
      // (Valtype(nNum)=='N' .AND. nNum>0)
      // aName[1]

      FOR EACH cMacro IN aExpr
         ? &cMacro
      NEXT
      ** Output:
      //  0
      // .F.
      // xHarbour

   RETURN
```

HB_AnsiToOem()

Converts a character string from the ANSI to the OEM character set.

Syntax

```
HB_AnsiToOem( <cANSI_String> ) --> cOEM_String
```

Arguments

`<cANSI_String>`

A character string holding characters from the ANSI character set.

Return

The function returns a string converted to the OEM character set.

Description

The function converts all characters of *<cANSI_String>* to the corresponding characters in the MS-DOS (OEM) character set. Both character sets differ in characters whose ASCII values are larger than 127.

If a character in the ANSI string does not have an equivalent in the OEM character set, it is converted to a similar character of the OEM character set. In this case, it may not be possible to reverse the ANSI => OEM conversion with function HB_OemToAnsi().

Note: all databases created with Clipper store their data using the OEM character set. Displaying data in xHarbour console applications occurs with the OEM character set. Data display in xHarbour GUI applications is done with the Windows ANSI character set.

Info

See also:	HB_OemToAnsi(), HB_SetCodePage()
Category:	Conversion functions, xHarbour extensions
Source:	rtl\oemansi.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates the effect of ANSI<->OEM conversion
// using a character string where the conversion is not reversible.
// The character string contains characters used for line graphics
// in DOS.

   #include "Box.ch"

   PROCEDURE Main
      LOCAL cOEM1 := B_SINGLE
      LOCAL cANSI := HB_OemToAnsi( B_SINGLE )
      LOCAL cOEM2 := HB_ANsiToOem( cANSI )

      DispBox(10, 10, 20, 20, cOEM1 )
      @ Row(), COl() SAY "OEM1"

      DispBox(10, 30, 20, 40, cANSI )
      @ Row(), COl() SAY "ANSI"

      DispBox(10, 50, 20, 60, cOEM2 )
      @ Row(), COl() SAY "OEM2"
```

```
   @ MaxROw()-1, 0
   WAIT
RETURN
```

HB_AParams()

Collects values of all parameters passed to a function, method or procedure.

Syntax

```
HB_AParams() --> aValues
```

Return

The function returns an array holding the values of all parameters passed.

Description

Function HB_AParams() provides a convenient way of collecting all parameters passed to a function, method or procedure with one function call in an array.

Info

See also: PCount(), PValue(), Valtype()
Category: Debug functions, Environment functions, xHarbour extensions
Source: ""
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates how an unknown number of command line
// arguments passed to an xHarbour application can be processed.

   PROCEDURE Main( ... )
      LOCAL aArg := HB_AParams()
      LOCAL cArg

      FOR EACH cArg IN aArg
         DO CASE
         CASE Upper( cArg ) IN ("-H/H-?/?")
            ? "Help requested"

         CASE .NOT. cArg[1] IN ( "-/" )
            ?? " argument:", cArg

         CASE Upper( cArg ) IN ("-X/X")
            ? "Execution requested"

         OTHERWISE
            ? "Unknown:", cArg
         ENDCASE
      NEXT

   RETURN
```

HB_ArgC()

Returns the number of command line arguments.

Syntax

```
HB_ArgC() --> nArgCount
```

Return

The function returns the number of command line arguments.

Description

When an xHarbour application is started, it can be passed arguments from the command line which are received by the initial routine of the application. HB_ArgC() determines this number of arguments. The contents of command line arguments can be retrieved with function HB_ArgV().

Info

See also:	HB_ArgV(), PCount(), PValue()
Category:	Debug functions, Environment functions, xHarbour extensions
Source:	rtl\cmdarg.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example outlines the difference between PCount() and
// HB_ArgC(). Command line arguments can be retrieved in the
// start routine with PCount()/PValue(), or anywhere in a program
// with HB_ArgC()/HB_ArgV()

   PROCEDURE Main( ... )
      LOCAL i

      FOR i:=1 TO PCount()
        ? i, PValue(i)
      NEXT

      ? "-----------"
      Test()
   RETURN


   PROCEDURE Test
      LOCAL i

      FOR i:=1 TO HB_ArgC()
        ? i, HB_ArgV(i)
      NEXT

   RETURN
```

HB_ArgCheck()

Checks if an internal switch is set on the command line.

Syntax

```
HB_ArgCheck( <cSwitch> ) --> lExists
```

Arguments

<cSwitch>

This is a character string holding the symbolic name of the internal command line switch to check.

Return

The function returns .T. (true) when the xHarbour application is started with *<cSwitch>* set on the command line, otherwise .F. (false) is returned.

Description

An xHarbour application can be started with regular and internal command line switches, or arguments. Internal switches are prefixed with two slashes, while regular switches are not prefixed. HB_ArgCheck() tests only the existence of prefixed switches.

Note: use function HB_ArgString() to retrieve the value of an internal command line switch.

Info

See also:	HB_ArgC(), HB_ArgV(), HB_ArgString(), PCount(), PValue()
Category:	Debug functions, Environment functions, xHarbour extensions
Source:	vm\cmdarg.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example tests for the existence of the command line switch TEST.
// The result is only true, when the program is started as follows
// (the command line switch is case insensitive):
//
//   c:\xhb\test.exe  //TEST
//   c:\xhb\test.exe  //test

   PROCEDURE Main( ... )

      ? HB_ArgCheck( "TEST" )

   RETURN
```

HB_ArgString()

Retrieves the vale of an internal switch set on the command line.

Syntax

```
HB_ArgString( <cSwitch> ) --> cValue | NIL
```

Arguments

<cSwitch>

This is a character string holding the symbolic name of the internal command line switch to check.

Return

The function returns the value of the commad line switch as a character string. If *<cSwitch>* is not used at program invocation, the return value is NIL. When it has no value, a null string ("") is returned.

Description

An xHarbour application can be started with regular and internal command line switches, or arguments. Internal switches are prefixed with two slashes, while regular switches are not prefixed. HB_ArgString() retrieves the value of prefixed switches.

Info

See also:	HB_ArgC(), HB_ArgCheck(), HB_ArgV(), PCount(), PValue()
Category:	Debug functions, Environment functions, xHarbour extensions
Source:	vm\cmdarg.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example retrieves the value of the command line switch TEST.
// The value is displayed, when the program is started as follows
// (the command line switch is case insensitive):
//
//   c:\xhb\test.exe  //TEST10
//   c:\xhb\test.exe  //test10

   PROCEDURE Main( ... )

      ? HB_ArgString( "TEST"  )  // result: 10
      ? HB_ArgString( "TEST1" )  // result: 0
      ? HB_ArgString( "TEST2" )  // result: NIL

   RETURN
```

HB_ArgV()

Retrieves the value of a command line argument.

Syntax

```
HB_ArgV( <nPos> ) --> cArgValue
```

Arguments

`<nPos>`

This is a numeric value indicating the ordinal position of the command line argument to check. The value zero is

Return

The function returns the value of the command line argument at position <*nPos*> passed to thexHarbour application as a character string. If <*nPos*> is larger than HB_ArgC(), a null string ("") is returned.

Description

HB_ArgV() is used to retrieve the contents of command line arguments passed to an xHarbour application when it is started. The values are retrieved by their ordinal position and are returned as character strings. To ordinal position zero has a special meaning: it retrieves the name of xHarbour application.

Info

See also:	HB_ArgC(), HB_ArgCheck(), HB_ArgString(), PCount(), PValue()
Category:	Debug functions, Environment functions, xHarbour extensions
Source:	rtl\cmdarg.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example implements the user-defined function AppName()
// which builds the application name upon initial call and
// returns it with or without complete path.

   PROCEDURE Main( ... )
      ? AppName()
      ? AppName(.T.)
   RETURN

   FUNCTION AppName( lFullName )
      STATIC scAppName

      IF scAppName == NIL           // path information
         scAppName := CurDrive() + ":\" + CurDir() + "\"

         scAppName += HB_ArgV(0)    // EXE file name

         IF .NOT. ".exe" IN Lower( scAppName )
            scAppName += ".exe"
         ENDIF
      ENDIF

      IF Valtype( lFullName ) <> "L"
         lFullName := .F.
```

```
   ENDIF

   IF .NOT. lFullName
      RETURN SubStr( scAppName, RAt( "\", scAppName ) + 1 )
   ENDIF

RETURN scAppName
```

HB_ArrayBlock()

Creates a set/get code block for an array.

Syntax

```
HB_ArrayBlock( <aArray>, <nIndex> ) --> bCodeblock
```

Arguments

`<aArray>`

This is a variable holding the array to create s set/get code block for.

`<nElement>`

This is a numeric expression indicating the ordinal position of the array element to access. It must be in the range between 1 and Len(*<aArray>*).

Return

The function returns a code block accessing the array element *<nIndex>* of *<aArray>*. The code block accepts one parameter used to assign a value to the array element.

Description

Function HB_ArrayBlock() creates a code block that accesses one element of an array. It is normally used when a one dimensional array ise edited with Get() objects. The code block can be assigned to Get():block for editing the values stored in the array elements.

Info

See also:	Array(), Get()
Category:	Array functions, xHarbour extensions
Source:	rtl\arrayblk.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_ArrayId()

Returns a unique identifier for an array or an object variable.

Syntax

```
HB_ArrayId( <aArray>|<oObject> ) --> pID
```

Arguments

`<aArray>`

This is an array to retrieve a unique identifier for.

`<oObject>`

Alternatively, an object can be passedThis is an array to retrieve a unique identifier for.

Return

The function returns a pointer value (Valtype()=="P") of the passed array or object. If neither an array nor an object is passed, a null pointer is returned.

Description

Function HB_ArrayID() is provided for debug purposes and returns a unique value for the passed array or object. If two variables hold arrays or objects, and HB_ArrayId() returns the same value for both variables, they reference the same array or object.

Note: the returned identifier is only valid during one program invocation. It cannot be used for identifying persistent data between multiple program invocations.

Info

See also: Array(), HBObject(), HB_ThisArray()
Category: Array functions, Debug functions, Object functions, xHarbour extensions
Source: vm\arrayshb.c
LIB: xhb.lib
DLL: xhbdll.dll

HB_ArrayToStructure()

Converts an array to a binary C structure.

Syntax

```
HB_ArrayToStructure( <aMembers>, ;
                     <aTypes>  , ;
                    [<nAlign>]   ) --> cBinaryStructure
```

Arguments

<aMember>

This is an an array holding the values of the structure members to convert to binary.

<aTypes>

The C-data types to convert each element of *<aMember>* to must be specified as a one dimensional array of the same length as *<aMember>*. #define constants must be used for *<aTypes>*. The following constants are available in the Cstruct.ch file:

Constants for C-data types

Constant	C-data type
CTYPE_CHAR	char
CTYPE_UNSIGNED_CHAR	unsigned char
CTYPE_CHAR_PTR	char *
CTYPE_UNSIGNED_CHAR_PTR	unsigned char*
CTYPE_SHORT	short
CTYPE_UNSIGNED_SHORT	unsigned short
CTYPE_SHORT_PTR	short *
CTYPE_UNSIGNED_SHORT_PTR	unsigned short *
CTYPE_INT	int
CTYPE_UNSIGNED_INT	unsigned int
CTYPE_INT_PTR	int *
CTYPE_UNSIGNED_INT_PTR	unsigned int *
CTYPE_LONG	long
CTYPE_UNSIGNED_LONG	unsigned long
CTYPE_LONG_PTR	long *
CTYPE_UNSIGNED_LONG_PTR	unsigned long *
CTYPE_FLOAT	float
CTYPE_FLOAT_PTR	float *
CTYPE_DOUBLE	double
CTYPE_DOUBLE_PTR	double *
CTYPE_VOID_PTR	void *
CTYPE_STRUCTURE	struct
CTYPE_STRUCTURE_PTR	struct *

<nAlign>

Optionally, the byte alignment for C-structure members can be specified. By default, the members are aligned at an eight byte boundary. Note that Windows API functions require a four byte alignment for structures.

Return

The function returns a character string holding the values of structure members in binary form.

Description

Function HB_ArrayToStructure() converts an array into a binary string representation of a C structure. The returned string contains the values of structure members in binary form and can be passed to external API functions via DllCall(). If the structure is an "out" parameter of the API function, it must be passed by reference and can be converted back to an array using HB_StructureToArray().

HB_ArrayToStructure() is appropriate for simple structures. It is recommended to declare a structure class using typedef struct when a structure is complex, e.g. when it is a nested structure.

Info

See also:	C Structure class, HB_StructureToArray(), pragma pack(), (struct), typedef struct
Category:	C Structure support, xHarbour extensions
Header:	cstruct.ch
Source:	vm\arrayshb.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example simulates the POINT structure which has only two members.
// An array of two elements represents a point with an x and y coordinate.
// The array is converted to a binary structure of 8 bytes.

   #include "Cstruct.ch"

   PROCEDURE Main
      LOCAL aMember := { 100, 200 }
      LOCAL aTypes  := { CTYPE_LONG, CTYPE_LONG }
      LOCAL nAlign  := 4
      LOCAL cBinary

      cBinary := HB_ArrayToStructure( aMember, aTypes, nAlign )

      ? Len( cBinary )                 // result: 8

      ? Bin2L( Left( cBinary, 4 ) )    // result: 100

      ? Bin2L( Right( cBinary, 4 ) )   // result: 200
   RETURN
```

HB_ATokens()

Splits a string into tokens based on a delimiter.

Syntax

```
HB_ATokens( <cString>           , ;
           [<cDelimiter>]       , ;
           [<lSkipQuotes>]      , ;
           [<lDoubleQuotesOnly>] ) --> aTokens
```

Arguments

`<cString>`

This is a character string which is tokenized based on the value of *<cDelimiter>*.

`<cDelimiter>`

A single character can be specified as delimiter used to tokenize the string *<cString>*. It defaults to a blank space (Chr(32)).

`<lSkipQuotes>`

This parameter defaults to .F. (false). When it is set to .T. (true), all portions of *<cString>* enclosed in single or double quotes are not searched for *<cDelimiter>*.

`<lDoubleQuoteOnly>`

The parameter is only relevant when *<lSkipQuotes>* is .T. (true). When *<lDoubleQuoteOnly>* is also .T. (true), only portion sof *<cString>* enclosed in double quotes are not searched for *<cDelimiter>*.

Return

The function returns an array of character strings.

Description

HB_ATokens() function splits *<cString>* into substrings, based on the value of *<cDelimiter>*. The delimiter is removed from *<cString>* and the resulting substrings are collected in an array, which is returned.

If *<lSkipQuotes>* is .T. (true), a quoted substring within *<cString>* is not split if it contains the delimiter sign. If the value of *<lSkipQuotes>* is .F. (false), which is the default, the quoted substring will be split, regardless of the quotes. The *<lDoubleQuoteOnly>* argument specifies if both the double and single quote signs are interpreted as a quote sign (.F.) or only the double quote is recognized as a quote sign (.T.).

Info

See also: At(), AtToken(), IN, Left(), Rat(), Right(), SubStr()
Category: Array functions, Character functions, Token functions
Source: vm\arrayshb.c

Example

```
// The example displays the result of HB_ATokens() using different
// combinations for <lSkipQuotes> and <lDoubleQuoteOnly>

   PROCEDURE Main
      LOCAL cString := [This\'_is\'_a_tok"e_n"ized_string]
      LOCAL aTokens := {}
      LOCAL i
```

```
   aTokens := HB_ATokens( cString, "_", .F., .F. )

   FOR i := 1 TO Len( aTokens )
      ? aTokens[i]
   NEXT
   // Result: array with 6 elements
   // This\'
   // is\'
   // a
   // tok"e
   // n"ized
   // string

   aTokens := HB_ATokens( cString, "_", .T., .F. )

   FOR i := 1 TO Len( aTokens )
      ? aTokens[i]
   NEXT
   // Result: array with 4 elements
   // This\'_is\'
   // a
   // tok"e_n"ized
   // string

   aTokens := HB_ATokens( cString, "_", .T., .T. )

   FOR i := 1 TO Len( aTokens )
      ? aTokens[i]
   NEXT
   // Result: array with 5 elements
   // This\'
   // is\'
   // a
   // tok"e_n"ized
   // string
RETURN
```

HB_AtX()

Locates a substring within a character string based on a regular expression.

Syntax

```
HB_AtX( <cRegEx>          , ;
        <cString>         , ;
       [<lCaseSensitive>], ;
       [@<nStart>]        , ;
       [@<nLen>]            ) --> cFoundString
```

Arguments

<cRegEx>

This is a character string holding the search pattern as a literal regular expression. Alternatively, the return value of HB_RegExComp() can be used.

<cString>

This is the character string being searched for a matching substring.

<lCaseSensitive>

This parameter defaults to .T. (true) so that a case sensitive search is performed. Passing .F. (false) results in a case insensitive search.

@<nStart>

If passed by reference, *<nStart>* gets assigned a numeric value indicating the start position of the found substring in *<cString>*. When there is no match, *<nStart>* is set to zero.

@<nLen>

If passed by reference, *<nLen>* gets assigned a numeric value indicating the length of the found substring in *<cString>*. When there is no match, *<nLen>* is set to zero.

Return

The function returns the first substring contained in *<cString>* that matches the regular expression *<cRegEx>*. If no match is found, the return value is NIL.

Description

Function HB_AtX() searches a character string for matching regular expression and returns the found substring. It is a simple pattern matching function which searches and extracts only the first occurance of a substring and optionally returns its position and size in the searched string when *<nStart>* and *<nLen>* are passed by reference.

Info

See also: At(), HB_RegEx(), HB_RegExAll(), HB_RegExSplit(), RAt(), SubStr()
Category: Character functions, Regular expressions, xHarbour extensions
Source: rtl\regex.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example extracts an eMail address from a text string.
// The search is case insensitive although the RegEx defines
// character classes only with upper case letters.

   PROCEDURE Main
```

```
   LOCAL cRegEx := "[A-Z0-9._%-]+@[A-Z0-9.-]+\.[A-Z]{2,4}"
   LOCAL cText  := "Send your request to info@xharbour.com " + ;
                   "for more information"
   LOCAL cEmail, nStart, nLen

   cEmail := HB_AtX( cRegEx, cText, .F., @nStart, @nLen )

   ? cEmail        // result: info@xharbour.com
   ? nStart        // result: 22
   ? nLen          // result: 17
RETURN
```

HB_BackGroundActive()

Queries and/or changes the activity of a single background task.

Syntax

```
HB_BackGroundActive( <nTaskHandle> [, <lNewActive>] ) --> lOldActive
```

Arguments

<nTaskHandle>

This is the numeric task handle as returned by HB_BackGroundAdd().

<lNewActive>

This is an optional logical value used to change the activity of a background task. Passing .T. (true) activates the background task, and .F. (false) deactivates it.

Return

The function returns the previous activation state of a background task as a logical value.

Description

Function HB_BackGroundActive() queries and optionally changes the activation state of a single background task. If the task handle, as returned from HB_BackGroundAdd(), is valid, the previous value for the activation state is returned. If an invalid task handle is passed, the return value is NIL.

Info

See also:	HB_BackGroundAdd(), HB_BackGroundDel(), HB_BackGroundReset(), HB_BackGroundRun(), HB_BackGroundTime(), HB_IdleAdd()
Category:	Background processing, xHarbour extensions
Source:	rtl\bkgtsks.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_BackGroundAdd()

Adds a new background task.

Syntax

```
HB_BackGroundAdd( <bAction>, [<nMillisecs>], [<lActive>] ) --> nTaskHandle
```

Arguments

<bAction>

This is a codeblock that will be executed in the background.

<nMillisecs>

This is a numeric value specifying the number of milliseconds after which the task is executed. The default value is zero,which means that the background task is executed each time background processing is invoked.

<lActive>

The default value of *<lActive>* is .T. (true) causing *<bAction>* being checked for immediate background activation. Passing .F. (false) adds the task to the internal task list but does not execute it until it is activated with HB_BackGroundActive().

Return

The function returns a numeric task handle that must be preserved for other background processing functions, such as HB_BackGroundDel().

Description

Function HB_BackGroundAdd() adds the passed codeblock to the list of background tasks that will be executed concurrently to the main program. There is no limit for the number of background tasks.

Background task processing must be activated using the command SET BACKGROUND TASKS ON. Background tasks are processed sequentially until a SET BACKGROUND TASKS OFF is issued or an idle state is encountered. The idle state is the state of the xHarbour virtual machine (VM) when it waits for user input from the keyboard or the mouse. The VM enters idle state during Inkey() calls. All applications that do not use Inkey() function calls can signal the idle state with a call to the HB_IdleState() function.

Note: to avoid interruption of background processing during idle states, an idle task must be defined like HB_IdleAdd(\{||HB_BackGroundRun()\}).

An alternative for background tasks is provided with threads. Refer to function StartThread() for running parts of an application simultaneously in multiple threads.

Info

See also:	HB_BackGroundActive(), HB_BackGroundDel(), HB_BackGroundReset(), HB_BackGroundRun(), HB_BackGroundTime(), HB_ExecFromArray(), HB_IdleAdd(), StartThread()
Category:	Background processing, xHarbour extensions
Source:	rtl\bkgtsks.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example uses a regular background task to display the time
// once in a second while MemoEdit() is active. To ensure continuous
```

```
// display while MemoEdit() waits for user input, an idle task is
// defined which enforces background task processing.

   PROCEDURE Main
      LOCAL nTask, nIdle

      DispOutAtSetPos( .F. )
      SET BACKGROUND TASKS ON

      nIdle := HB_IdleAdd( {|| HB_BackGroundRun() } )

      nTask := HB_BackGroundAdd( {|| ShowTime() }, 1000 )

      MemoEdit( MemoRead( "Test.prg" ), 1, 0, MaxRow()-2, MaxCOl() )

      HB_BackGroundDel( nTask )
      HB_IdleDel( nIdle )
   RETURN


   PROCEDURE ShowTime()
      DispoutAt( MaxRow(), MaxCol()-7, Time() )
   RETURN
```

HB_BackGroundDel()

Removes a background task from the internal task list.

Syntax

```
HB_BackGroundDel( <nTaskHandle> ) --> bAction
```

Arguments

<nTaskHandle>

This is the numeric task handle as returned by HB_BackGroundAdd().

Return

The function returns the code block assoiated with the task handle, or NIL if an invalid task handle is specified.

Description

Function HB_BackGroundDel() removes the task associated with the passed task handle from the internal list of background tasks. The task handle must be a value returned by a previous call to the HB_BackGroundAdd() function. If the specified task exists, it is deactivated and the associated code block is returned.

Info

See also:	HB_BackGroundActive(), HB_BackGroundAdd(), HB_BackGroundReset(), HB_BackGroundRun(), HB_BackGroundTime(), HB_ExecFromArray()
Category:	Background processing, xHarbour extensions
Source:	rtl\bkgtsks.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_BackGroundReset()

Resets the internal counter of background tasks.

Syntax

```
HB_BackGroundReset() --> NIL
```

Return

The return value is always NIL.

Description

Function HB_BackGroundReset() resets the internal counter identifying the next background task to be executed to 1. As a result, the next cycle for background processing starts with the first task defined.

Info

See also:	HB_BackGroundActive(), HB_BackGroundAdd(), HB_BackGroundDel(), HB_BackGroundRun(), HB_BackGroundTime()
Category:	Background processing, xHarbour extensions
Source:	rtl\bkgtsks.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_BackGroundRun()

Enforces execution of one or all background tasks.

Syntax

```
HB_BackGroundRun( [<nTaskHandle>] ) --> NIL
```

Arguments

<nTaskHandle>

This is the numeric task handle as returned by HB_BackGroundAdd(). If omitted, all background tasks are executed.

Return

The return value is always NIL.

Description

Function HB_BackGroundRun() executes a single background task defined with *<nTaskHandle>*, or all tasks from the internal task list when no task handle is specified. The function can be used to enforce execution of background tasks during regular program execution or during idle states of a program.

Note: HB_BackGroundRun() executes the task specified with *<nTaskHandle>* immediately, even if the wait interval defined for this task has not elapsed, or if the task is not active.

Info

See also:	HB_BackGroundActive(), HB_BackGroundAdd(), HB_BackGroundDel(), HB_BackGroundReset(), HB_BackGroundTime(), HB_IdleAdd()
Category:	Background processing, xHarbour extensions
Source:	rtl\bkgtsks.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_BackGroundTime()

Queries or changes the wait interval in milliseconds after which the task is executed.

Syntax

```
HB_BackGroundTime( ( <nTaskHandle> [, <nNewInterval>] ) --> nOldInterval
```

Arguments

<nTaskHandle>

This is the numeric task handle as returned by HB_BackGroundAdd().

<nNewInterval>

This is an optional numeric value specifying the time interval in milliseconds a background task should wait between two execution cycles. When set to zero, a background task is invoked each time the internal task list is checked for background execution.

Return

The function returns the previous wait interval of the background task as a numeric value. If an invalid task handle id passed, the return value is NIL.

Description

Function HB_BackGroundTime() queries or changes the number of milliseconds after which the task is executed. Normally, the time interval is defined with the second parameter of HB_BackGroundAdd().

Info

See also:	HB_BackGroundActive(), HB_BackGroundAdd(), HB_BackGroundDel(), HB_BackGroundReset(), HB_BackGroundRun()
Category:	Background processing, xHarbour extensions
Source:	rtl\bkgtsks.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_Base64Decode()

Decodes a base 64 encoded character string.

Syntax

```
HB_Base64Decode( <cBase64> ) --> cString
```

Arguments

<cBase64>

This is an encoded character string previously returned from HB_Base64Encode().

Return

The function returns the decoded character string.

Description

HB_Base64Decode() decodes a base 64 encoded character string in memory and returns the original data. Use function HB_Base64DecodeFile() to decode an entire file.

Info

See also:	HB_Base64DecodeFile(), HB_Base64Encode(), HB_Base64EncodeFile()
Category:	Encoding/Decoding, xHarbour extensions
Source:	tip\encoding\Base64.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_Base64DecodeFile()

Decodes a base 64 encoded file.

Syntax

```
HB_Base64DecodeFile( <cBase64File>, <cTargetFile> ) --> NIL
```

Arguments

`<cBase64File>`

This is a character string holding the name of a base 64 encoded file. It can be created by HB_Base64EncodeFile().

`<cTargetFile>`

This is a character string holding the name of the file where the decoded data is written to.

Return

The return value is always NIL.

Description

HB_Base64DecodeFile() reads an entire base 64 encoded file, decodes it, and writes the original data into a second file.

Info

See also:	HB_Base64Decode(), HB_Base64Encode(), HB_Base64EncodeFile()
Category:	Encoding/Decoding, xHarbour extensions
Source:	tip\encoding\Base64.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_Base64Encode()

Encodes a character string base 64.

Syntax

```
HB_Base64Encode( <cString>, <nBytes> ) --> cBase64
```

Arguments

<cString>

This is a character string to encode with the base 64 algorithm.

<nBytes>

The number of bytes to encode from *<cString>* must be passed as second parameter. Use the expression Len(*<cString>*) to encode the entire string.

Return

The function returns a base 64 encoded character string.

Description

HB_Base64Encode() uses the Base 64 algorithm for encoding a character string. The encoded string is about one third larger than the original string, but contains only alphanumeric characters.

Pass the resulting string to function HB_Base64Decode() to obtain the original data.

Info

See also: HB_Base64Decode(), HB_Base64DecodeFile(), HB_Base64EncodeFile()
Category: Encoding/Decoding, xHarbour extensions
Source: tip\encoding\Base64.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example outlines base 64 encoding and decoding.

   PROCEDURE Main
      LOCAL cString := "xHarbour"
      LOCAL cBase64 := HB_Base64Encode( cString, Len(cString ) )

      ? cBase64                       // result: eEhhcmJvdXI=
      ? HB_Base64Decode( cBase64 )    // result: xHarbour

   RETURN
```

HB_Base64EncodeFile()

Encodes a file base 64.

Syntax

```
HB_Base64EncodeFile( <cFilename>, <cBase64File> ) --> NIL
```

Arguments

<cFilename>

This is a character string holding the name of the file to encode.

<cBase64File>

This is a character string holding the name of the file where base 64 encoded data is written to.

Return

The return value is always NIL.

Description

HB_Base64EncodeFile() reads an entire file, encodes it base 64, and writes the encoded data into a second file.

Info

See also:	HB_Base64Decode(), HB_Base64DecodeFile(), HB_Base64Encode()
Category:	Encoding/Decoding, xHarbour extensions
Source:	tip\encoding\Base64.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example outlines base 64 encoding and decoding of files.
// A PRG file is encoded in a second file which is decoded into
// a third file. The contents of the original and third file are
// identical.

   PROCEDURE Main
       LOCAL cFileIn  := "HB_Base64.prg"
       LOCAL cFileOut := "HB_Base64.enc"

       HB_Base64EncodeFile( cFileIn, cFileOut )

       HB_Base64DecodeFile( cFileOut, "Test.txt" )

       ? Memoread( cFileIn ) == MemoRead( "Test.txt" ) // Result: .T.

   RETURN
```

HB_BitAnd()

Performs a bitwise AND operation with numeric integer values.

Syntax

```
HB_BitAnd( <nInteger1>, <nInteger2> ) --> nResult
```

Arguments

<nInteger>

Two numeric integer values must be passed. If numbers with decimal fractions or values of other data types are specified, a runtime error is generated. Function Int() can be used to make sure both parameters are integer values.

Return

The function returns a numeric value. It is the result of the bitwise AND operation with both parameters.

Description

The function performs bitwise AND operation with two integer values. It compares individual bits of both values at the same position. If bits at the same position are set in both values, the bit at this position is also set in the return value.

Info

See also:	& (bitwise AND), HB_BitIsSet(), HB_BitNot(), HB_BitOr(), HB_BitXOr()
Category:	Bitwise functions, xHarbour extensions
Source:	rtl\hbBitf.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_BitIsSet()

Checks if a bit is set in a numeric integer value.

Syntax

```
HB_BitIsSet( <nInteger>, <nPosition> ) --> lBitIsSet
```

Arguments

`<nInteger>`

This is a numeric integer value to check for set bits.

`<nPosition>`

This numeric parameter specifies the position of the bit to test. The least significant bit has position zero, so that the range for *<nPosition>* is 0 to 31 on a 32-bit operating system.

Return

The function returns .T. (true) when the bit is set at the specified position, otherwise .F. (false).

Description

Function HB_BitIsSet() tests individual bits of a numeric integer value. This allows, for example, to encode multiple status variables in a single numeric value and detect different values from a single memory variable, or to build the binary character representation for an integer number.

Info

See also:	HB_BitReset(), HB_BitSet(), HB_BitShift()
Category:	Bitwise functions, xHarbour extensions
Source:	rtl\hbBitf.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example implements function Num2Bit() which creates
// a characters string consisting of "0" and "1"

   PROCEDURE Main()
      ? Num2Bit( 1 )         // result: 0000000000000001
      ? Num2Bit( 64 )        // result: 0000000001000000
      ? Num2Bit( 1234 )      // result: 0000010011010010
      ? Num2Bit( 12345678 )  // result: 00000000101111000110000101001110
   RETURN

   FUNCTION Num2Bit( nNumber )
      LOCAL nInt := Int( nNumber )
      LOCAL nLen := IIf( Abs(nInt) > 2^15, 31, 15 )
      LOCAL cBin := Replicate( "0", nLen+1 )
      LOCAL nPos

      FOR nPos := 0 TO nLen
         IF HB_BitIsSet( nInt, nPos )
            cBin[ nLen-nPos+1 ] := "1"
         ENDIF
      NEXT
   RETURN cBin
```

HB_BitNot()

Performs a bitwise NOT operation with a numeric integer value.

Syntax

```
HB_BitNot( <nInteger> ) --> nResult
```

Arguments

<nInteger>

A numeric integer value must be passed. If a number with decimal fraction or a value of other data types is specified, a runtime error is generated. Function Int() can be used to make sure the parameter is an integer value.

Return

The function returns a numeric value. It is the result of the bitwise NOT operation with the bits of the parameter.

Description

The function performs bitwise NOT operation with the bits of the passed parameter. That is, all bits are inverted.

Info

See also: HB_BitAnd(), HB_BitIsSet(), HB_BitOr(), HB_BitXOr()
Category: Bitwise functions, xHarbour extensions
Source: rtl\hbBitf.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example shows the result of a bitwise NOT operation

   PROCEDURE Main
      LOCAL i

      FOR i := -5 TO 5
         ? i, HB_BitNot(i)
      NEXT

      ** Output
      //        -5          4
      //        -4          3
      //        -3          2
      //        -2          1
      //        -1          0
      //         0         -1
      //         1         -2
      //         2         -3
      //         3         -4
      //         4         -5
      //         5         -6
   RETURN
```

HB_BitOr()

Performs a bitwise OR operation with numeric integer values.

Syntax

```
HB_BitOr( <nInteger1>, <nInteger2> ) --> nResult
```

Arguments

<nInteger>

Two numeric integer values must be passed. If numbers with decimal fractions or values of other data types are specified, a runtime error is generated. Function Int() can be used to make sure both parameters are integer values.

Return

The function returns a numeric value. It is the result of the bitwise OR operation with both parameters.

Description

The function performs bitwise OR operation with two integer values. It compares individual bits of both values at the same position. If bits at the same position are set in either value, the bit at this position is also set in the return value. The bit in the return value is not set when the bits of both values at this position are not set.

Info

See also:	| (bitwise OR), HB_BitAnd(), HB_BitIsSet(), HB_BitNot(), HB_BitXOr()
Category:	Bitwise functions, xHarbour extensions
Source:	rtl\hbBitf.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_BitReset()

Sets a bit in a numeric integer value to 0.

Syntax

```
HB_BitReset( <nInteger>, <nPosition> ) --> nResult
```

Arguments

`<nInteger>`

This is a numeric integer value.

`<nPosition>`

This numeric parameter specifies the position of the bit to set to 0. The least significant bit has position zero, so that the range for <*nPosition*> is 0 to 31 on a 32-bit operating system.

Return

The function returns the integer number where the specified bit is set to 0.

Description

Function HB_BitReset() sets a single bit of a numeric integer to 0. If the bit at position <*nPosition*> is not set in <*nInteger*>, the result has the same value, otherwise its value is different.

Info

See also:	HB_BitIsSet(), HB_BitSet(), HB_BitShift()
Category:	Bitwise functions, xHarbour extensions
Source:	rtl\hbBitf.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays the result of HB_BitReset() with the
// help of a user defined function Num2Bit(). The bit sequence
// and the numeric value resulting from the operation is shown.

   PROCEDURE Main
      LOCAL nInt := 4567
      LOCAL n, i

      ? "Original :", " ", Num2Bit( nInt ), LTrim(Str(nInt))
      ?
      FOR i:= 0 TO 7
         n :=  HB_BitReset( nInt, i )
         ? "Reset Bit:", LTrim(Str(i)), Num2Bit( n ), LTrim(Str(n))
      NEXT

      ** output
      // Original :   11010111 4567

      // Reset Bit: 0 11010110 4566
      // Reset Bit: 1 11010101 4565
      // Reset Bit: 2 11010011 4563
      // Reset Bit: 3 11010111 4567
      // Reset Bit: 4 11000111 4551
      // Reset Bit: 5 11010111 4567
      // Reset Bit: 6 10010111 4503
```

```
   // Reset Bit: 7 01010111 4439
RETURN


FUNCTION Num2Bit( nNumber )
   LOCAL nInt := Int( nNumber )
   LOCAL nLen := 7
   LOCAL cBin := Replicate( "0", nLen+1 )
   LOCAL nPos

   FOR nPos := 0 TO nLen
      IF HB_BitIsSet( nInt, nPos )
         cBin[ nLen-nPos+1 ] := "1"
      ENDIF
   NEXT
RETURN cBin
```

HB_BitSet()

Sets a bit in a numeric integer value to 1.

Syntax

```
HB_BitSet( <nInteger>, <nPosition> ) --> nResult
```

Arguments

`<nInteger>`

This is a numeric integer value.

`<nPosition>`

This numeric parameter specifies the position of the bit to set to 1. The least significant bit has position zero, so that the range for <*nPosition*> is 0 to 31 on a 32-bit operating system.

Return

The function returns the integer number where the specified bit is set to 1.

Description

Function HB_BitSet() sets a single bit of a numeric integer to 1. If the bit at position <*nPosition*> is set in <*nInteger*>, the result has the same value, otherwise its value is different.

Info

See also:	HB_BitIsSet(), HB_BitReset(), HB_BitShift()
Category:	Bitwise functions, xHarbour extensions
Source:	rtl\hbBitf.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays the result of HB_BitSet() with the
// help of a user defined function Num2Bit(). The bit sequence
// and the numeric value resulting from the operation is shown.

   PROCEDURE Main
      LOCAL nInt := 45678
      LOCAL n, i

      ? "Original:", " ", Num2Bit( nInt ), LTrim(Str(nInt))
      ?
      FOR i:= 0 TO 7
         n :=  HB_BitSet( nInt, i )
         ? "Set Bit: ", LTrim(Str(i)), Num2Bit( n ), LTrim(Str(n))
      NEXT

      ** output
      // Original:   01101110 45678

      // Set Bit:  0 01101111 45679
      // Set Bit:  1 01101110 45678
      // Set Bit:  2 01101110 45678
      // Set Bit:  3 01101110 45678
      // Set Bit:  4 01111110 45694
      // Set Bit:  5 01101110 45678
      // Set Bit:  6 01101110 45678
```

```
   // Set Bit:  7 11101110 45806
RETURN


FUNCTION Num2Bit( nNumber )
   LOCAL nInt := Int( nNumber )
   LOCAL nLen := 7
   LOCAL cBin := Replicate( "0", nLen+1 )
   LOCAL nPos

   FOR nPos := 0 TO nLen
      IF HB_BitIsSet( nInt, nPos )
         cBin[ nLen-nPos+1 ] := "1"
      ENDIF
   NEXT
RETURN cBin
```

HB_BitShift()

Shifts bits in a numeric integer value.

Syntax

```
HB_Bitshift( <nInteger>, <nShift> ) --> nResult
```

Arguments

<nInteger>

This is a numeric integer value.

<nShift>

This numeric integer value specifies how far to shift bits. When a negative number is passed, bits are shifted to the right. A positive value shifts bits to the left.

Return

The function returns a numeric value. It is the result of the bit-shift operation.

Description

Function HB_BitShift() shifts the bits of a numeric integer to the left or the right, depending on the sign of *<nShift>*.

A shift operation involves all bits of *<nInteger>*. When the bits are shifted one place to the left, the most significant, or highest, bit is discarded and the least significant (or lowest) bit is set to 0. When the bits are shifted one place to the right, the lowest bit is discarded and the highest bit is set to 0.

A numeric value has 32 bits on a 32 bit operating system.

Info

See also: <<, >>, HB_BitIsSet(), HB_BitReset(), HB_BitSet()
Category: Bitwise functions, xHarbour extensions
Source: rtl\hbBitf.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays the result of bit-shift operations to
// the left and the right with the aid of a user defined function
// Num2Bit(). Note that the initial integer has only one bit set.
// This bit "drops off" when shifted 3 places to the right.

   PROCEDURE Main
      LOCAL nInt := 4
      LOCAL i, n

      ? "Original:", "  ", Num2Bit( nInt ), Str( nInt, 4 )
      ?
      FOR i:= -3 TO 3
         n :=  HB_BitShift( nInt, i )
         ? "Shifted :", Str(i,2), Num2Bit( n ),  Str( n, 4 )
      NEXT

      ** output
      // Original:    00000100    4

      // Shifted : -3 00000000    0 (bit dropped off)
```

```
   // Shifted : -2 00000001    1
   // Shifted : -1 00000010    2
   // Shifted :  0 00000100    4
   // Shifted :  1 00001000    8
   // Shifted :  2 00010000   16
   // Shifted :  3 00100000   32
RETURN


FUNCTION Num2Bit( nNumber )
   LOCAL nInt := Int( nNumber )
   LOCAL nLen := 7
   LOCAL cBin := Replicate( "0", nLen+1 )
   LOCAL nPos

   FOR nPos := 0 TO nLen
      IF HB_BitIsSet( nInt, nPos )
         cBin[ nLen-nPos+1 ] := "1"
      ENDIF
   NEXT
RETURN cBin
```

HB_BitXOr()

Performs a bitwise XOR operation with numeric integer values.

Syntax

```
HB_BitXOr( <nInteger1>, <nInteger2> ) --> nResult
```

Arguments

`<nInteger>`

Two numeric integer values must be passed. If numbers with decimal fractions or values of other data types are specified, a runtime error is generated. Function Int() can be used to make sure both parameters are integer values.

Return

The function returns a numeric value. It is the result of the bitwise XOR operation with both parameters.

Description

The function performs bitwise XOR operation with two integer values. It compares individual bits of both values at the same position. The bit at the same position is set in the return value, when both integer values have the bit set differently at the same position. If both integers have the same bit set or not set, i.e. when both bits are the same, the corresponding bit in the return value is not set.

Info

See also: ^^ (bitwise XOR), HB_BitAnd(), HB_BitIsSet(), HB_BitNot(), HB_BitOr()
Category: Bitwise functions, xHarbour extensions
Source: rtl\hbBitf.c
LIB: xhb.lib
DLL: xhbdll.dll

HB_BldLogMsg()

Converts a list of parameters into a text string.

Syntax

```
HB_BldLogMsg( <xParam,...> ) --> cParamValues
```

Arguments

`<xParam,...>`

This is an arbitrary list of parameters to create a text string from.

Return

The function returns a character string holding the values of all passed parameters in textual form.

Description

HB_BldLogMsg() is used to convert an arbitrary number of variables into a text string that can be written to a log file. It is a utility function used in xHarbour's Log system.

Info

See also:	INIT LOG, TraceLog(), HB_LogDateStamp()
Category:	Debug functions, Log functions, xHarbour extensions
Source:	rtl\hblog.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_BuildDate()

Retrieves the formatted build date of the xHarbour compiler

Syntax

```
HB_BuildDate() --> cDateTime
```

Return

The function returns a character string formatted as "MMM dd yyyy hh:mm:ss" where "MMM" is the abbreviated month name, "dd" is the day, "yyyy" is the year and "hh:mm:ss" is the time.

Description

FUnction HB_BuildDate() is used to retrieve a formatted date and time string of the day, the xHarbour compiler was built.

Info

See also:	HB_BuildInfo(), HB_Compiler(), Os(), Os_VersionInfo()
Category:	Environment functions, xHarbour extensions
Source:	rtl\version.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
PROCEDURE Main
   ? HB_BuildDate()
RETURN
```

HB_BuildInfo()

Retrieves build information of the xHarbour compiler.

Syntax

```
HB_BuildInfo( <nWhichInfo> ) -->  cBuildInfo
```

Arguments

<nWhichInfo>

This is a numeric value indicating the information to retrieve. #define constants are listed in the Hbver.ch file that can be used for *<nWhichInfo>*.

Return

The function returns the requested build information. The data type of the return value depends on *<nWhichInfo>*.

Description

Function HB_BuildInfo() is used to retrieve information about the current build of the xHarbour compiler. The following information is returned depending on the #define constant used for *<nWichInfo>*:

Build information for the xHarbour compiler

Constant	Value	Valtype()	Description
_HB_VER_MAJOR	1	N	Major version number
_HB_VER_MINOR	2	N	Minor version number
_HB_VER_REVISION	3	N	Revision number
_HB_VER_LEX	4	C	Lex version
_HB_VER_AS_STRING	5	C	Complete version as character string
_HB_PCODE_VER	6	N	Version number of PCode engine
_HB_VER_COMPILER	7	C	Version of C compiler
_HB_VER_PLATFORM	8	C	Platform xHarbour is running on
_HB_VER_BUILD_DATE	9	C	Date xHarbour was built
_HB_VER_BUILD_TIME	10	C	Time xHarbour was built
_HB_VER_LENTRY	11	C	Last entry in CVS ChangeLog file
_HB_VER_CHLCVS	12	C	Revision of last entry in CVS
_HB_VER_C_USR	13	C	Reserved
_HB_VER_L_USR	14	C	Reserved
_HB_VER_PRG_USR	15	C	Reserved
_HB_EXTENSION	16	L	#define HB_EXTENSION is used for current build
_HB_C52_UNDOC	17	L	#define HB_C52_UNDOC is used for current build
_HB_C52_STRICT	18	L	#define HB_C52_STRICT is used for current build
_HB_COMPAT_C53	19	L	#define HB_COMPAT_C53 is used for current build
_HB_COMPAT_XPP	20	L	#define HB_COMPAT_XPP is used for current build
_HB_COMPAT_VO	21	L	#define HB_COMPAT_VO is used for current build
_HB_COMPAT_FLAGSHIP	22	L	#define HB_COMPAT_FLAGSHIP is used for current build
_HB_COMPAT_FOXPRO	23	L	#define HB_COMPAT_FOXPRO

			is used for current build
_HB_COMPAT_DBASE	24	L	#define HB_COMPAT_DBASE is used for current build
_HB_HARBOUR_OBJ_GENERATION	25	L	Reserved
_HB_HARBOUR_STRICT_ANSI_C	26	L	Reserved
_HB_CPLUSPLUS	27	L	Reserved
_HB_HARBOUR_YYDEBUG	28	L	Reserved
_HB_SYMBOL_NAME_LEN	29	N	Maximum length of symbolic variable names
_HB_MULTITHREAD	30	L	Multi-threading is supported
_HB_VM_OPTIMIZATION	31	N	Optimization level of Virtual Machine
_HB_LANG_ID	32	C	Language ID of language used for build
_HB_ARRAY_MODE	33	N	Reserved
_HB_CREDITS	34	A	Credits to xHarbour developers

Info

See also: HB_BuildDate(), HB_Compiler(), Os(), Os_VersionInfo()
Category: Environment functions, xHarbour extensions
Header: hbver.ch
Source: rtl\version.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays build information of the xHarbour compiler
// and the operating system

   #include "hbver.ch"

   PROCEDURE Main
      CLS
      ? "Compiler:", HB_BuildInfo( _HB_VER_AS_STRING )
      ?
      ? "Operating System:", HB_BuildInfo( _HB_VER_PLATFORM )
   RETURN
```

HB_CheckSum()

Calculates the checksum for a stream of data using the Adler32 algorithm.

Syntax

```
HB_CheckSum( <cString> ) --> nAdler32
```

Arguments

<cString>

This is a character string to calulate the checksum for.

Return

The function returns the calculated checksum as a numeric value.

Description

HB_CheckSum() implements the Adler32 algorithm for calculating the checksum of a character string. This algorithm computes faster than the algorithm of the HB_CRC32() function, but is less reliable, especially for short data streams. The following table compares the speed of checksum algorithms available in xHarbour

Algorithm comparison

Function	Algorithm	Relative Speed
HB_CheckSum()	Adler 32 bit	1.00
HB_CRC32()	CRC 32 bit	1.46
HB_MD5()	MD5 128 bit	4.51

Info

See also: HB_CRC32(), HB_MD5()
Category: Checksum functions, xHarbour extensions
Source: rtl\hbchksum.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example calculates the checksum for two character
// strings. Note that if the example is compiled, it will
// display a different checksum for cString2 than shown in
// the example. The result changes the checksum of the file
// "Checksum.prg"

   PROCEDURE Main
      LOCAL cString1 := "Hello world"
      LOCAL cString2 := Memoread( "Checksum.prg" )

      ? HB_CheckSum( cString1 )   // result:  413140028.00

      ? HB_CheckSum( cString2 )   // result: 1460748119.00

   RETURN
```

HB_Clocks2Secs()

Calculates seconds from CPU ticks.

Syntax

```
HB_Clocks2Secs( <nTicks> ) --> nSeconds
```

Arguments

<nTicks>

This is a numeric value representing the CPU ticks to convert to seconds.

Return

The function returns the time in seconds representing the CPU ticks as a numeric value.

Info

See also: ElapTime(), Seconds(), SecondsCpu()
Category: Date and time, xHarbour extensions
Source: rtl\seconds.c
LIB: xhb.lib
DLL: xhbdll.dll

HB_CloseProcess()

Closes a child process

Syntax

```
HB_CloseProcess( <nProcessHandle>, ;
                [<lWaitForTermination>] ) --> lSuccess
```

Arguments

`<nProcessHandle>`

This is the numeric process handle of the child process to close. It is returned from HB_OpenProcess().

`<lWaitForTermination>`

This parameter defaults to .T. (true) so that the parent process waits until the child process has regularly ended with program execution. When .F. (false) is passed, the program running in the child process is unconditionally terminated (it gets "killed").

Return

The function returns .T. (true) when the child process has successfully ended program execution, otherwise .F. (false) is returned.

Description

The function waits for the end of program execution in a child process previously opened with HB_OpenProcess(). If the child process is still executing its program, HB_CloseProcess() waits for the termination of the program running in the child process, unless *<lWaitForTermination>* is set to .F. (false).

Use function HB_ProcessValue() to determine if a child process is still executing its program.

Note: when program execution in the child process has ended, the handle *<nProcessHandle>* must be explicitly closed by passing it to FClose().

Info

See also:	HB_OpenProcess(), HB_ProcessValue(), StopThread(), QUIT
Category:	Process functions, xHarbour extensions
Source:	rtl\philes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_ClrArea()

Replaces a color attribute in a screen region.

Syntax

```
HB_ClrArea( <nTop>      , ;
            <nLeft>     , ;
            <nBottom>   , ;
            <nRight>    , ;
            <nColorAttr> ) --> NIL
```

Arguments

<nTop> and <nLeft>

Numeric values indicating the screen coordinates for the upper left corner of the screen area.

<nBottom> and <nRight>

Numeric values indicating the screen coordinates for the lower right corner of the screen area.

<nColorAttr>

This is a numeric color attribute in the range between 0 and 255 for coloring the screen area. Refer to ColorToN() for obtaining a numeric color attribute from a color string.

Return

The function changes the color in the specified screen area and returns NIL.

Info

See also: ColorRepl(), ColorToN(), ColorWin(), HB_Shadow()
Category: Screen functions, xHarbour extensions
Source: rtl\shadow.c
LIB: xhb.lib
DLL: xhbdll.dll

HB_CmdArgArgV()

Returns the first command line argument (EXE file name).

Syntax

```
HB_CmdArgArgV() --> cExeFile
```

Return

The function returns the first command line argument as a character string.

Description

Function HB_CmdArgArgV() queries the command line and returns the first argument. It is the name of the EXE file as entered by the user. Note that the file name may not include the ".exe" extension or directory information.

Info

See also:	ExeName(), HB_ArgC(), HB_ArgCheck(), HB_ArgV(), PCount(), PValue()
Category:	Debug functions, Environment functions, xHarbour extensions
Source:	vm\cmdarg.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_ColorIndex()

Extracts a color value from a color string.

Syntax

```
HB_ColorIndex( <cColorString>, <nColorIndex> ) --> cColorValue
```

Arguments

<cColorString>

This is a SetColor() compliant character string holding color values as a comma separated list.

<nColorIndex>

A numeric value identifying the ordinal position of a color value in the color string. *<nColorIndex>*is zero based, i.e. the first color value has the ordinal position 0.

Return

The function returns the color value at position *<nPos>* in the passed color string as a character string. If *<cColorString>* has less than *<nPos>*+1 colors, the return value is a null string ("").

Description

The function HB_ColorIndex() extracts a color value from *<cColorString>*. Constants from the COLOR.CH file can be used to address a specific color value:

#define constants for HB_ColorIndex()

Constant	Value	Description
CLR_STANDARD	0	All screen output commands and functions
CLR_ENHANCED	1	GETs and selection highlights
CLR_BORDER	2	Screen border (not supported on EGA and VGA monitors)
CLR_BACKGROUND	3	Not supported
CLR_UNSELECTED	4	Unselected GETs

Info

See also: ColorSelect(), GetClrPair(), SetClrPair(), SetColor()
Category: Screen functions, xHarbour extensions
Header: color.ch
Source: rtl\colorind.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates return values of HB_ColorIndex()

   #include "Color.ch"

   PROCEDURE Main
      LOCAL cColor := "N/W,W+/N,W+/W,W+/B,GR+/B"

      SetColor( cColor )

      ? HB_ColorIndex( cColor, CLR_STANDARD )    // result: N/W
      ? HB_ColorIndex( cColor, CLR_ENHANCED )    // result: W+/N
      ? HB_ColorIndex( cColor, CLR_BORDER )      // result: W+/W
      ? HB_ColorIndex( cColor, CLR_BACKGROUND )  // result: W+/B
```

```
      ? HB_ColorIndex( cColor, CLR_UNSELECTED )  // result: GR+/B
   RETURN
```

HB_ColorToN()

Converts a color value to a numeric color attribute.

Syntax

```
HB_ColorToN( <cColor> ) --> nColorAttribute
```

Arguments

<cColor>

This is a character string holding a SetColor() compliant color value.

Return

The function converts a color value to the corresponding color attribute and returns the attribute as a numeric value.

Info

See also:	NtoColor()
Category:	Screen functions, xHarbour extensions
Source:	rtl\setcolor.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_Compiler()

Retrieves the version of the C compiler shipped with xHarbour.

Syntax

```
HB_Compiler() --> cCompilerVersion
```

Return

The function returns a character string holding the name and version of the C compiler shipped with xHarbour.

Description

Function HB_Compiler() is used to retrieve information about the C compiler shipped with xHarbour. The C compiler is an integral part of xHarbour and produces OBJ files.

Info

See also:	HB_BuildDate(), HB_BuildInfo(), HB_PCodeVer(), Os(), Os_VersionInfo()
Category:	Environment functions, xHarbour extensions
Source:	rtl\version.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_Compress()

Compresses a character string (ZIP).

Syntax

```
HB_Compress( <cString> ) --> cCompressed

or

HB_Compress( <nComprFactor>, ;
             <cString>     , ;
             <nStringLen>  , ;
            @<cBuffer>     , ;
            @<nBytes>        ) --> nErrorCode
```

Arguments

<cString>

This is a character string to be ZIP compressed.

<nComprFactor>

When a numeric value is passed as first parameter, it indicates the compression factor. This factor influences the compression speed and result. A higher compression takes more time. #define constants are available in the file HbCompress.ch that can be used for *<nComprFactor>*.

ZIP compression factors

Constant	Value	Description
HB_Z_NO_COMPRESSION	0	No compression
HB_Z_BEST_SPEED	1	Fastest compression
HB_Z_BEST_COMPRESSION	9	Highest compression
HB_Z_DEFAULT_COMPRESSION *)	(-1)	Default compression

**) default*

<nStringLen>

This numeric value indicates the number of bytes of the input string to compress. Use the expression Len(*<cString>*) to compress the entire input string.

@<cBuffer>

This is a pre-allocated character string. It must be passed by reference and receives the compressed data. Use function HB_CompressBufLen() to calculate the number of bytes required for *<cBuffer>*.

@<nBytes>

This parameter must be passed by reference. It receives the actual number of compressed bytes when the compression is complete.

Return

The function returns either the compressed character string, or a numeric error code indicating success of the compression operation. See the description for *Simple* and *Advanced* usage below.

Description

HB_Compress() is a ZIP compression function for character strings. It is implemented in two "flavours" allowing for simple and advanced data compression.

Simple usage

The easiest way of compressing a character string is by passing it as a single parameter to HB_Compress(). The function returns the compressed data as a character string.

Advanced usage

The function allows for optimizing the compression between speed and compression result. This requires five parameters be passed, the first of which determines the compression factor. The result of the compression is received in two reference parameters. *<cBuffer>* must be a string large enough to hold the compression result. The function HB_CompressBufLen() can be used to calculate the size of *<cBuffer>*.

When the compression is complete, the return value is numeric. The value zero indicates a successful operation. Values other than zero can be passed to HB_CompressErrorDesc() to obtain a descriptive error message.

Info

See also:	HB_CompressBufLen(), HB_CompressErrorDesc(), HB_Uncompress()
Category:	ZIP compression, xHarbour extensions
Header:	HbCompress.ch
Source:	rtl\hbcomprs.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates simple and advanced compression

   #include "HbCompress.ch"

   PROCEDURE Main
      LOCAL cText := MemoRead( "CompressTest.prg" )
      LOCAL cSimple
      LOCAL cAdvanced, nRequiredBytes, nError

      t1 := Seconds()
      cSimple := HB_Compress( cText )
      t2 := Seconds()

      nRequiredBytes := HB_CompressBufLen( Len(cText) )
      cAdvanced      := Space( nRequiredBytes )
      t3 := Seconds()
      nError := HB_Compress( HB_Z_BEST_COMPRESSION, ;
                             cText, ;
                             Len( cText ), ;
                             @cAdvanced, ;
                             @nRequiredBytes )
      t4 := Seconds()

      ? "Simple  :", t2-t1, Len(Trim(cSimple))
      ? "Advanced:", t4-t3, Len(Trim(cAdvanced))
   RETURN
```

HB_CompressBufLen()

Calculates the buffer size required for compression.

Syntax

```
HB_CompressBufLen( <nUncompressedLen> )--> nDefaultBufLen
```

Arguments

<nUncompressedLen>

This is a numeric value indicating the length of the input string to compress in bytes.

Return

The function returns a numeric value which is the number of bytes a preallocated buffer must have to receive the compression result. This is required for the advanced form of HB_Compress().

Info

See also:	HB_Compress()
Category:	ZIP compression, xHarbour extensions
Header:	HbCompress.ch
Source:	rtl\hbcomprs.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_CompressError()

Returns the error code of the last (un)compression.

Syntax

```
HB_CompressError() --> nLastError
```

Return

The function returns a numeric error code of the last compression operation. Passing it to HB_CompressErrorDesc() yields a textual description of an error. When the function returns zero, the last (un)compression operation was successful.

Info

See also:	HB_Compress(), HB_Uncompress(), HB_CompressErrorDesc()
Category:	ZIP compression, xHarbour extensions
Header:	HbCompress.ch
Source:	rtl\hbcomprs.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_CompressErrorDesc()

Returns an error description from a numeric error code.

Syntax

```
HB_CompressErrorDesc( <nErrorCode> ) --> cErrorDescription
```

Arguments

`<nErrorCode>`

This is a numeric value returned from HB_CompressError().

Return

The function returns a textual error description for the passed error code as a character string.

Info

See also:	HB_Compress(), HB_Uncompress(), HB_CompressError()
Category:	ZIP compression, xHarbour extensions
Header:	HbCompress.ch
Source:	rtl\hbcomprs.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Hb_CRC32()

Calculates the checksum for a stream of data using the CRC 32 algorithm.

Syntax

```
HB_CRC32( <cString> ) --> nCRC32
```

Arguments

`<cString>`

This is a character string to calulate the checksum for.

Return

The function returns the calculated checksum as a numeric value.

Description

HB_CRC32() implements the CRC32 algorithm (Cyclic Redundancy Code) for calculating the checksum of a character string. This algorithm computes faster than the algorithm of the HB_MD5() function, but is less reliable. The following table compares the speed of checksum algorithms available in xHarbour:

Algorithm comparison

Function	Algorithm	Relative Speed
HB_CheckSum()	Adler 32 bit	1.00
HB_CRC32()	CRC 32 bit	1.46
HB_MD5()	MD5 128 bit	4.51

Info

See also: HB_CheckSum(), HB_MD5()
Category: Checksum functions, xHarbour extensions
Source: rtl\hbcrc32.c
LIB: xhb.lib
DLL: xhbdll.dll

HB_CreateLen8()

Converts a numeric value to an eight byte character string in network byte order.

Syntax

```
HB_CreateLen8( <nValue> ) --> cBinary
```

Arguments

<nValue>

This is a numeric value.

Return

The function returns a character string of eight bytes length. It holds the passed number in network byte order.

Description

HB_CreateLen8() is a utility function used in xHarbour's serialization system. HB_Serialize() converts any data type to a binary character string that can be transferred between local and remote processes.

Note: refer to function HB_Serialize() for more information on serialization, and read the file source\rtl\hbserial.prg for a usage example of HB_CreateLen8().

Info

See also:	HB_Deserialize(), HB_GetLen8(), HB_Serialize()
Category:	Serialization functions, xHarbour extensions
Source:	rtl\hbsrlraw.c, rtl\hbserial.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_Crypt()

Encrypts a character string.

Syntax

```
HB_Crypt( <cString>, <cKey> ) --> cEncryptedString
```

Arguments

<cString>

This is a character string to encrypt.

<cKey>

This is a character string holding the encryption key.

Return

The function returns the encrypted character string.

Description

Function HB_Crypt() encrypts a character string using the encryption key *<cKey>*. The key should have at least six characters and cannot be longer than 512 characters. The returned encrypted string can be decrypted by passing it to function HB_Decrypt() along with the same encryption key.

Info

See also: HB_Decrypt()
Category: Character functions, Conversion functions, xHarbour extensions
Source: rtl\hbcrypt.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example shows how to encrypt a decrypt a character string.

   PROCEDURE Main
      LOCAL cText  := "Hello world"
      LOCAL cKey   := "xHarbour"
      LOCAL cCipher

      cCipher := HB_Crypt( cText, cKey )

      ? cCipher

      ? HB_Decrypt( cCipher, cKey )
   RETURN
```

HB_Decode()

Provides a functional equivalent for the DO CASE statement.

Syntax

```
HB_Decode( <xInitVal>, <xCase1>, <xRet1> , ;
                      [<xCaseN>, <xRetN>], ;
                      [<xDefault>]         ) --> xReturn

or

HB_Decode( <xInitVal>, <hHash>, [<xDefault>] ) --> xReturn
```

Arguments

<xInitVal>

This is an arbitrary value being tested against *<xCase1>* to *<xCaseN>* or the keys of *<hHash>*.

<xCase>, <xRet>

The parameters *<xCase>* and *<xRet>* must be specified in pairs. *<xCase>* is the value compared with *<xInitVal>*. When both are the same, the return value is *<xRet>*

<xDefault>

This is an optional default value being returned when no match is found. It defaults to NIL.

<hHash>

Instead of passing pairs of *<xCase>* and *<xRet>* parameters, a single Hash() can be passed holding key/value pairs. *<xCase>* is compared with the hash keys. When a match is found, the hash value is returned.

Return

The function returns the value *<xRet>* when *<xInitValue>* matches *<xCase>*. If no match is found, *<xDefault>*, if specified, is returned. Otherwise, the return value is NIL.

Description

HB_Decode() can be viewed as a functional equivalent of to DO CASE statement. An initial value *<xInitVal>* is passed and compared to a list of *<xCase>* values. When a match is found, the corresponding *<xRet>* parameter is returned. *<xCase>* and *<xRet>* can be specified pairwise or may be combined in a single hash.

Info

See also:	DO CASE, Hash(), HB_DecodeOrEmpty()
Category:	Conversion functions, xHarbour extensions
Source:	rtl\decode.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example illustrates possibilities of using HB_Decode()

   PROCEDURE Main
      LOCAL nChoice, hHash := {=>}

      nChoice := 2
      ? HB_Decode( nChoice, 1, "COM1", 2, "COM2", 3, "COM3", "NUL" )
```

```
                                          // result: COM2

   nChoice := 4
   ? HB_Decode( nChoice, 1, "COM1", 2, "COM2", 3, "COM3", "NUL" )
                                          // result: NUL
   hHash := { 0 =>  "Yesterday", ;
              1 =>  "Today"    , ;
              2 =>  "Tomorrow"   }

   nChoice := 0
   ? HB_Decode( nChoice, hHash, "Never" )  // result: Yesterday

   nChoice := 3
   ? HB_Decode( nChoice, hHash, "Never" )  // result: Never

RETURN
```

HB_DecodeOrEmpty()

Provides a functional equivalent for the DO CASE statement.

Syntax

```
HB_DecodeOrEmpty( <xInitVal>        , ;
                  <xCase1>, <xRet1> , ;
                 [<xCaseN>, <xRetN>], ;
                 [<xDefault>]        ) --> xReturn

or

HB_DecodeOrEmpty( <xInitVal>, ;
                  <hHash>    , ;
                 [<xDefault>] ) --> xReturn
```

Arguments

<xInitVal>

This is an arbitrary value being tested against *<xCase1>* to *<xCaseN>* or the keys of *<hHash>*.

<xCase>, <xRet>

The parameters *<xCase>* and *<xRet>* must be specified in pairs. *<xCase>* is the value compared with *<xInitVal>*. When both are the same, the return value is *<xRet>*

<xDefault>

This is an optional default value being returned when no match is found. It defaults to an empty value having the same data type as *<xInitVal>*.

<hHash>

Instead of passing pairs of *<xCase>* and *<xRet>* parameters, a single Hash() can be passed holding key/value pairs. *<xCase>* is compared with the hash keys. When a match is found, the hash value is returned.

Return

The function works exactly the same as HB_Decode(): it returns the value *<xRet>* when *<xInitValue>* matches *<xCase>*. If no match is found *<xDefault>*, if specified, is returned. The difference to HB_Decode() is the default return value when no match is found and *<xDefault>* is not specified: the function returns an empty value of the same data type as *<xInitVal>* as default.

Info

See also:	HB_Decode()
Category:	Conversion functions, xHarbour extensions
Source:	rtl\decode.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_Decrypt()

Decrypts an encrypted character string.

Syntax

```
HB_Decrypt( <cEncryptedString>, <cKey> ) --> cString
```

Arguments

`<cEncryptedString>`

This is a previously encrypted character string to decrypt.

`<cKey>`

This is a character string holding the encryption key.

Return

The function returns the decrypted character string.

Description

Function HB_Decrypt() decrypts a character string previously encrypted with function HB_Crypt(). A successful decryption requires the same key *<cKey>* as used for encrypting the unencrypted string.

Info

See also:	HB_Crypt()
Category:	Character functions, Conversion functions, xHarbour extensions
Source:	rtl\hbcrypt.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_DeserialBegin()

Initiates deserialization of a group of variables of simple or complex data types.

Syntax

```
HB_DeserialBegin( <cBinaryData> ) --> cFirstSerialdata
```

Arguments

<cBinaryData>

This is a character string obtained from one or multiple calls to HB_Serialize(). It holds the binary data of one ore more previously serialized variable(s).

Return

The function returns a binary character string.

Description

Function HB_DeserialBegin() prepares a binary character string holding data of serialized variables for deserialization. Actual data is extracted from <*cBinaryData*> with HB_DeserialNext().

Info

See also:	HB_Deserialize(), HB_DeserialNext(), HB_Serialize()
Category:	Serialization functions, xHarbour extensions
Source:	rtl\hbsrlraw.c, rtl\hbserial.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example outlines when and where HB_DeserialBegin() is used in
// the serialization modules of xHarbour. Values of four different
// data types are serialized and converted back to their original data
// types.

   PROCEDURE Main
      LOCAL cBinary, cSerial, xValue

      cBinary := HB_Serialize( "xHarbour compiler" )
      cBinary += HB_Serialize( 123.45 )
      cBinary += HB_Serialize( StoD( "20070201" ) )
      cBinary += HB_Serialize( { 1, 2, { "a", "b" }, 3 } )

      cSerial := HB_DeserialBegin( cBinary )

      xValue  := HB_DeserialNext( @cSerial )

      DO WHILE xValue <> NIL
         ? Valtype( xValue ), ValToPrg( xValue )

         xValue  := HB_DeserialNext( @cSerial )
      ENDDO

   RETURN
```

HB_DeSerialize()

Converts a binary string back to its original data type.

Syntax

```
HB_DeSerialize( <cBinary> ) --> xValue
```

Arguments

`<cBinary>`

This is a character string returned from HB_Serialize().

Return

The function returns the value previously converted by HB_Serialize().

Description

HB_DeSerialize() reverses the result of HB_Serialize(), i.e. the binary string representation of a value is converted back to its original data type and returned.

Info

See also:	HB_DeserialBegin(), HB_DeserialNext(), HB_RestoreBlock(), HB_Serialize(), RESTORE, SAVE
Category:	Conversion functions, Serialization functions, xHarbour extensions
Source:	rtl\hbserial.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_DeserializeSimple()

Deserializes values of simple data types.

Syntax

```
HB_DeserializeSimple( <cBinaryData> ) --> xValue
```

Arguments

<cBinaryData>

This is a character string obtained from a call to HB_Serialize(). It holds the binary data of a previously serialized variable of data type C, D, L, M, N or U. Refer to function Valtype() for data type encoding.

Return

The function returns the deserialized value.

Description

Function HB_DeserializeSimple() is optimized for deserializing simple data types only. These are Character, Date, Logic, Memo, Numeric and undefined (NIL). Complex data types (Array, Code block, Hash and Object) must be deserialized using the functions HB_DeserialBegin() and HB_DeserialNext().

Info

See also:	HB_Deserialize(), HB_DeserialBegin(), HB_Serialize(), HB_SerializeSimple()
Category:	Serialization functions, xHarbour extensions
Source:	rtl\hbsrlraw.c, rtl\hbserial.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates (de)serialization of a simple data type

   PROCEDURE Main
      LOCAL cBinary, cSerial, xValue

      cBinary := HB_SerializeSimple( Date() )

      xValue  := HB_DeserializeSimple( @cBinary )

      ? ValToPrg( xValue )

   RETURN
```

HB_DeserialNext()

Deserialization the next variable of simple or complex data types.

Syntax

```
HB_DeserialNext( @<cSerial>, <nBytes> ) --> xValue
```

Arguments

<cSerialNext>

This parameter must be passed by reference. The first call to HB_DeserialNext() requires the return value of HB_DeserialBegin() while subsequent calls expect @*<cSerial>* be passed as reference parameter.

<nBytes>

This numeric value specifies the number of bytes to use from *<cSerial>* for decoding a serialized value. It is the return value of HB_GetLen8().

Return

The function returns the next value from a serialized binary string.

Description

HB_DeserialNext() is a utility function used in xHarbour's serialization system. HB_Serialize() converts any data type to a binary character string that can be transferred between local and remote processes.

Note: refer to function HB_Deserialize() for more information on deserialization, and read the file source\rtl\hbserial.prg for a usage example of HB_DeserialNext().

Info

See also:	HB_GetLen8(), HB_Deserialize(), HB_DeserialBegin(), HB_Serialize()
Category:	Serialization functions, xHarbour extensions
Source:	rtl\hbserial.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_DiskSpace()

Returns the free storage space for a disk drive by drive letter.

Syntax

```
HB_DiskSpace( <cDrive>, [<nType>] ) --> nBytes
```

Arguments

<cDrive>

A single character from A to Z specifies the disk drive to query.

<nType>

Optionally, the type of storage space on a drive to query can be specified using #define constants from the FILEIO.CH file. Valid constants are listed below:

Storage space types for HB_DiskSpace()

Constant	Value	Description
HB_DISK_AVAIL *)	0	Free disk space avialable to the application
HB_DISK_FREE	1	Total free disk space
HB_DISK_USED	2	Used disk space
HB_DISK_TOTAL	3	Total disk space

**) default*

Return

The function returns a numeric value indicating the storage space of a disk drive. If no parameters are passed, DiskSpace() returns the free disk space on the current drive that is available to the application.

Description

HB_DiskSpace() determines free and used bytes on a disk drive accepting a drive letter. Some operating systems distinguish between total free bytes on disk and free bytes that are avalaible to the current user, or application. If the current operating system makes no such distinction, HB_DISK_AVAIL and HB_DISK_FREE yield the same results.

Note: if information is requested on a disk that is not available, a runtime error 2018 is generated.

Info

See also: Directory(), DiskName(), DiskSpace(), File(), FOpen(), FSeek(), IsDisk()
Category: Disks and Drives, xHarbour extensions
Source: rtl\disksphb.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example lists two disk storage types for all drives taking
// advantage of the runtime error created for unaccessible drives.

   #include "Fileio.ch"

   PROCEDURE Main
      LOCAL cDrive, i, bError := ErrorBlock( {|e| Break(e) } )

      FOR i:=1 TO 26
         cDrive := Chr(64+i)
```

```
      ? cDrive+":"
      BEGIN SEQUENCE
        ?? HB_DiskSpace( cDrive, HB_DISK_FREE ), ;
           HB_DiskSpace( cDrive, HB_DISK_TOTAL)
      RECOVER
        ?? " not ready or not existent"
      END SEQUENCE
   NEXT

   ErrorBlock( bError )
RETURN
```

HB_DumpVar()

Converts a value to a character string holding human readable text.

Syntax

```
HB_DumpVar( <xValue> ) --> cText
```

Arguments

`<xValue>`

This is a value of any data type.

Return

The function returns a character string containing the passed value as a human readable text.

Description

Function HB_DumpVar() accepts a value of any data type and returns a character string holding a human readable text. This is especially useful for getting data of complex data types, such as Array, Hash and Object, and writing it to a log file. HB_DumpVar() is a plain debugging function.

Info

See also:	CStr(), HB_Serialize(), ValToPrg(), ValToPrgExp()
Category:	Debug functions, xHarbour extensions
Source:	rtl\dumpvar.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_EnumIndex()

Returns the current ordinal position of a FOR EACH iteration.

Syntax

```
HB_EnumIndex() --> nIteration
```

Return

The function returns a numeric value. It is the ordinal position of the current item within a FOR EACH control structure.

Description

Function HB_EnumIndex() is only used within a FOR EACH control structure. It returns a numeric value which can be viewed as the "loop counter" of the FOR EACH statement. HB_EnumIndex() identifies the ordinal position of the item that is currently processed by FOR EACH.

Info

See also:	FOR, FOR EACH, HB_QSelf(), HB_QWith()
Category:	Control structures, Environment functions, xHarbour extensions
Source:	vm\hvm.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example compares a regular FOR..NEXT loop with a
// FOR EACH loop. FOR..NEXT uses a loop counter while FOR EACH
// does not. Instead, the "loop counter" is retrieved with
// function HB_EnumIndex().

   PROCEDURE Main
      LOCAL aArray1 := { "A", "B", "C" }
      LOCAL aArray2 := { "a", "b", "c" }
      LOCAL i, cValue

      CLS

      ? "FOR loop"

      FOR i:=1 TO Len( aArray1 )
         ? i, aArray1[i]

         IF i == 3
            AAdd( aArray1, "D" )
         ENDIF
      NEXT

      ?
      ? "FOR EACH loop"

      FOR EACH cValue IN aArray2
         ? HB_EnumIndex(), cValue

         IF HB_EnumIndex() == 3
            AAdd( aArray2, "d" )
         ENDIF
```

```
   NEXT
RETURN
```

HB_Exec()

Executes a function, procedure or method from its pointer.

Syntax

```
HB_Exec( pFunction_or_Method_Pointer [, <NIL_or_SelfObject> [, ...] ] )

or

HB_Exec( pSomeFunctionPinter [, NIL, Arg1 [, ArgN] ] )

or

HB_Exec( pSomeFunctionPinter [, Self, Arg1 [, ArgN] ] )
```

Arguments

<pFunctionPointer>

This is a pointer to a function or procedure to execute. It can be obtained with the function-reference operator or the HB_FuncPtr() function. If such a pointer is used, the second parameter must be NIL.

<pMethodPointer>

This is a pointer to an object's method to execute. It can be obtained with the HB_ObjMsgPtr() function. If such a pointer is used, the second parameter must be an object.

<oObject>

This is an object whose method is executed.

<nArg1 [, ArgN]>

An optional list of parameters can be specified. They are passed to the function, method or procedure when executed.

Return

HB_Exec() returns the result of the executed function, method or procedure.

Description

Function HB_Exec() accepts as first parameter a pointer to a function, procedure or method. If a function/procedure pointer is used, the second parameter must be NIL. In case of a method pointer, the second parameter must be an object. All other optional parameters are passed on to the executed function, procedure or method.

A pointer is a special data type in xHarbour and is of Valtype()=="P". It represents the memory address of a function, procedure or method. If a memory address is known, the corresponding routine can be executed without the need of looking up the symbolic name. This can lead to performance advantages.

Info

See also: @(), Eval(), HB_ExecFromArray(), HB_FuncPtr(), HB_ObjMsgPtr()
Category: Indirect execution, Pointer functions, xHarbour extensions
Source: vm\eval.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example uses a function pointer of a built-in function and
// a method pointer to a user-defined class for HB_Exec().

   #include "hbclass.ch"

   PROCEDURE Main
      LOCAL oObject
      LOCAL pPointer := ( @QOut() )

      // displays "Hello World"
      HB_Exec( pPointer, NIL, "Hello", "World" )

      oObject := Test():new( "xHarbour" )

      pPointer := HB_ObjMsgPtr( oObject, "display" )

      // displays five times "xHarbour"
      HB_Exec( pPointer, oObject, 5 )
   RETURN


   CLASS Test
     PROTECTED:
     DATA name

     EXPORTED:
     METHOD init CONSTRUCTOR
     METHOD display
   ENDCLASS

   METHOD init( cName ) CLASS Test
      ::name := cName
   RETURN self

   METHOD display( nTimes ) CLASS Test
      LOCAL i
      IF nTimes == NIL
         nTimes := 1
      ENDIF

      FOR i:=1 TO nTimes
         QOut( ::name )
      NEXT
   RETURN self
```

HB_ExecFromArray()

Executes a function, procedure or method indirectly.

Syntax

```
HB_ExecFromArray( <aExecutableArray> ) --> xResult

HB_ExecFromArray( <bBlock>|<cFuncName>|<pFuncPtr> ;
                [,<aParams>] ) --> xResult

HB_ExecFromArray( <oObject>, <cMethodName>|<pMethodPtr> ;
                [,<aParams>] ) --> xResult
```

Arguments

<aExecutableArray>

This is a one dimensional array whose first one or two elements contain executable data while all following elements are passed as parameters to the executable portion of the array. See details in the description.

<bBlock>

This is a code block to be executed. It receives the values stored in the array *<aParams>* as parameters, if specified.

<cFuncName>

This is a character string holding the symbolic name of a function or procedure to be executed. The values stored in the array *<aParams>* are passed as parameters, if specified.

<pFuncPtr>

This is a pointer to a function or procedure to be executed. It can be obtained using function HB_FuncPtr(). The values stored in the array *<aParams>* are passed as parameters, if specified.

<oObject>

If the first parameter is an object (Valtype()=="O"), the second parameter specifies the method to call for the object.

<cMethodName>

This is a character string holding the symbolic name of the method to be executed. The values stored in the array *<aParams>* are passed as parameters, if specified.

<pMethodPrt>

This is a pointer to a method to be executed. It can be obtained using function HB_ObjMsgPtr(). The values stored in the array *<aParams>* are passed as parameters to the method, if specified.

<aParams>

An optional, one dimensional array can be specified. Its elements are passed as parameters to a code block, function, method or procedure.

Return

The function returns the return value of the executed code.

Description

Function HB_ExecFromArray() is used for indirect execution of program code in xHarbour. The function can be called in different ways and accepts parameters of different data types, the first of which must contain an executable value, or item. Depending on the data type of an executable item, HB_ExecFromArray() uses different approaches for execution:

Paremeters and data types of executable items

1st param.	2nd param.	3rd param.	Description
Array	NIL	NIL	Executable array is processed
Character string	<*aParams*>	NIL	Macro operator executes function or procedure
Code block	<*aParams*>	NIL	Eval() executes code block
Pointer	<*aParams*>	NIL	HB_Exec() executes function call
Object	Character string	<*aParams*>	Macro operator executes method of object
Object	Pointer	<*aParams*>	HB_Exec() executes method of object

The following notes explain how HB_ExecFromArray() works depending on the data type of the first parameter passed:

Array

When an array is passed as first parameter, it must be a so called "executable array". This is a one dimensional array whose first element contains an "executable item". An executable item can be of Valtype()=="C" (symbolic name of function or procedure to execute), Valtype()=="B" (code block to execute) or Valtype()=="P" (pointer to function or procedure to execute). The following elements of the executable array are passed as parameters to the executable item.

If the executable item is of Valtype()=="O" (object), the second element of the executable array must contain either a character string specifying the name of the method to call, or it must be a pointer to a method obtained with HB_ObjMsgPtr(). All other values stored in the third to the last element of the executable array are passed as parameters to the object's method.

Character string

When the first parameter is a character string, it must contain the symbolic name of a function or procedure which is visible at runtime. HB_ExecFromArray() uses the macro operator to call the corresponding function or procedure. The second parameter can be optionally a one dimensional array. Its elements are passed as parameters to the function or procedure.

Code block

When the first parameter is a code block, the optional second parameter can be a one dimensional array whose elements are passed as parameters to the code block. It is executed with the Eval() function.

Pointer

When the first parameter is a pointer, it must be obtained with the function-reference operator or the HB_FuncPtr() function. The optional second parameter <*aParams*> contains the parameters to pass to the corresponding function or procedure. It is executed with HB_Exec().

Object

When the first parameter is an object, the second parameter must be one of

1. character string containing the symbolic name of a method to execute
2. pointer to a method otained with HB_ObjMsgPtr()

The optional third parameter <*aParams*> contains the parameters to pass to the corresponding method.

Info

See also: @(), {|| }, { }, Array(), Eval(), HB_Exec(), HB_FuncPtr(), HB_ObjMsgPtr()
Category: Indirect execution, Pointer functions, xHarbour extensions
Source: vm\eval.c
LIB: xhb.lib
DLL: xhbdll.dll

Examples

```
// The example demonstrates various calls to HB_ExecFromArray() using
// an executable array, code block, function name and pointer.

   PROCEDURE Main
      LOCAL cFunc := "QOUT"
      LOCAL pFunc := HB_FuncPtr( cFunc )
      LOCAL aExec := { cFunc, "Hello", "World" }

      // executable array with function name
      HB_ExecFromArray( aExec )       // output: Hello World

      // function pointer and parameter array
      HB_ExecFromArray( pFunc, { "Hi", "there" } )
                                      // output: Hi there

      // function name and parameter array
      ? HB_ExecFromArray( "CDOW", { Date() } )
                                      // output: wednesday

      // executable array with code block and parameter
      bBlock := {|d| Year(d) }
      aExec := { bBlock, Date() }

      ? HB_ExecFromArray( aExec )     // output: 2006

   RETURN

// This example uses an object for HB_ExecFromArray() and
// demonstrates possibilities of calling a method.

   #include "hbclass.ch"

   PROCEDURE Main
      LOCAL pMethod, oObject
      LOCAL aExec, bBlock

      // function name and no parameters
      oObject := HB_ExecFromArray( "NumStat" )

      // executable array with object, method name and parameters
      aExec   := { oObject, "set", 1, 2, 3, 4, 5 }
      HB_ExecFromArray( aExec )

      // executable array with code block and parameter
      bBlock := {|o| o:total() }
      aExec  := { bBlock, oObject }

      ? HB_ExecFromArray( aExec )                // result: 15

      // executable array with object, method pointer and no parameters
      pMethod := HB_ObjMsgPtr( oObject, "average" )
      aExec  := { oObject, pMethod }
```

```
   ? HB_ExecFromArray( aExec )                // result:  3

   // code block and parameter array
   ? HB_ExecFromArray( bBlock, { oObject } )  // result: 15
RETURN


CLASS NumStat
   PROTECTED:
   DATA numbers
   DATA count

   EXPORTED:
   METHOD set
   METHOD average
   METHOD total
ENDCLASS


METHOD set( ... ) CLASS NumStat
   ::numbers := HB_AParams()
   ::count   := Len( ::numbers )
RETURN self


METHOD average CLASS NumStat
RETURN ::total() / ::count


METHOD total CLASS NumStat
   LOCAL nTotal := 0
   LOCAL nNumber

   FOR EACH nNumber IN ::numbers
      nTotal += nNumber
   NEXT
RETURN nTotal
```

HB_FCommit()

Forces a disk write.

Syntax

```
HB_FCommit( <nFileHandle> ) --> NIL
```

Arguments

<nFileHandle>

This is a numeric file handle returned from function FOpen(), FCreate(), or HB_FCreate().

Return

The return value is always NIL.

Description

HB_FCommit() forces a hard disk write and flushes pending memory buffers to the file *<nFileHandle>*, previously opened with FCReate(), FOpen() or HB_FCreate().

The function is only required in multi-threaded programs when multiple threads write data to the same file handle.

Info

See also:	HB_FCreate(), HB_OsError()
Category:	File functions, Low level file functions, xHarbour extensions
Source:	rtl\philes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_FCreate()

Creates and/or opens a binary file.

Syntax

```
HB_FCreate( <cFileName>, [<nFileAttr>], [<nOpenFlags>] ) --> nFileHandle
```

Arguments

<cFilename>

This is a character string holding the name of the file to create. It must include path and file extension. If the path is omitted from *<cFileName>*, the file is created in the current directory.

<nFileAttr>

A numeric value specifying one or more attributes associated with the created or opened file. #define constants from the FILEIO.CH file can be used for *<nFileAttr>* as listed in the table below:

Attributes for binary file creation

Constant	Value	Attribute	Description
FC_NORMAL *)	0	Normal	Creates a normal read/write file
FC_READONLY	1	Read-only	Creates a read-only file
FC_HIDDEN	2	Hidden	Creates a hidden file
FC_SYSTEM	4	System	Creates a system file

**) default attribute*

<nOpenFlags>

A numeric value specifying one or more flags indication how to create a new or open an existing file #define constants from the FILEIO.CH file can be used for *<nOpenFlags>* as listed in the table below:

Flags for file opening or creation

Constant	Value	Description
FO_COMPAT *)	0	Compatibility mode
FO_EXCLUSIVE	16	Exclusive use
FO_DENYWRITE	32	Prevent other applications from writing
FO_DENYREAD	48	Prevent other applications from reading
FO_DENYNONE	64	Allow others to read or write
FO_SHARED	64	Same as FO_DENYNONE
FO_CREAT	0x0100	Create and open file
FO_TRUNC	0x0200	Open existing file and truncate it
FO_EXCL	0x0400	Create and open only if the file does not exist

**) default attribute*

Return

The function returns a numeric file handle to be used later for accessing the file using FRead() or FWrite(). The return value is -1 when the operation fails. Use FError() to determine the cause of failure.

Description

The low-level file function HB_FCreate() is a combination of FCreate() and FOpen(). HB_FCreate() opens an existing file or creates a new one.

If <*cFileName*> specifys an existing file, the third parameter <*nOpenFlags*> can be used to indicate if the file should be truncated, i.e. all data should be removed from the file after opening.

Refer to FRead() and FWrite() for examples that read or write data to a binary file using a file handle returned from HB_FCreate().

Info

See also:	FCreate(), FOpen(), HB_FCommit(), HB_F_Eof()
Category:	File functions, Low level file functions, xHarbour extensions
Header:	FileIO.ch
Source:	rtl\philes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_FEof()

Tests if the end-of-file is reached in the currently selected text file.

Syntax

```
HB_FEof() --> lEof
```

Return

The function returns .T. (true) when the record pointer of the currently selected text file has reached the end-of-file, otherwise .F. (false).

Description

The HB_FEof() function indicates if the record pointer of the currently selected text file has reached the end-of-file mark during file navigation. The end-of-file mark is a state variable that exists for each text file open with HB_FUse().

Info

See also:	FOpen(), FSeek(), HB_FSelect(), HB_FUse()
Category:	File functions, Text file functions, xHarbour extensions
Source:	misc\hb_f.c
LIB:	libmisc.lib

HB_FGoBottom()

Moves the file pointer to the last line in a text file.

Syntax

```
HB_FGoBottom() --> NIL
```

Return

The return value is always NIL.

Description

The function moves the record pointer of the currently selected text file to the last line.

Info

See also:	FOpen(), FSeek(), HB_FGoto(), HB_FGoTop(), HB_FSelect(), HB_FUse()
Category:	File functions, Text file functions, xHarbour extensions
Source:	misc\hb_f.c
LIB:	libmisc.lib

HB_FGoto()

Moves the record pointer to a specific line in the currently selected text file.

Syntax

```
HB_FGoto( <nLine> ) --> NIL
```

Arguments

<nLine>

This numeric parameter identifies the line number to move the record pointer to. It is in the range between 1 and HB_FLastRec().

Return

The return value is always NIL.

Description

The HB_FGoto() function moves the record pointer of the currently selected text file to the line number specified with *<nLine>*.

Info

See also:	FOpen(), FSeek(), HB_FGoBottom(), HB_FGoTop(), HB_FSelect(), HB_FUse()
Category:	File functions, Text file functions, xHarbour extensions
Source:	misc\hb_f.c
LIB:	libmisc.lib

HB_FGoTop()

Moves the record pointer to the begin-of-file.

Syntax

```
HB_FGoTop() --> NIL
```

Return

The return value is always NIL.

Description

The function moves the record pointer of the currently selected text file to the begin-of-file.

Info

See also: FOpen(), FSeek(), HB_FGoBottom(), HB_FGoto(), HB_FSelect(), HB_FUse()
Category: File functions, Text file functions, xHarbour extensions
Source: misc\hb_f.c
LIB: libmisc.lib

HB_FInfo()

Retrieves status information about the currently selected text file.

Syntax

```
HB_Finfo() --> aInfo
```

Return

The function returns a one dimensional array with six elements, holding information about the currently selected text file.

File information array

Element	Description
1	Currently selected text file area
2	Line count according to HB_FLastRec()
3	Current line according to HB_FRecno()
4	Position of file pointer
5	File size
6	End-of-file flag according to HB_FEof()

Info

See also: FOpen(), FSeek(), HB_FSelect(), HB_FUse()
Category: File functions, Text file functions, xHarbour extensions
Source: misc\hb_f.c
LIB: libmisc.lib

HB_FLastRec()

Returns the number of lines in the currently selected text file.

Syntax

```
HB_FLastRec() --> nLineCount
```

Return

The function returns the number of lines in the currently selected text file as a numeric value.

Info

See also: FOpen(), FSeek(), HB_FGoBottom(), HB_FGoto(), HB_FRecno(), HB_FSelect(), HB_FUse()
Category: File functions, Text file functions, xHarbour extensions
Source: misc\hb_f.c
LIB: libmisc.lib

HB_FNameMerge()

Composes a file name from individual components.

Syntax

```
HB_FNameMerge( [<cPath>]     , ;
               [<cFileName>] , ;
               [<cExtension>]  ) --> cResult
```

Arguments

<cPath>

This is a character string holding the path component for a file.

<cFileName>

This is a character string holding the name component for the file.

<Extension>

This is a character string holding the file extension.

Return

The function returns a character string containing all components passed to the function. When no parameter is passed, an empty string ("") is returned.

Description

Function HB_FNameMerge() is used to build a full qualified file name from individual components. They can be obtained by calling HB_FNameSplit().

Info

See also:	CurDir(), CurDrive(), Directory(), File(), HB_FNameSplit()
Category:	Character functions, File functions, xHarbour extensions
Source:	rtl\fnsplit.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The function splits a full qualified file name and
// merges different components

   PROCEDURE Main
      LOCAL cString := "C:\xhb\source\data\test.dbf"
      LOCAL cPath, cFileName, cExtension

      HB_FNameSplit( cString, @cPath, @cFileName, @cExtension )

      ? HB_FNameMerge( , cFileName, cExtension )
      // result: test.dbf

      ? HB_FNameMerge( cPath, cFileName )
      // result: C:\xhb\source\data\test
   RETURN
```

HB_FNameSplit()

Splits a file name into individual components.

Syntax

```
HB_FNameSplit( <cString>      , ;
               [@<cPath>]     , ;
               [@<cFileName>] , ;
               [@<cExtension>]  ) --> NIL
```

Arguments

<cString>

This is a character string to split into different components of a file name.

@<cPath>

If specified, *<cPath>* must be passed by reference. It receives the path component for a file as a character string.

@<cFileName>

If specified, *<cFileName>* must be passed by reference. It receives the name component for a file as a character string.

@<cExtension>

If specified, *<cExtension>* must be passed by reference. It receives the file extension as a character string.

Return

The function returns always NIL. The components of a file name are assigned to the parameters passed by reference.

Description

Function HB_FNameSplit() is used to split a full qualified file name into individual components. They can be merged later with function HB_FNameMerge().

Info

See also:	CurDir(), CurDrive(), Directory(), File(), HB_FNameMerge()
Category:	Character functions, File functions, xHarbour extensions
Source:	rtl\fnsplit.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The function splits a full qualified file name and
// displays the result

   PROCEDURE Main
      LOCAL cString := "C:\xhb\source\data\test.dbf"
      LOCAL cPath, cFileName, cExtension

      ? HB_FNameSplit( cString, @cPath, @cFileName, @cExtension )

      ? cPath                  // result: C:\xhb\source\data\
      ? cFileName              // result: test
      ? cExtension             // result: .dbf
```

```
RETURN
```

HB_FReadAndSkip()

Reads the current line and moves the record pointer.

Syntax

```
HB_FReadAndSkip() --> cLine
```

Return

The function returns the current line in the currently selected text file as a character string.

Description

HB_FReadAndSkip() reads the current line in the currently selected text file and advances the record pointer to the next line. The function is especially designed for reading CSV files that may contain quoted Carriage return, Line feed characters (CRLF). CRLF characters are not recognized as end-of-line markers when they are part of a quoted character string.

Info

See also:	FOpen(), FSeek(), HB_FReadLine(), HB_FReadLN(), HB_FSelect(), HB_FUse()
Category:	File functions, Text file functions, xHarbour extensions
Source:	misc\hb_f.c
LIB:	libmisc.lib

Example

```
// The example fills an array with the lines of a text file,
// taking advantage of implicit record pointer movement of
// function HB_FReadAndSkip().

   PROCEDURE Main
      LOCAL cFile := "Textfile.txt"
      LOCAL aLines, nLine := 0, nCount
      LOCAL nFile

      nFile  := HB_FUse( cFile )
      nCount := HB_FLastRec() + 1
      aLines := Array( nCount )

      DO WHILE ++nLine <= nCount
         aLines[ nLine ] := HB_FReadAndSkip()
      ENDDO

      HB_FUse()

      AEval( aLines, {|cLine| QOut( cLine ) } )
   RETURN
```

HB_FReadLine()

Extracts the next line from a text file.

Syntax

```
HB_FReadLine( <nFileHandle> , ;
              @<cLine>      , ;
             [<caEndOfLine>], ;
             [<nLineLength>]  ) --> nReturn
```

Arguments

`<nFileHandle>`

This is a numeric file handle returned from function FOpen().

`@<cLine>`

A memory variable that must be passed by reference to HB_FReadLine(). It receives the character string read from the text file. <*cLine*> does not need to be initialized.

`<caEndOfLine>`

The parameter is optional. It is either a character string containing the end-of-line characters, or an array of character strings, each element of which contains end-of-line characters. The default value for <*caEndOfLine*> is Chr(13)+Chr(10) (Carriage return, Line feed).

`<nLineLength>`

This is an optional numeric value specifying the maximum line length the function searches for the next <*caEndOfLine*>. The default value is 4096.

Return

The function returns 0 when the next text line is read from the file. A value not equal zero is returned when no more lines are available to read.

Description

HB_FReadLine() is an efficient extraction routine for getting the next available line from an ASCII text file. It differs from FReadStr() in allowing a programmer to define the end-of-line character(s) HB_FReadLine() should recognize. Also, the maximum length of a line to search for the next end-of-line character(s) can be specified. The default value of 4096 is sufficient for most common text files.

Info

See also:	FOpen(), FClose(), FCreate(), FReadStr(), MemoLine(), MemoRead()
Category:	Low level file functions, xHarbour extensions
Source:	rtl\fnsplit.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates an efficient way of loading single lines
// from an ASCII file into an array.

   PROCEDURE Main
      LOCAL cFileName   := "HB_FReadLine.prg"
      LOCAL aLines      := {}
      LOCAL nFileHandle := FOpen( cFileName )
      LOCAL cLine
```

```
      DO WHILE HB_FReadLine( nFileHandle, @cLine ) == 0
         AAdd( aLines, cLine )
      ENDDO
      // Add the last line
      AAdd( aLines, cLine )

      FClose( nFileHandle )

      ? Len( aLines )
      AEval( aLines, {|c| QOut( c ) } )
   RETURN
```

HB_FreadLN()

Reads the current line and without moving the record pointer.

Syntax

```
HB_FreadLN() --> cCurrentLine
```

Return

The function returns the current line in the currently selected text file as a character string.

Description

HB_FreadLN() reads the current line in the currently selected text file and leaves the record pointer unchanged. The current line is the posrtion of the text file beginning at the file pointer position up to the next Carriage return, Line feed pair.

Use function HB_FSkip() to advance the record pointer to the next line.

Info

See also: FOpen(), FSeek(), HB_FReadAndSkip(), HB_FReadLine(), HB_FSkip(), HB_FSelect(), HB_FUse()
Category: File functions, Text file functions, xHarbour extensions
Source: misc\hb_f.c
LIB: libmisc.lib

Example

```
// The example fills an array with the lines of a text file,
// by skipping through the entire file.

   PROCEDURE Main
      LOCAL aLines := {}
      LOCAL cFile  := "Textfile.txt"
      LOCAL nFile

      nFile := HB_FUse( cFile )

      DO WHILE .NOT. HB_FEof()
         AAdd( aLines, HB_FReadLN() )
         HB_FSkip(1)
      ENDDO

      HB_FUse()

      AEval( aLines, {|cLine| QOut( cLine ) } )
   RETURN
```

HB_FRecno()

Returns the current line number of the currently selected text file.

Syntax

```
HB_FRecno() --> nLineNumber
```

Return

The function returns the position of the record pointer (line number) of the currently selected text file as a numeric value.

Info

See also: FOpen(), FSeek(), HB_FLastRec(), HB_FGoto(), HB_FReadLN(), HB_FSkip(), HB_FSelect(), HB_FUse()
Category: File functions, Text file functions, xHarbour extensions
Source: misc\hb_f.c
LIB: libmisc.lib

Example

```
// The example lists the lines of a text file along with
// their line numbers

   PROCEDURE Main
      LOCAL cFile  := "HB_FRecno.prg"

      nFile := HB_FUse( cFile )

      DO WHILE .NOT. HB_FEof()
         ? "Line #" + LTrim( Str( HB_FRecno() ) ), HB_FReadLN()
         HB_FSkip(1)
      ENDDO

      HB_FUse()
   RETURN
```

HB_FSelect()

Queries or changes the currently selected text file area.

Syntax

```
HB_FSelect( [<nNewArea>] ) --> nOldArea
```

Arguments

<nNewArea>

This is a numeric value between 1 and 10 specifying the text file area to select as current. The number of text file areas is limited to 10.

Return

The function returns the previously selected text file area as a numeric value.

Description

Text files opened with HB_FUse() are open in one of ten text file areas. This is similar to the work area concept of databases. However, a text file area cannot be addressed via alias names. All text file functions operate only in the currently selected text file area.

Info

See also:	HB_FLastRec(), HB_FGoto(), HB_FReadLN(), HB_FSkip(), HB_FUse()
Category:	File functions, Text file functions, xHarbour extensions
Source:	misc\hb_f.c
LIB:	libmisc.lib

HB_FSize()

Returns the size of a file in bytes.

Syntax

```
HB_FSize( <cFileName> ) --> nFileSize
```

Arguments

<cFileName>

This is a character string holding the name of the file to query. It must include path and file extension. If the path is omitted from *<cFileName>*, the file is searched in the current directory.

Return

The function returns a numeric value indicating the size of the file *<cFileName>* in bytes. If the file does not exist, the return value is 0.

Description

Function HB_FSize() is used to determine the size of a single file in bytes. Note that the return value of this function is 0 for an existing file with no contents and for a none existing file. Use function File() to check for the existence of a file.

Info

See also:	Directory(), FileStats(), FCreate(), FOpen()
Category:	Low level file functions, xHarbour extensions
Source:	rtl\fssize.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays the size of a single file

   PROCEDURE Main
      LOCAL cFileName := "HB_FSize.prg"

      ? HB_FSize( cFileName )
   RETURN
```

HB_FSkip()

Moves the record pointer in the currently selected text file.

Syntax

```
HB_FSkip( [<nLines>] ) --> NIL
```

Arguments

<nLines>

This numeric parameter specifies the numbers of lines to move the record pointer in the currently selected text file. Positive values for *<nLines>* move the record pointer forwards (towards the end of file), negative values move it backwards. The default value is 1, i.e. calling HB_FSkip() with no parameter advances the record pointer to the next text line.

Return

The return value is always NIL.

Description

One record in a text file is one line of text terminated by a Carriage return, Line feed pair. HB_FSkip() moves the record pointer by *<nLines>* lines. Note, however, that the record pointer cannot be moved outside the range between 1 and HB_FLastRec().

Info

See also:	HB_FEof(), HB_FLastRec(), HB_FGoto(), HB_FReadLN(), HB_FSelect(), HB_FUse()
Category:	File functions, Text file functions, xHarbour extensions
Source:	misc\hb_f.c
LIB:	libmisc.lib

HB_FTempCreate()

Creates and opens a temporary file.

Syntax

```
HB_FTempCreate( [<cTempDir>] , ;
                [<cPrefix>]  , ;
                [<nFileAttr>], ;
               [@<cFileName>]  ) --> nFileHandle
```

Arguments

<cTempDir>

This is an optional character string specifying the directory where to create the temporary file. It defaults to the operating system variable SET TEMP or SET TMP. Operating system variables can be queried with function GetEnv().

<cPrefix>

This is an optional character string of three characters specifying the prefix to use for the name of the temporary file. It defaults to the string "xht".

<nFileAttr>

This is an optional numeric value defining how to create and open the temporary file. See function FCreate() for a description of *<nFileAttr>*.

@<cFileName>

If specified, *<cFileName>* must be passed by reference. It receives the file name of the created temporary file as a character string.

Return

The function returns a numeric value > 0 when the temporary file is successfully created. This is the file handle of the created temporary file.

Description

Function HB_FTempCreate() creates a temporary file and opens it. The return value is the file handle of the temporary file. It can be used with low level file functions, such as FWrite() to store data in the file.

Temporary files are often required in an application. HB_FTempCreate() guarantees that the file name of the newly created file is unique, so that no existing file will be overwritten. To obtain the file name of the temporary file, parameter *<cFileName>* must be passed by reference. It can then later be passed to function FErase() in order to remove the temporary file from disk.

Info

See also:	FCreate(), FErase(), FWrite(), GetEnv()
Category:	File functions, Low level file functions, xHarbour extensions
Source:	rtl\fstemp.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example implements a user defined function that creates
// a temporary file and returns its file name. The temporary
// file is then erased.
```

```
PROCEDURE Main
   LOCAL cTempFile := TempFileName()

   ? cTempFile            // result: C:\temp\xht93.tmp

   FErase( cTempFile )
RETURN


FUNCTION TempFileName()
   LOCAL nFileHandle
   LOCAL cFileName

   nFileHandle := HB_FTempCreate( ,,, @cFileName )

   IF nFileHandle > 0
      FClose( nFileHandle )
   ENDIF
RETURN cFileName
```

HB_FuncPtr()

Obtains the pointer to a function or procedure.

Syntax

```
HB_FuncPtr( <cFuncName> ) --> pFuncPointer
```

Arguments

<cFuncName>

This is a character string holding the symbolic name of an xHarbour function, or a declared user-defined FUNCTION or PROCEDURE.

Return

The return value is a pointer to *<cFuncName>* or NIL, when the symbolic name of the function or procedure does not exist at runtime.

Description

Function HB_FuncPtr() retrieves the pointer to an xHarbour function or a user-defined FUNCTION or PROCEDURE whose symbolic name must exist at runtime. Pointers to STATIC declared functions and procedures cannot be obtained at runtime.

A function pointer is the memory address of a function or procedure. It can be used with HB_Exec() or HB_ExecFromArray() to execute a function from its pointer. This is similar to embedding a function call within a code block and passing the block to the Eval() function. A code block can contain multiple function calls. If only one function is to be executed indirectly, a function pointer along with HB_Exec() or HB_ExecFromArray() is faster than code block execution.

The advantage of a function pointer is that the dynamic lookup of the symbolic function name can be avoided once the pointer is obtained. This can improve the performance of time critical functions considerably when they must be called very often.

Info

See also:	@(), HB_Exec(), HB_ExecFromArray(), HB_ObjMsgPtr()
Category:	Indirect execution, xHarbour extensions
Source:	vm\hvm.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_FUse()

Opens or closes a text file in a text file area.

Syntax

```
HB_FUse( [<cTextfile>], [<nMode>] ) --> nFilehandle
```

Arguments

<cTextfile>

This is a character string holding the name of the text file to open. It must include path and file extension. If the path is omitted from *<cTextfile>*, the file is searched in the current directory.

If no parameter is passed, the text file open in the currently selected text file area is closed.

<nMode>

A numeric value specifying the open mode and access rights for the file. #define constants from the FILEIO.CH file can be used for *<nMode>* as listed in the table below:

File open modes

Constant	Value	Description
FO_READ *)	0	Open file for reading
FO_WRITE	1	Open file for writing
FO_READWRITE	2	Open file for reading and writing

**) default*

Constants that define the access or file sharing rights can be added to an FO_* constant. They specify how file access is granted to other applications in a network environment.

File sharing modes

Constant	Value	Description
FO_COMPAT *)	0	Compatibility mode
FO_EXCLUSIVE	16	Exclusive use
FO_DENYWRITE	32	Prevent other applications from writing
FO_DENYREAD	48	Prevent other applications from reading
FO_DENYNONE	64	Allow others to read or write
FO_SHARED	64	Same as FO_DENYNONE

**) default*

Return

The function returns a numeric file handle when the text file is successfully open, or zero on failure.

Description

HB_FUse() opens an existing text file named *<cTextfile>* in the currently selected text file area. All text file functions operate only in the currently selected text file area. Call HB_FSelect() to change a text file area for opening multiple text files.

The file open in the currently selected text file area is closed when HB_FUse() is called without parameters.

Info

See also: FOpen(), FSeek(), HB_FLastRec(), HB_FGoto(), HB_FReadLN(), HB_FSkip(), HB_FSelect()
Category: File functions, Text file functions, xHarbour extensions
Source: misc\hb_f.c
LIB: libmisc.lib

Example

```
// The example implements a simple text file viewer using a
// TBrowse object and text file functions.

   #include "Common.ch"
   #include "Inkey.ch"
   #include "TBrowse.ch"

   PROCEDURE Main( cFile )
      LOCAL nFile, oTBrowse

      SET CURSOR OFF
      nFile := HB_FUse( cFile )

      IF nFile < 1
         ? "File not found", cFile
         QUIT
      ENDIF

      oTBrowse := TxtBrowse()

      RunTxtBrowse( oTBrowse )

      HB_FUse()
   RETURN

   PROCEDURE RunTxtBrowse( oTBrowse )
      LOCAL nKey, nLen, lRun := .T.

      DO WHILE lRun
         oTBrowse:forceStable()
         nKey := Inkey(0)

         SWITCH nKey
         CASE K_LEFT
            IF oTBrowse:cargo > 1
               oTBrowse:cargo --
               oTBrowse:refreshAll()
            ENDIF
            EXIT

         CASE K_RIGHT
            oTBrowse:cargo ++
            oTBrowse:refreshAll()
            EXIT

         CASE K_HOME
            IF oTBrowse:cargo > 1
               oTBrowse:cargo := 1
               oTBrowse:refreshAll()
            ENDIF
            EXIT
```

```
      CASE K_END
         nLen := Len( HB_FReadLN() )
         IF nLen-oTBrowse:cargo+1 > 72
            oTBrowse:cargo := nLen - 72 + 1
            oTBrowse:refreshAll()
         ENDIF
         EXIT

      CASE K_CTRL_HOME
         EXIT

      CASE K_CTRL_END
         EXIT

      DEFAULT
         IF oTBrowse:applyKey( nKey ) == TBR_EXIT
            lRun := .F.
         ENDIF
      END
   ENDDO
RETURN


FUNCTION TxTBrowse( nT, nL, nB, nR )
   LOCAL oTBrowse, oTBCol1, oTBCol2

   DEFAULT nT TO 0, ;
           nL TO 0, ;
           nB TO MaxRow(), ;
           nR TO MaxCol()

   oTBrowse := TBrowseNew( nT, nL, nB, nR )
   oTBrowse:cargo := 1

   oTBCol1  := TBColumnNew( " ",  ;
                        {|| Padr(HB_FRecno(),5)+":" } )

   oTBCol2  := TBColumnNew( " ",  ;
                        {|| Padr( SubStr( HB_FReadLN(), oTBrowse:cargo), 72 ) } )

   WITH OBJECT oTBrowse
      :addColumn( oTbCol1 )
      :addColumn( oTbCol2 )

      :goTopBlock    := {|| HB_FGotop() }
      :goBottomBlock := {|| HB_FGoBottom() }
      :skipBlock     := {|n| TxtSkipper(n) }
      :colPos        := 2
   END

RETURN oTBrowse


FUNCTION TxtSkipper( nRequest )
   LOCAL nSkip := 0

   DO CASE
   CASE nRequest == 0
   CASE nRequest < 0

      DO WHILE nSkip > nRequest .AND. HB_FRecno() > 1
```

```
            HB_FSkip(-1)
            nSkip --
         ENDDO

      CASE nRequest > 0
         DO WHILE nSkip < nRequest
            HB_FSkip(1)
            IF HB_FEof()
               EXIT
            ENDIF
            nSkip ++
         ENDDO
      ENDCASE

   RETURN nSkip
```

HB_F_Eof()

Tests for End-of-file with binary files.

Syntax

```
HB_F_Eof( <nFileHandle> ) --> lIsEndOfFile
```

Arguments

<nFileHandle>

This is a numeric file handle returned from function FOpen(), FCreate(), or HB_FCreate().

Return

The function returns .T. (true) when the file pointer has reached the End-of-file and no more bytes can be read from *<nFileHandle>* with FRead(). Otherwise, .F. (false) is returned.

Info

See also:	FRead(), FSeek(), HB_FCreate(), HB_FCommit()
Category:	File functions, Low level file functions, xHarbour extensions
Source:	rtl\philes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_GCAll()

Scans the memory and releases all garbage memory blocks.

Syntax

```
HB_GCAll( [<lForce>] ) --> NIL
```

Arguments

<lForce>

Passing .T. (true) enforces a complete garbage collection, even if there are only few memory objects to collect. If *<lForce>* is omitted, or set to .F. (false), the garbage collector is not invoked when there is little or no garbage to collect.

Return

The return value is always NIL.

Description

xHarbour's garbage collector is normally invoked automatically during idle states. An idle state is the state of the xHarbour virtual machine (VM) when it waits for user input from the keyboard or the mouse. The VM enters idle state during Inkey() calls. All applications that do not use Inkey() function calls can signal the idle state with a call to the HB_IdleState() function.

Alternatively, garbage collection can be enforced programmatically with HB_GCAll(.T.). This can be advantageous when there is massive use of memory during a loop, for exemple, that includes no user input and no idle state is signalled.

Info

See also:	HB_GCStep(), HB_IdleState(), Memory()
Category:	Debug functions, Environment functions, xHarbour extensions
Source:	vm\garbage.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_GCStep()

Invokes the garbage collector for one collection cycle.

Syntax

```
HB_GCStep() --> NIL
```

Return

The return value is always NIL.

Description

Function HB_GCStep() invokes the garbage collector for one garbage collection cycle, thus allowing for incremental garbage collection. Refer to HB_GCAll() for more information on the garbage collector.

Info

See also:	HB_GCAll(), Memory()
Category:	Debug functions, Environment functions, xHarbour extensions
Source:	vm\garbage.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_GetLen8()

Retrieves the length of data in a serialized binary string.

Syntax

```
HB_GetLen8( <cBinaryData> ) --> nBytes
```

Arguments

<cBinaryData>

This is a character string obtained from one or multiple calls to HB_Serialize(). It holds the binary data of one ore more previously serialized variable(s).

Return

The function extracts the eight bytes encoding the length of binary data to read from a serialized binary character string and returns it as a numeric value.

Description

HB_GetLen8() is a utility function used in xHarbour's serialization system. It reads eight bytes previously encoded with HB_CreateLen8() and determines the length of the subsequent serialized character string that must be deserialized to obtain the original data.

Note: the first byte of a serialized string contains a single letter encoding the data type (the letter is Valtype() compliant). The next eight bytes are the length of the binary data, and the rest is the binary data.

Refer to the file source\rtl\hbserial.prg for a usage example of HB_GetLen8().

Info

See also:	HB_CreateLen8(), HB_Deserialize(), HB_Serialize()
Category:	Serialization functions, xHarbour extensions
Source:	rtl\hbsrlraw.c, rtl\hbserial.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_IdleAdd()

Adds a background task for being executed during idle states.

Syntax

```
HB_IdleAdd( <bAction> ) --> nTaskHandle
```

Arguments

<bAction>

This is a codeblock that will be executed during idle states. The code block receives no parameters when executed.

Return

The function returns a numeric task handle that must be preserved for other idle processing functions, such as HB_IdleDel().

Description

Function HB_IdleAdd() adds the passed codeblock to the list of background tasks that will be executed when the main program enters an idle state. There is no limit for the number of idle tasks.

The idle state is the state of the xHarbour virtual machine (VM) when it waits for user input from the keyboard or the mouse. The VM enters idle state during Inkey() calls. All applications that do not use Inkey() function calls can signal the idle state with a call to the HB_IdleState() function.

Note: tasks for regular background processing are created with function HB_BackGroundAdd().

An alternative for background tasks is provided with threads. Refer to function StartThread() for running parts of an application simultaneously in multiple threads.

Info

See also:	HB_BackGroundAdd(), HB_IdleDel(), HB_IdleState(), HB_IdleSleepMSec(), StartThread()
Category:	Background processing, xHarbour extensions
Source:	rtl\idle.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// This example halts the program for 10 seconds with the Inkey()
// function. A message is continuously displayed until either
// a key is pressed or 10 seconds have elapsed.

   PROCEDURE Main
      LOCAL nCounter := 10
      CLS
      DispOutAtSetPos( .F. )

      nTask := HB_IdleAdd( {|| DispMsg( --nCounter ) } )

      ? "Test program for idle state"
      ?
      ?
      InKey( 10 )

      HB_IdleDel( nTask )
```

```
   ?
   ? "Program resumes"
RETURN


PROCEDURE DispMsg( nCounter )
   LOCAL cMsg

   cMsg := "Program resumes in "
   cMsg += Ltrim( Str(nCounter) )
   cMsg += " seconds unless you press a key"

   DispoutAt( Row(), Col(), cMsg )
   Hb_IdleSleep( 1 )
RETURN
```

HB_IdleDel()

Removes a task from the list of idle tasks.

Syntax

```
HB_IdleDel( <nTaskHandle> ) --> bAction
```

Arguments

`<nTaskHandle>`

This is the numeric task handle as returned by HB_IdleAdd().

Return

The function returns the code block assoiated with the task handle, or NIL if an invalid task handle is specified.

Description

Function HB_IdleDel() removes the task associated with the passed task handle from the internal list of idle tasks. The task handle must be a value returned by a previous call to the HB_IdleAdd() function. If the specified task exists, it is deactivated and the associated code block is returned.

Info

See also:	HB_BackGroundAdd(), HB_IdleAdd(), HB_IdleState(), HB_IdleSleepMSec()
Category:	Background processing, xHarbour extensions
Source:	rtl\idle.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_IdleReset()

Resets the internal counter of idle tasks.

Syntax

```
HB_IdleReset() --> NIL
```

Return

The return value is always NIL.

Description

Function HB_IdleReset() resets the internal counter identifying the next idle task to be executed to 1. As a result, the next cycle for idle processing starts with the first task defined.

Info

See also:	HB_BackGroundAdd(), HB_IdleAdd(), HB_IdleDel(), HB_IdleState()
Category:	Background processing, xHarbour extensions
Source:	rtl\idle.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_IdleSleep()

Halts idle task processing for a number of seconds.

Syntax

```
HB_IdleSleep( <nSeconds> ) --> NIL
```

Arguments

<nSeconds>

This is a numeric value specifying the number of seconds to wait until idle task processing resumes.

Return

The return value is always NIL.

Description

Function HB_IdleSleep() defines a time interval in seconds during which idle task processing is paused. Idle tasks are executed again after *<nSeconds>* seconds have elapsed.

Info

See also:	HB_BackGroundAdd(), HB_IdleAdd(), HB_IdleState(), HB_IdleSleepMSec(), HB_IdleWaitNoCPU()
Category:	Background processing, xHarbour extensions
Source:	rtl\idle.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_IdleSleepMSec()

Queries or changes the default time interval for idle task processing.

Syntax

```
HB_IdleSleepMSec( [<nNewInterval>] ) --> nOldInterval
```

Arguments

`<nNewInterval>`

This is a numeric value specifying the time interval to wait between the execution of idle tasks. The unit is 1/1000th of a second (milliseconds). The default value is 20.

Return

The function returns the previous wait interval for idle tasks as a numeric value.

Description

Function HB_IdleSleepMSec() queries or changes the number of milliseconds after which the next idle task is executed. This allows for fine tuning idle task processing and to optimize CPU usage on multi-user systems, such as Terminal Server, where a minimum amount of idle task processing is desirable.

Info

See also:	HB_IdleAdd(), HB_IdleDel(), HB_IdleState(), HB_IdleSleepMSec(), HB_IdleWaitNoCPU()
Category:	Background processing, xHarbour extensions
Source:	rtl\idle.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_IdleState()

Signals an idle state.

Syntax

```
HB_IdleState() --> NIL
```

Return

The return value is always NIL.

Description

Function HB_IdleState() is used to explicitely signal an idle state while the program is not wating for user input. The function requests garbage collection and executes a single idle task defined with the codeblock passed to the HB_IdleAdd() function. Each call to HB_IdleState() function evaluates the next idle task in the order of task creation.

Info

See also:	HB_GCAll(), HB_IdleAdd(), HB_IdleDel(), HB_IdleSleepMSec()
Category:	Background processing, xHarbour extensions
Source:	rtl\idle.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_IdleWaitNoCPU()

Toggles the mode for CPU usage in Idle wait states.

Syntax

```
HB_IdleWaitNoCPU( [<lNewMode>] ) --> lOldMode
```

Arguments

`<lNewMode>`

This is a logical value determining the mode how a wait state is executed in idle tasks.

Return

The function returns the previous mode for wait states as a logical value.

Description

By default, an idle wait state is accomplished by an internal function that monitors a time interval. This requires CPU resources. When *<lNewMode>* is set to .T. (true), a wait state is supervised by the operating system, which requires no extra CPU resources.

Info

See also:	HB_BackGroundAdd(), HB_IdleAdd(), HB_IdleState(), HB_IdleSleepMSec()
Category:	Background processing, xHarbour extensions
Source:	rtl\idle.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_IsArray()

Tests if the value returned by an expression is an array.

Syntax

```
HB_IsArray( <expression> ) --> lIsArray
```

Arguments

<expression>

This is an expression of any data type.

Return

The function returns .T. (true) if the value returned by *<expression>* is an array, otherwise .F. (false) is returned.

Description

Function HB_IsArray() is used to test if a variable contains an array or if the result of an expression is of data type Array. The function can be used as a replacement for the expression: Valtype(*<expression>*)=="A".

Info

See also:	HB_IsBlock(), HB_IsByRef(), HB_IsDate(), HB_IsHash(), HB_IsLogical(), HB_IsMemo(), HB_IsNIL(), HB_IsNull(), HB_IsNumeric(), HB_IsObject(), HB_IsPointer(), HB_IsString(), Type(), Valtype()
Category:	Debug functions, Logical functions, xHarbour extensions
Source:	rtl\valtype.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_IsBlock()

Tests if the value returned by an expression is a Code block.

Syntax

```
HB_IsBlock( <expression> ) --> lIsCodeblock
```

Arguments

<expression>

This is an expression of any data type.

Return

The function returns .T. (true) if the value returned by *<expression>* is a code block, otherwise .F. (false) is returned.

Description

Function HB_IsBlock() is used to test if a variable contains a code block or if the result of an expression is of data type Code block. The function can be used as a replacement for the expression: Valtype(*<expression>*)=="B".

Info

See also: HB_IsArray(), HB_IsByRef(), HB_IsDate(), HB_IsHash(), HB_IsLogical(), HB_IsMemo(), HB_IsNIL(), HB_IsNull(), HB_IsNumeric(), HB_IsObject(), HB_IsPointer(), HB_IsString(), Type(), Valtype()
Category: Debug functions, Logical functions, xHarbour extensions
Source: rtl\valtype.c
LIB: xhb.lib
DLL: xhbdll.dll

HB_IsByRef

Tests if a parameter is passed by reference.

Syntax

```
HB_IsByRef( @<variable> ) --> lIsByReference
```

Arguments

`@<variable>`

<variable> is the symbolic name of any parameter passed to a function, method or procedure.

Return

The function returns .T. (true) if the variable *<variable>* is passed by reference, otherwise .F. (false) is returned.

Description

Function HB_IsByRef() is used to test if a variable is passed by reference to a function, method or procedure. This requires to pass *<variable>* by reference to HB_IsByRef(). If *<variable>* is not passed by reference to HB_IsByRef(), a runtime error is raised.

Info

See also:	HB_IsArray(), HB_IsBlock(), HB_IsDate(), HB_IsHash(), HB_IsLogical(), HB_IsMemo(), HB_IsNIL(), HB_IsNull(), HB_IsNumeric(), HB_IsObject(), HB_IsPointer(), HB_IsString(), Type(), Valtype()
Category:	Debug functions, Logical functions, xHarbour extensions
Source:	rtl\valtype.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates how to test if a variable is
// passed by reference to a subroutine.

   PROCEDURE Main
      LOCAL cMsg := "Hello World"
      LOCAL dDate:= Date()

      Test( cMsg, @dDate )
   RETURN

   PROCEDURE Test( p1, p2 )
      ? "First parameter is "

      IF HB_IsByRef( @p1 )
         ?? "passed by reference"
      ELSE
         ?? "not passed by reference"
      ENDIF

      ? "Second parameter is "

      IF HB_IsByRef( @p2 )
         ?? "passed by reference"
      ELSE
         ?? "not passed by reference"
```

```
      ENDIF
   RETURN
```

HB_IsDate()

Tests if the value returned by an expression is a Date.

Syntax

```
HB_IsDate( <expression> ) --> lIsDate
```

Arguments

<expression>

This is an expression of any data type.

Return

The function returns .T. (true) if the value returned by <*expression*> is of data type Date, otherwise .F. (false) is returned.

Description

Function HB_IsDate() is used to test if a variable contains a date value or if the result of an expression is of data type Date. The function can be used as a replacement for the expression: Valtype(<*expression*>)=="D".

Info

See also: HB_IsArray(), HB_IsBlock(), HB_IsByRef(), HB_IsDateTime(), HB_IsHash(), HB_IsLogical(), HB_IsMemo(), HB_IsNIL(), HB_IsNull(), HB_IsNumeric(), HB_IsObject(), HB_IsPointer(), HB_IsString(), Type(), Valtype()
Category: Debug functions, Logical functions, xHarbour extensions
Source: rtl\valtype.c
LIB: xhb.lib
DLL: xhbdll.dll

HB_IsDateTime()

Tests if the value returned by an expression is a DateTime value.

Syntax

```
HB_IsDateTime( <expression> ) --> lIsDateTime
```

Arguments

<expression>

This is an expression of any data type.

Return

The function returns .T. (true) if the value returned by *<expression>* is a DateTime value, otherwise .F. (false) is returned.

Description

Function HB_IsDateTime() is used to test if a variable contains a DateTime value or if the result of an expression is a DateTime value. The function is the only way to distinguish a DateTime value from a Date value, since Valtype(*<expression>*) yields "D" for both, Date and DateTime.

Info

See also: HB_IsArray(), HB_IsBlock(), HB_IsByRef(), HB_IsDate(), HB_IsHash(), HB_IsLogical(), HB_IsMemo(), HB_IsNIL(), HB_IsNull(), HB_IsNumeric(), HB_IsObject(), HB_IsPointer(), HB_IsString(), Type(), Valtype()
Category: Debug functions, Logical functions, xHarbour extensions
Source: rtl\dateshb.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates how to ditingush Date from DateTime values

   PROCEDURE Main
      LOCAL dDate     := DateTime()
      LOCAL dDateTime := Date()

      ? dDate                      // result: 04/27/07 13:21:26.00
      ? dDateTime                  // result: 04/27/07

      ? Valtype( dDate )           // result: D
      ? Valtype( dDateTime )       // result: D

      ? HB_IsDate( dDate )         // result: .T.
      ? HB_IsDate( dDateTime )     // result: .T.

      ? HB_IsDateTime( dDate )     // result: .T.
      ? HB_IsDateTime( dDateTime ) // result: .F.
   RETURN
```

HB_IsHash()

Tests if the value returned by an expression is a Hash.

Syntax

```
HB_IsHash( <expression> ) --> lIsHash
```

Arguments

`<expression>`

This is an expression of any data type.

Return

The function returns .T. (true) if the value returned by <*expression*> is of data type Hash, otherwise .F. (false) is returned.

Description

Function HB_IsHash() is used to test if a variable contains a hash value or if the result of an expression is of data type Hash. The function can be used as a replacement for the expression: Valtype(<*expression*>)=="H".

Info

See also: HB_IsArray(), HB_IsBlock(), HB_IsByRef(), HB_IsDate(), HB_IsLogical(), HB_IsMemo(), HB_IsNIL(), HB_IsNull(), HB_IsNumeric(), HB_IsObject(), HB_IsPointer(), HB_IsString(), Type(), Valtype()
Category: Debug functions, Logical functions, xHarbour extensions
Source: rtl\valtype.c
LIB: xhb.lib
DLL: xhbdll.dll

HB_IsLogical()

Tests if the value returned by an expression is a logical value.

Syntax

```
HB_IsLogical( <expression> ) --> lIsLogical
```

Arguments

<expression>

This is an expression of any data type.

Return

The function returns .T. (true) if the value returned by *<expression>* is of data type Logical, otherwise .F. (false) is returned.

Description

Function HB_IsLogical() is used to test if a variable contains a logical value or if the result of an expression is of data type Logical. The function can be used as a replacement for the expression: Valtype(*<expression>*)=="L".

Info

See also: HB_IsArray(), HB_IsBlock(), HB_IsByRef(), HB_IsDate(), HB_IsHash(), HB_IsMemo(), HB_IsNIL(), HB_IsNull(), HB_IsNumeric(), HB_IsObject(), HB_IsPointer(), HB_IsString(), Type(), Valtype()
Category: Debug functions, Logical functions, xHarbour extensions
Source: rtl\valtype.c
LIB: xhb.lib
DLL: xhbdll.dll

HB_IsMemo()

Tests if the value returned by an expression is a Memo value.

Syntax

```
HB_IsMemo( <expression> ) --> lIsMemo
```

Arguments

<expression>

This is an expression of any data type.

Return

The function returns .T. (true) if the value returned by *<expression>* is of data type Memo, otherwise .F. (false) is returned.

Description

Function HB_IsMemo() is used to test if a variable contains a memo value or if the result of an expression is of data type Memo. The function can be used as a replacement for the expression: Valtype(*<expression>*)=="M".

Info

See also:	HB_IsArray(), HB_IsBlock(), HB_IsByRef(), HB_IsDate(), HB_IsHash(), HB_IsLogical(), HB_IsNIL(), HB_IsNull(), HB_IsNumeric(), HB_IsObject(), HB_IsPointer(), HB_IsString(), Type(), Valtype()
Category:	Debug functions, Logical functions, xHarbour extensions
Source:	rtl\valtype.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_IsNIL()

Tests if the value returned by an expression is NIL.

Syntax

```
HB_IsNIL( <expression> ) --> lIsNIL
```

Arguments

<expression>

This is an expression of any data type.

Return

The function returns .T. (true) if the value returned by *<expression>* is NIL, otherwise .F. (false) is returned.

Description

Function HB_IsNIL() is used to test if a variable contains the value NIL or if the result of an expression is of undefined data type. The function can be used as a replacement for the expression: Valtype(*<expression>*)=="U".

Info

See also:	HB_IsArray(), HB_IsBlock(), HB_IsByRef(), HB_IsDate(), HB_IsHash(), HB_IsLogical(), HB_IsMemo(), HB_IsNull(), HB_IsNumeric(), HB_IsObject(), HB_IsPointer(), HB_IsString(), Type(), Valtype()
Category:	Debug functions, Logical functions, xHarbour extensions
Source:	rtl\valtype.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_IsNull()

Tests if the value returned by an expression is empty.

Syntax

```
HB_IsNull( <expression> ) --> lIsNull
```

Arguments

<expression>

This is an expression of data type Array, Character or Hash. If a value of a different data type is passed, a runtime error is raised.

Return

The function returns .T. (true) if the value returned by *<expression>* is an empty array, character string or hash, otherwise .F. (false) is returned.

Description

Function HB_IsNull() is used to test if a variable contains an empty value of data type Array, Character or Hash. The function can be used as a replacement for the expression: Len(*<expression>*)==0.

Info

See also:	HB_IsArray(), HB_IsBlock(), HB_IsByRef(), HB_IsDate(), HB_IsHash(), HB_IsLogical(), HB_IsMemo(), HB_IsNIL(), HB_IsNumeric(), HB_IsObject(), HB_IsPointer(), HB_IsString(), Type(), Valtype()
Category:	Debug functions, Logical functions, xHarbour extensions
Source:	rtl\valtype.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_IsNumeric()

Tests if the value returned by an expression is a numeric value.

Syntax

```
HB_IsNumeric( <expression> ) --> lIsNumeric
```

Arguments

<expression>

This is an expression of any data type.

Return

The function returns .T. (true) if the value returned by *<expression>* is of data type Numeric, otherwise .F. (false) is returned.

Description

Function HB_IsNumeric() is used to test if a variable contains a numeric value or if the result of an expression is of data type Numeric. The function can be used as a replacement for the expression: Valtype(*<expression>*)=="N".

Info

See also:	HB_IsArray(), HB_IsBlock(), HB_IsByRef(), HB_IsDate(), HB_IsLogical(), HB_IsHash(), HB_IsMemo(), HB_IsNIL(), HB_IsNull(), HB_IsObject(), HB_IsPointer(), HB_IsString(), Type(), Valtype()
Category:	Debug functions, Logical functions, xHarbour extensions
Source:	rtl\valtype.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_IsObject()

Tests if the value returned by an expression is an object.

Syntax

```
HB_IsObject( <expression> ) --> lIsObject
```

Arguments

<expression>

This is an expression of any data type.

Return

The function returns .T. (true) if the value returned by *<expression>* is an object, otherwise .F. (false) is returned.

Description

Function HB_IsObject() is used to test if a variable contains an object or if the result of an expression is of data type Object. The function can be used as a replacement for the expression: Valtype(*<expression>*)=="O".

Info

See also:	HB_IsArray(), HB_IsBlock(), HB_IsByRef(), HB_IsDate(), HB_IsLogical(), HB_IsHash(), HB_IsMemo(), HB_IsNIL(), HB_IsNull(), HB_IsNumeric(), HB_IsPointer(), HB_IsString(), Type(), Valtype()
Category:	Debug functions, Logical functions, xHarbour extensions
Source:	rtl\valtype.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_IsPointer()

Tests if the value returned by an expression is a pointer.

Syntax

```
HB_IsPointer( <expression> ) --> lIsPointer
```

Arguments

<expression>

This is an expression of any data type.

Return

The function returns .T. (true) if the value returned by *<expression>* is a pointer, otherwise .F. (false) is returned.

Description

Function HB_IsPointer() is used to test if a variable contains a pointer or if the result of an expression is of data type Pointer. The function can be used as a replacement for the expression: Valtype(*<expression>*)=="P".

Info

See also: HB_IsArray(), HB_IsBlock(), HB_IsByRef(), HB_IsDate(), HB_IsLogical(), HB_IsHash(), HB_IsMemo(), HB_IsNIL(), HB_IsNull(), HB_IsNumeric(), HB_IsObject(), HB_IsString(), Type(), Valtype()
Category: Debug functions, Logical functions, xHarbour extensions
Source: rtl\valtype.c
LIB: xhb.lib
DLL: xhbdll.dll

HB_IsRegExString()

Checks if a character string is a compiled regular expression.

Syntax

```
HB_IsRegExString( <cString> ) --> lIsRegExComp
```

Arguments

`<cString>`

<cString> is a character string to test.

Return

The function returns .T. (true) if the passed string is a compiled regular expression, otherwise .F. (false) is returned.

Description

All regular expression functions and operators can process character strings containing literal regular expressions or compiled regular expressions. A literal regular expression can be compiled with HB_RegExComp() to speed up pattern matching. HB_IsRegExString() detects if a character string is a compiled regular expression.

Info

See also: HAS, HB_RegEx(), HB_RegExComp(), LIKE
Category: Regular expressions, xHarbour extensions
Source: rtl\regex.c

HB_IsString()

Tests if the value returned by an expression is a character string.

Syntax

```
HB_IsString( <expression> ) --> lIsString
```

Arguments

<expression>

This is an expression of any data type.

Return

The function returns .T. (true) if the value returned by *<expression>* is a character string, otherwise .F. (false) is returned.

Description

Function HB_IsString() is used to test if a variable contains a character string or if the result of an expression is of data type Character. The function can be used as a replacement for the expression: Valtype(*<expression>*)=="C".

Info

See also:	HB_IsArray(), HB_IsBlock(), HB_IsByRef(), HB_IsDate(), HB_IsLogical(), HB_IsHash(), HB_IsMemo(), HB_IsNIL(), HB_IsNull(), HB_IsNumeric(), HB_IsObject(), HB_IsString(), Type(), Valtype()
Category:	Debug functions, Logical functions, xHarbour extensions
Source:	rtl\valtype.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_KeyPut()

Puts an inkey code into the keyboard buffer.

Syntax

```
HB_KeyPut( <nInkeyCode> ) --> NIL
```

Arguments

<nInkeyCode>

This is a numeric Inkey() code to add to the keyboard buffer. #define constants from the file Inkey.ch can be used for <*nInkeyCode*>. The numeric value Zero is ignored.

Return

The return value is always NIL.

Description

Function HB_KeyPut() extends the KEYBOARD command and allows for adding numeric Inkey() codes to the keyboard buffer whose values are outside the range of 1 to 255.

HB_KeyPut() adds <*nInkeyCode*> to the keyboard buffer without removing pending keys from the buffer. The function can add only one Inkey() code at a time. To add multiple key codes, the function must be called repeatedly.

Info

See also:	KEYBOARD, CLEAR TYPEAHEAD, Inkey()
Category:	Keyboard functions
Header:	inkey.ch
Source:	rtl\inkey.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example outlines the difference between the KEYBOARD command
// and HB_KeyPut() with an Inkey() code greater than 255

   #include "Inkey.ch"

   PROCEDURE Main

      ? K_ALT_PGDN                // result: 417

      KEYBOARD Chr( K_ALT_PGDN )

      ? Inkey()                   // result: 161
      ? Asc( Chr( K_ALT_PGDN ) )  // result: 161

      HB_KeyPut( K_ALT_PGDN )

      ? Inkey()                   // result: 417

   RETURN
```

HB_LangErrMsg()

Returns language specific error messages.

Syntax

```
HB_LangErrMsg( <nGenericError> ) --> cErrorMessage
```

Arguments

`<nGenericError>`

This is a numeric error code identifying the error message to retrieve. #define constants from the Error.ch file can be used for this generic error code. They begin with the prefix EG_*.

Return

The function returns a charcater string describing the error with the numeric code <*nGenericError*> in textual form. If the passed parameter is no error code, a null string ("") is returned.

Description

HB_LangErrMsg() is part of xHarbour's language specific system. The returned character string depemds on the currently selected national language (see HB_LangSelect()).

The function is used in user-defined error handling routines that create Error() objects. The generic error code assigned to oError:genCode can be passed to HB_LangErrMsg() to obtain a meaningful character string for the error description in the currently selected national language. This error description can be assigned to oError:description.

Info

See also:	BEGIN SEQUENCE, Break(), Error(), HB_LangMessage(), HB_LangName(), HB_LangSelect()
Category:	Error functions, Language specific, xHarbour extensions
Source:	rtl\langapi.c, lang\msg*.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_LangMessage()

Returns language specific messages or text strings.

Syntax

```
HB_LangMessage( <nMessageID> ) --> cMessage
```

Arguments

<nMessageID>

This is a numeric code identifying the message to retrieve. The message ID is composed of a base number plus an offset. #define constants are available in the file Hblang.ch that can be used as base number for various language specific messages:

Bases for language specific messages

Constant	Value	Max. offset	Description
HB_LANG_ITEM_BASE_ID	0	6	Identification messages
HB_LANG_ITEM_BASE_MONTH	6	12	Month names
HB_LANG_ITEM_BASE_DAY	18	7	Day names
HB_LANG_ITEM_BASE_NATMSG	25	13	Clipper compatible NationMsg() messages
HB_LANG_ITEM_BASE_ERRDESC	38	52	PRG error descriptions
HB_LANG_ITEM_BASE_ERRINTR	90	26	xHarbour internal error descriptions
HB_LANG_ITEM_BASE_TEXT	116	3	Miscellaneous text strings

Return

The function returns a character string identified by its numeric identifier according to the currently selected national language.

Description

HB_LangMessage() is part of xHarbour's language specific system. The returned character string depemds on the currently selected national language (see HB_LangSelect()) and the passed numeric message ID. The message ID is composed of a base value and an offset.

Note: a programmer is rarely required to use HB_LangMessage() directly, since other functions exist that take the offset for the message ID into account. For example, the functions CMonth() or CDoW() are functions that retrieve character strings holding the name of a month or day. Also HB_LangErrMsg() accepts the #define constants found in Error.ch as message identifier and returns an appropriate error message.

HB_LangMessage() is the most generic function of xHarbour's national language support system and returns a built-in character string identified by a numeric identifier for the currently selected national language.

Info

See also: HB_LangName(), HB_LangErrMsg(), HB_LangSelect(), NationMsg()
Category: Language specific, xHarbour extensions
Header: HbLang.ch
Source: rtl\langapi.c, lang\msg*.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates how HB_LangMessage() can be used, and compares
// it to other functions embedded into the national language support. The
// example uses the English and Spanish languages.

   #include "HbLang.ch"

   REQUEST HB_LANG_ES   // request the Spanish language module

   PROCEDURE Main
      LOCAL dDate := StoD( "20070212" )

      ? HB_LangMessage( HB_LANG_ITEM_BASE_DAY + 1 )
                                       // result: Monday
      ? CDoW( dDate )                  // result: Monday

      ? HB_LangMessage( HB_LANG_ITEM_BASE_MONTH + 2 )
                                       // result: February
      ? CMonth( dDate )                // result: February

      ? HB_LangMessage( HB_LANG_ITEM_BASE_ERRDESC + 1 )
                                       // result: Argument error
      ? HB_LangErrMsg( 1 )             // result: Argument error

      ** select Spanisch as national labguage
      HB_LangSelect( "ES" )

      ? HB_LangMessage( HB_LANG_ITEM_BASE_DAY + 1 )
                                       // result: Lunes
      ? CDoW( dDate )                  // result: Lunes

      ? HB_LangMessage( HB_LANG_ITEM_BASE_MONTH + 1 )
                                       // result: Febrero
      ? CMonth( dDate )                // result: Febrero

      ? HB_LangMessage( HB_LANG_ITEM_BASE_ERRDESC + 1 )
                                       // result: Error de argumento
      ? HB_LangErrMsg( 1 )             // result: Error de argumento

   RETURN
```

HB_LangName()

Returns the currently selected language.

Syntax

```
HB_LangName() --> cLanguageID
```

Return

The function returns a 2-byte character string holding the ISO identifier for the currently selected national language. Refer to HB_LangSelect() for a table of language specific identifiers.

Info

See also:	HB_LangMessage(), HB_LangSelect(), NationMsg()
Category:	Language specific, xHarbour extensions
Source:	rtl\langapi.c, lang\msg*.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_LangSelect()

Queries or changes the current national language .

Syntax

```
HB_LangSelect( [<cLanguageID>] ) --> cOldLanguageID
```

Arguments

<cLanguageID>

This is a character string identifying the national language to select as current (see table below).

Return

The function returns a character string identifying the national language selected before the function is called.

Description

Function HB_LangSelect() queries and optionally changes the current national language used for functions returning constant character strings. Functions CDoW() or CMonth() are examples where return values differ by the national language.

By default, the English language is selected. In order to change the language, the corresponding module must be linked. That is, the symbolic names of the national languages to be available at runtime must be REQUESTed.

When the language module is linked, the language can be changed at runtime by passing the language identifier *<cLanguageID>* to HB_LangSelect(). The following table lists the REQUEST symbols and language identifiers available in xHarbour. Note that the code page must be set correctly with HB_SetCodePage() for the national languages as well.

National language support in xHarbour

Language name	Code page	REQUEST symbol	*<cLanguageID>*	Source file
Basque	850	HB_LANG_EU	"EU"	msgeu.c
Belorussian	866	HB_LANG_BY866	"BY866"	msgby866.c
Belorussian	Windows-1251	HB_LANG_BYWIN	"BYWIN"	msgbywin.c
Bulgarian	DOS-MIK	HB_LANG_BGMIK	"BGMIK"	msgbgmik.c
Bulgarian	Windows-1251	HB_LANG_BGWIN	"BGWIN"	msgbgwin.c
Catalan	850	HB_LANG_CA	"CA"	msgca.c
Chinese Simplified	936 for ZH, ZH-CN, ZH-SG	HB_LANG_ZHGB	"ZHGB"	msgzhgb.c
Chinese Traditional	950 for ZH-HK, ZH-TW	HB_LANG_ZHB5	"ZHB5"	msgzhb5.c
Croatian	1250	HB_LANG_HR1250	"HR1250"	msghr1250.c
Croatian	437	HB_LANG_HR437	"HR437"	msghr437.c
Croatian	852	HB_LANG_HR852	"HR852"	msghr852.c
Croatian	ISO-8859-2	HB_LANG_HRISO	"HRISO"	msghriso.c
Czech	852	HB_LANG_CS852	"CS852"	msgcs852.c
Czech	ISO-8859-2	HB_LANG_CSISO	"CSISO"	msgcsiso.c
Czech	Kamenickych	HB_LANG_CSKAM	"CSKAM"	msgcskam.c
Czech	1250	HB_LANG_CSWIN	"CSWIN"	msgcswin.c
Dutch	850	HB_LANG_NL	"NL"	msgnl.c
English	437	HB_LANG_EN	"EN"	msgen.c
Esperanto	850	HB_LANG_EO	"EO"	msgeo.c
French	850	HB_LANG_FR	"FR"	msgfr.c
Galician	850	HB_LANG_GL	"GL"	msggl.c

German	850	HB_LANG_DE	"DE"	msgde.c
German WIN	ANSI	HB_LANG_DEWIN	"DEWIN"	msgdewin.c
Hebrew - Dos	862	HB_LANG_HE862	"HE862"	msghe862.c
Hebrew - Windows	Windows-1255	HB_LANG_HEWIN	"HEWIN"	msghewin.c
Hungarian	852	HB_LANG_HU852	"HU852"	msghu852.c
Hungarian	CWI-2	HB_LANG_HUCWI	"HUCWI"	msghucwi.c
Hungarian	Windows-1	HB_LANG_HUWIN	"HUWIN"	msghuwin.c
Icelandic	850	HB_LANG_IS850	"IS850"	msgis850.c
Indonesian	437	HB_LANG_ID	"ID"	msgid.c
Italian	437	HB_LANG_IT	"IT"	msgit.c
Korean	949	HB_LANG_KO	"KO"	msgko.c
Lithuanian	Windows-1257	HB_LANG_LT	"LT"	msgltwin.c
Polish	852	HB_LANG_PL852	"PL852"	msgpl852.c
Polish	ISO-8859-2	HB_LANG_PLISO	"PLISO"	msgpliso.c
Polish	Mazowia	HB_LANG_PLMAZ	"PLMAZ"	msgplmaz.c
Polish	1250	HB_LANG_PLWIN	"PLWIN"	msgplwin.c
Portuguese	850	HB_LANG_PT	"PT"	msgpt.c
Romanian	852	HB_LANG_RO	"RO"	msgro.c
Russian	866	HB_LANG_RU866	"RU866"	msgru866.c
Russian	KOI-8	HB_LANG_RUKOI8	"RUKOI8"	msgrukoi.c
Russian	Windows-1251	HB_LANG_RUWIN	"RUWIN"	msgruwin.c
Serbian	Latin II 852	HB_LANG_SR852	"SR852"	msgsr852.c
Serbian	ISO-8859-2	HB_LANG_SRISO	"SRISO"	msgsriso.c
Serbian	Windows-1251	HB_LANG_SRWIN	"SRWIN"	msgsrwin.c
Slovenian	Latin II 852	HB_LANG_SL852	"SL852"	msgsl852.c
Slovenian	ISO-8859-2	HB_LANG_SLISO	"SLISO"	msgsliso.c
Slovenian	1250	HB_LANG_SLWIN	"SLWIN"	msgslwin.c
Spanish	850	HB_LANG_ES	"ES"	msges.c
Spanish WIN	ANSI	HB_LANG_ESWIN	"ESWIN"	msgeswin.c
Ukrainian	866	HB_LANG_UA866	"UA866"	msgua866.c
Ukrainian	KOI-8U	HB_LANG_UAKOI8	"UAKOI8"	msguakoi.c
Ukrainian	Windows-1251	HB_LANG_UAWIN	"UAWIN"	msguawin.c

Info

See also: HB_AnsiToOem(), HB_LangName(), HB_OemToAnsi(), HB_SetCodePage()
Category: Language specific, xHarbour extensions
Source: rtl\langapi.c, lang\msg*.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example shows the result of CDoW() depending
// on the selected national language

   REQUEST HB_LANG_ES              // Spanish language
   REQUEST HB_LANG_IT              // Italian language

   PROCEDURE Main

      ? HB_LangSelect()            // result: EN
      ? Cdow( Date() )             // result: Friday

      ? HB_LangSelect( "ES" )      // result: EN
      ? HB_LangSelect()            // result: ES
      ? CdoW( Date() )             // result: Viernes
```

```
   ? HB_LangSelect( "IT" )      // result: ES
   ? HB_LangSelect()            // result: IT
   ? CdoW( Date() )             // result: Venerdì

RETURN
```

HB_LibDo()

Executes a function located in a dynamically loaded xHarbour DLL.

Syntax

```
HB_LibDo( <cFuncName> [, <xParams,...>] ) --> xResult
```

Arguments

<cFuncName>

This is a character string holding the symbolic name of the function or procedure to execute.

<xParams,...>

The values of all following parameters specified in a comma separated list are passed on to the DLL function.

Return

The function returns the result of the called DLL function.

Description

Function HB_LibDo() is used to execute functions located in DLL files created by the xHarbour compiler and linker. This applies to DLL files loaded dynamically at runtime with function LibLoad(). If a DLL file is bound statically at link time, a DLL function can be called directly.

HB_LibDo() looks up the symbolic function name *<cFuncName>* and passes the values of *<xParams,...>* to it. The return value of HB_LibDo() is the result of *<cFundName>*. If *<cFuncNmae>* cannot be found, a runtime error is generated.

Important: the EXE file loading the xHarbour DLL with LibLoad() must be created for DLL usage for HB_LibDo() to work properly (compiler switch -usedll, linker switch /DLL).

Info

See also:	DllCall(), LibLoad(), LoadLibrary()
Category:	DLL functions, xHarbour extensions
Source:	vm\dynlibhb.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example consists of two file. The first is compiled to an EXE
// and the second to a DLL. Function SayHello() resides in the DLL
// and is called from the EXE.

   /* ----------------------------------------------------------
   // File: DYNDLL.PRG
   //       Compile and link as EXE for DLL usage
   //       e.g. XBuild -usedll dyndll.exe dyndll.prg
   // --------------------------------------------------------- */
   PROCEDURE Main
      LOCAL pDLL

      ? Type( "SayHello()" )
        // result is "U" because the function SayHello()
        // does not exist
```

```
   ? pDLL := LibLoad( "dyndll1.dll" )
     // result is a pointer to the loaded DLL


   ? Type( "SayHello()" )
     // result is "UI" because function SayHello()
     // exists now (in loaded DLL)


   ? HB_LibDo( "SayHello" )
     // result is:  1
     // displays : Hello from DLL


   ? HB_LibDo( "SayHello", "Called via HB_LibDo()" )
     // result is:  2
     // displays : Called via HB_LibDo()


   ? &("SayHello")( "Called via Macro operator" )
     // result is:  3
     // displays : Called via Macro operator
     //
     // NOTE:
     // This is the fastest way of calling a DLL function:
     //
     // The macro operator "&" looks up the function symbol.
     // The execution operator "()" calls the function and
     // passes parameter(s) on to it


   ? LibFree( pDll )
     // result is .T., DLL is successfully unloaded


   ? Type( "SayHello()" )
     // result is "U" because the function SayHello()
     // does not exist anymore

RETURN


/* ------------------------------------------------------------
// File: DYNDLL1.PRG
//       Compile and link as DLL
//       e.g. XBuild dyndll1.dll dyndll.prg
// ---------------------------------------------------------- */
FUNCTION SayHello( cText )
   STATIC nCalls := 0

   IF cText == NIL
      cText := "Hello from DLL"
   ENDIF

   ? cText

RETURN ++nCalls
```

HB_LogDateStamp()

Returns a character string holding a date stamp.

Syntax

```
HB_LogDateStamp() --> cDateStamp
```

Return

The function returns a character string holding a date stamp in the format YYYY-MM-DD.

Description

HB_LogDateStamp() is a utility function for xHarbour's Log system to create an ASCII sortable date stamp.

Info

See also:	INIT LOG, TraceLog(), HB_BldLogMsg()
Category:	Debug functions, Log functions, xHarbour extensions
Source:	rtl\hblog.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_MacroCompile()

Compiles a macro string into a PCode sequence.

Syntax

```
HB_MacroCompile( <cMacro> ) --> cPCode
```

Arguments

<cMacro>

This is a character string holding a macro expression to compile.

Return

The function returns a characters string holding the PCode sequences of the compiled macro expression.

Description

Function HB_MacroCompile() accepts a character string and compiles it into a PCode sequence. This, again, is a character string holding executable code, which is executed by passing it to function HB_VMExecute().

If *<cMacro>* is not a valid macro expression, a runtime error is raised.

Info

See also: & (macro operator), Eval(), HB_VMExecute()
Category: Indirect execution, xHarbour extensions
Source: vm\macro.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example outlines two possibilities of calling a function
// indirectly via macro expression. The first compiles a macro
// to PCode, and the second creates a code block with the macro
// operator. Note that functions that are only included in macro
// expressions should be REQUESTed to make sure they are linked.

   REQUEST QOut, Time

   PROCEDURE Main
      LOCAL cMacro := "QOut( Time() )"
      LOCAL cPCode := HB_MacroCompile( cMacro )
      LOCAL bBlock := &( "{||" + cMacro + "}" )

      HB_VMExecute( cPCode )

      EVal( bBlock )
   RETURN
```

HB_MD5()

Calculates a message digest for a stream of data using the MD5 algorithm.

Syntax

```
HB_MD5( <cString> ) --> cMD5
```

Arguments

<cString>

This is a character string to calulate the message disgest for.

Return

The function returns a character string of 32 characters which is the message digest of <*cString*>.

Description

HB_MD5() is a cryptographic hash function wich accepts a variable length character string (or message) as input, and produces a fixed length string as output, also known as Message Digest or Digital Fingerprint. The algorithm used is the MD5 algorithm.

Info

See also: HB_CheckSum(), HB_CRC32(), HB_MD5File()
Category: Checksum functions, xHarbour extensions
Source: rtl\hbmd5.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays the message digest of severeal character
// strings, including null string.

   PROCEDURE Main
      ? HB_MD5("")      // result: d41d8cd98f00b204e9800998ecf8427e

      ? HB_MD5("a")     // result: 0cc175b9c0f1b6a831c399e269772661

      ? HB_MD5("abc")   // result: 900150983cd24fb0d6963f7d28e17f72

      ? HB_MD5("message digest")
                        // result: f96b697d7cb7938d525a2f31aaf161d0
   RETURN
```

HB_MD5File()

Calculates a message digest for a file using the MD5 algorithm.

Syntax

```
HB_MD5File( <cFileName> ) --> cMD5
```

Arguments

<cFileName>

This is a character string holding the name of the file to process. It must include path and file extension. If the path is omitted from *<cFileName>*, the file is searched in the current directory.

Return

The function returns a character string of 32 characters which is the message digest of the file specified.

Description

HB_MD5File() does the same as HB_MD5() but processes an entire file rather than a character string. The result is the message digest of the file *<cFileName>* which can be used to detect if the file is changed.

Info

See also:	HB_CheckSum(), HB_CRC32(), HB_MD5()
Category:	Checksum functions, xHarbour extensions
Source:	rtl\hbmd5.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example implements a simple command line tool that
// displays the MD5 message digest of a file.

   PROCEDURE Main( cFileName )
      IF PCount() > 0 .AND. File( cFileName )
         ? HB_MD5File( cFileName )
      ELSE
         ? "Usage: MD5File <filename>"
      ENDIF
   RETURN
```

HB_MultiThread()

Checks if an application is created for multi-threading.

Syntax

```
HB_MultiThread() --> lIsMultiThread
```

Return

The functions returns .T. when the application supports multiple threads, otherwise .F. (false) is returned.

Description

The function is used to check if an application is created with the multi-thread enabled libraries of xHarbour. This is accomplished with the -mt compiler switch.

Note: the multi-threading capabilities require additional code to be linked and create a larger executable than a single-threaded application.

Info

See also:	StartThread()
Category:	Multi-threading functions, xHarbour extensions
Source:	rtl\hvm.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_MutexCreate()

Creates a Mutex.

Syntax

```
HB_MutexCreate() --> pMutexHandle
```

Return

The function returns a handle to the created Mutex.

Description

Function HB_MutexCreate() creates a Mutex and returns a handle to it. A Mutex (the name comes from MUTual EXclusion) is a complex data structure required in multi-threaded programming to protect memory resources from concurrent access by multiple threads. There are two distinct programming techniques applied in Mutex usage: Locks and Notifications. No matter which technique is used, a Mutex is only required when at least two threads access the same memory resources or execute the same program code simultaneously. If this is not the case, a Mutex is not required.

Mutex lock

A Mutex can be locked with HB_MutexLock(). The first thread calling this function obtains a lock on the Mutex and can proceed with program execution. Any subsequent thread calling HB_MutexLock() with the same Mutex is suspended by the operating system until the Mutex lock is freed by the locking thread with a call to HB_MutexUnlock().

When the locking thread calls HB_MutexLock() again with the same Mutex, the lock counter is increased and the thread proceeds. HB_MutexUnlock() must then be called as many times as HB_MutexLock() to reset the lock counter to zero for releasing the Mutex lock.

Mutex locking/unlocking is always done in the same thread. A thread holding a lock prevents other threads from executing the same, Mutex protected, program code simultaneously.

Mutex notification

Mutex notifications provide some sort of communication protocol between two threads. This is accomplished by the functions Notify() and Subscribe(). Both functions must operate on the same Mutex but are called in two different threads. If, for example, thread "A" subscribes to a Mutex, it is suspended by the operating system until a second thread "B" notifies the Mutex. When thread "B" calls Notify(), the suspended thread "A" "wakes up" and resumes program execution.

The Subscribe()/Notify() protocol allows for controlling the execution of a thread "A" by a second thread "B".

Info

See also:	GetCurrentThread(), GetThreadID(), HB_MutexCreate(), HB_MutexLock(), Notify(), StartThread(), Subscribe()
Category:	Multi-threading functions, Mutex functions, xHarbour extensions
Source:	vm\thread.c
LIB:	xhbmt.lib
DLL:	xhbmtdll.dll

Examples

Mutex locking

```
// The example demonstrates how a routine that is safe for
// single-threaded applications can be made safe for
```

```
// a multi-threaded application by means of a Mutex. The two
// routines ShowTIDs_ST() and ShowTIDs_MT() differ only in
// Mutex usage. The output of ShowTIDs_ST() is totally
// scrambled and differs from one program invocation to the
// next when running simultaneously in ten threads. The
// Mutex protected routine ShowTIDs_MT(), in contrast,
// produces always the same, proper output when executed
// by multiple threads.

   GLOBAL pMutex

   PROCEDURE Main
      LOCAL i

      CLS
      ?
      ? "--- Unsafe operation ----"
      FOR i:=1 TO 10
         StartThread( "ShowTIDs_ST" )
      NEXT
      WaitForThreads()

      pMutex := HB_MutexCreate()

      ?
      ? "---- Safe operation -----"
      FOR i:=1 TO 10
         StartThread( "ShowTIDs_MT" )
      NEXT
      WaitForThreads()

   RETURN


   PROCEDURE ShowTIDs_ST()
      LOCAL cTID

      ?  "APP TID:"  , Str( GetThreadID(), 5 )
      ?? "  SYS TID:", Str( GetSystemThreadID(), 5 )
   RETURN


   PROCEDURE ShowTIDs_MT()
      LOCAL cTID

      HB_MutexLock( pMutex )

      ?  "APP TID:"  , Str( GetThreadID(), 5 )
      ?? "  SYS TID:", Str( GetSystemThreadID(), 5 )

      HB_MutexUnlock( pMutex )
   RETURN
```

Mutex notification

```
// The example demonstrates the technique of Mutex notification.
// Ten threads are started in the FOR...NEXT loop and immediately
// suspended in procedure GetTIDs by subscribing to the Mutex. All
// threads are then notified with NotifyAll() and collect their
// thread IDs in an array which is finally displayed. The result
// of the example is not predictable since the operating system
// schedules the threads for execution on its own behalf.
```

```
// I.e. the example produces different output each time it is
// started.
//
// Note that the Mutex is held in a GLOBAL variable to be visible
// when child threads Subscribe() to the Mutex.

   GLOBAL pMutex

   PROCEDURE Main
      LOCAL i, aTID := {}, pThread, aThread := {}

      pMutex := HB_MutexCreate()

      FOR i:=1 TO 10
         pThread := StartThread( "GetTIDs", aTID )
         AAdd( aThread, pThread )
      NEXT

      NotifyAll( pMutex )
      ThreadSleep( 100 )

      AEval( aTID, {|c| QOut(c) } )

   RETURN


   PROCEDURE GetTIDs( aTID )
      LOCAL cTID
      // A thread waits infinitely until notification
      Subscribe( pMutex )

      cTID :=   "APP TID:" + Str( GetThreadID(), 5 )
      cTID += "  SYS TID:" + Str( GetSystemThreadID(), 5 )

      AAdd( aTID, cTID )
   RETURN
```

HB_MutexLock()

Obtains a permanent lock on a Mutex.

Syntax

```
HB_MutexLock( <pMutexHandle> ) --> lSuccess
```

Arguments

`<pMutexHandle>`

This is the handle of the Mutex to lock. It is the return value of HB_MutexCreate().

Return

The function returns .T. (true) when the Mutex is locked, and .F. (false) when an error occurs.

Description

Function HB_MutexLock() locks the Mutex *<pMutexHandle>* for the current thread. If the Mutex is already locked by another thread, the current thread is suspended. That is, HB_MutexLock() waits infinitely and returns only when a lock is obtained.

If the infinite wait period is not desired, a thread can call HB_MutexTryLock() to check if the Mutex can be locked, or HB_MutexTimeoutLock(), to restrict the wait period to a specified amount of time.

A locked Mutex must be unlocked with HB_MutexUnlock() to grant other threads access to the Mutex protected program code.

Refer to function HB_MutexCreate() for more information on Mutexes.

Info

See also:	GetCurrentThread(), GetThreadID(), HB_MutexCreate(), HB_MutexTimeoutLock(), HB_MutexTryLock(), HB_MutexUnlock(), JoinThread(), Notify(), StartThread(), Subscribe()
Category:	Multi-threading functions, Mutex functions, xHarbour extensions
Source:	vm\thread.c
LIB:	xhbmt.lib
DLL:	xhbmtdll.dll

HB_MutexTimeoutLock()

Tries to obtain a lock on a Mutex with timeout.

Syntax

```
HB_MutexTimeoutLock( <pMutexHandle> , ;
                     <nWaitMilliSecs> ) --> lSuccess
```

Arguments

`<pMutexHandle>`

This is the handle of the Mutex to lock. It is the return value of HB_MutexCreate().

`<nWaitMilliSecs>`

A numeric value specifying the maximum period of time to wait before the function returns when a lock cannot be obtained. The unit for this timeout period is 1/1000th of a second. If *<nWaitMilliSecs>* is omitted, the thread waits infinitely, i.e. the function does not return until a lock is obtained.

Return

The function returns .T. (true) when the Mutex is locked within the timeout period, and .F. (false) when no lock is obtained.

Description

The function works like HB_MutexLock() but accepts as second parameter a timeout value. The function returns when either a lock is obtained on the Mutex, or the timeout period has expired. The return value indicates if the current thread has obtained the lock.

Info

See also:	GetCurrentThread(), GetThreadID(), HB_MutexCreate(), HB_MutexTimeoutLock(), HB_MutexTryLock(), HB_MutexUnlock(), Notify(), StartThread(), Subscribe()
Category:	Multi-threading functions, Mutex functions, xHarbour extensions
Source:	vm\thread.c
LIB:	xhbmt.lib
DLL:	xhbmtdll.dll

HB_MutexTryLock()

Tries to obtain a permanent lock on a Mutex.

Syntax

```
HB_MutexTryLock( <pMutexHandle> ) --> lSuccess
```

Arguments

<pMutexHandle>

This is the handle of the Mutex to lock. It is the return value of HB_MutexCreate().

Return

The function returns .T. (true) when the Mutex is locked, otherwise .F. (false).

Description

The function works like HB_MutexLock() but does not wait when the Mutex is currently locked. Instead, it returns .F. (false) so that the current thread continues to run. If .T. (true) is returned, the current thread has obtained a Mutex lock, which must be unlocked later with HB_MutexUnlock().

Info

See also:	GetCurrentThread(), GetThreadID(), HB_MutexCreate(), HB_MutexLock(), HB_MutexTimeoutLock(), HB_MutexUnlock(), Notify(), StartThread(), Subscribe()
Category:	Multi-threading functions, Mutex functions, xHarbour extensions
Source:	vm\thread.c
LIB:	xhbmt.lib
DLL:	xhbmtdll.dll

HB_MutexUnlock()

Unlocks a Mutex.

Syntax

```
HB_MutexUnlock( <pMutexHandle> ) --> NIL
```

Arguments

<pMutexHandle>

This is the handle of the Mutex to unlock. It is the return value of HB_MutexCreate().

Return

The function returns always NIL.

Description

Function HB_MutexUnlock() releases a lock previously set with HB_MutexLock(), HB_MutexTimeoutLock() or HB_MutexTryLock(). A Mutex must be unlocked in the same thread that has obtained a lock.

Refer to HB_MutexCreate() for more information on Mutexes.

Info

See also:	GetCurrentThread(), GetThreadID(), HB_MutexCreate(), HB_MutexLock(), HB_MutexTimeoutLock(), HB_MutexTryLock(), Notify(), StartThread(), Subscribe()
Category:	Multi-threading functions, Mutex functions, xHarbour extensions
Source:	vm\thread.c
LIB:	xhbmt.lib
DLL:	xhbmtdll.dll

HB_ObjMsgPtr()

Retrieves the pointer to a method.

Syntax

```
HB_ObjMsgPtr( <oObject>, <cMethod> ) --> nMethodPointer
```

Arguments

<oObject>

This is an object whose method pointer is retrieved.

<cMessage>

This is a character string holding the symbolic name of the method to retrieve the pointer for.

Return

The function returns a pointer to the method *<cMethod>* of the object *<oObject>*, or NIL, if the object does not have the specified method.

Description

Function HB_ObjMsgPtr() retrieves the pointer to a method of an object. A pointer is the memory address of a method. It can be used with HB_Exec() or HB_ExecFromArray() to execute a method from its pointer. This is similar to embedding a method call within a code block and passing the block to the Eval() function. A code block can contain multiple method calls. If only one method is to be executed indirectly, a method pointer along with HB_Exec() or HB_ExecFromArray() is faster than code block execution.

The advantage of a method pointer is that the dynamic lookup of the symbolic method name can be avoided once the pointer is obtained. This can improve the performance of time critical methods considerably when they must be called very often.

Info

See also:	@(), CLASS, HB_Exec(), HB_ExecFromArray(), HB_FuncPtr(), METHOD (Declaration)
Category:	Indirect execution, xHarbour extensions
Source:	vm\classes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example compares indirect method execution using
// a method pointer and a code block.

   #include "hbclass.ch"

   PROCEDURE Main
      LOCAL oObject  := Test():new( "xHarbour" )
      LOCAL pPointer := HB_ObjMsgPtr( oObject, "display" )
      LOCAL bBlock   := {|o| o:display() }

      HB_Exec( pPointer, oObject )

      Eval( bBlock, oObject )
   RETURN
```

```
CLASS Test
  PROTECTED:
  DATA name

  EXPORTED:
  METHOD init CONSTRUCTOR
  METHOD display
ENDCLASS

METHOD init( cName ) CLASS Test
   ::name := cName
RETURN self

METHOD display CLASS Test
   QOut( ::name )
RETURN self
```

HB_OemToAnsi()

Converts a character string from the OEM to the ANSI character set.

Syntax

```
HB_OemToAnsi( <cOEM_String> ) --> cANSI_String
```

Arguments

<cOEM_String>

A character string holding characters from the OEM character set.

Return

The function returns a string converted to the ANSI character set.

Description

The function converts all characters of *<cOEM_String>* to the corresponding characters in the Windows ANSI character set. Both character sets differ in characters whose ASCII values are larger than 127.

If a character in the OEM string does not have an equivalent in the ANSI character set, it is converted to a similar character of the ANSI character set. In this case, it may not be possible to reverse the OEM => ANSI conversion with function HB_AnsiToOem().

Note: all databases created with Clipper store their data using the OEM character set. Displaying data in xHarbour console applications occurs with the OEM character set. Data display in xHarbour GUI applications is done with the Windows ANSI character set.

Info

See also:	HB_AnsiToOem(), HB_SetCodePage()
Category:	Conversion functions, xHarbour extensions
Source:	rtl\oemansi.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// See the example for function HB_AnsiToOem()
```

HB_OpenProcess()

Opens a child process.

Syntax

```
HB_OpenProcess( <xExeFile>    , ;
               [@<nStdIN>]    , ;
               [@<nStdOUT>]   , ;
               [@<nStdERR>]   , ;
               [<lBackGround>], ;
               [@<nProcessID>   ) --> nProcessHandle
```

Arguments

<xExeFile>

This is a character string holding the name of the executable file to run in the child process. It must be specified as a full qualified file name, including path and file extension. When the path is omitted, the file is searched in the current directory.

@<nStdIN>, @<nStdOUT>, @<nStdERR>

Three optional parameters can be passed by reference. They receive standard file handles for STDIN, STDOUT and STDERR. These file handles can be used to exchange data with the child process.

<lBackGround>

This parameter defaults to .F. (false). When set to .T. (true), the child process is started asynchronously as a background process.

<nProcessID>

If a sixth parameter is passed by reference, it receives the numeric identifier of the child process as assigned by the operating system. It is of informational nature.

Return

The function returns the process handle (similar to a file handle) of the new child process as a numeric value, or -1 if an error occurs. Use function FError() to identify the cause of failure.

Description

Function HB_OpenProcess() opens a new child process and executes the program *<cExeFile>* in the new process. The program is executed synchronously (*<lBackGround>*==.T.) or asynchronously (*<lBackGround>*==.F.).

When the parent process needs to exchange data with the child process, it can use the standard handlers STDIN, STDOUT and STDERR. Numeric handles for them are assigned to the reference parameters *<nStdIN>*, *<nStdOUT>* and *<nStdERR>*.

The parent process can wait for the end of a synchronously started child process with function HB_ProcessValue(), or it can terminate the child process explicitely with function HB_CloseProcess().

When the child process is complete, its process handle and the standard handles, if obtained, must be closed with FCLose().

Info

See also: HB_CloseProcess(), HB_ProcessValue(), RUN, StartThread()
Category: Process functions, xHarbour extensions
Source: rtl\philes.c
LIB: xhb.lib
DLL: xhbdll.dll

Examples

Parent process

```
// This example implements a parent process launching a child process
// (it is shown in the next example).

   PROCEDURE Main()
      LOCAL cData  := "Hello World"
      LOCAL cEXE   := "ChildProc.exe"
      LOCAL nBytes, nChild, nError, nStdIN, nStdOUT, nStdERR

      CLS
      ? "Opening child process:", cEXE

      nChild := HB_OpenProcess( cEXE, @nStdIN, @nStdOUT, @nStdERR )

      IF nChild < 0
         ? "Error:", FError()
         QUIT
      ENDIF

      ? "Sending data         :", FWrite( nStdIN, cData )

      ? "Receiving data       : "
      cData := Space( 1000 )
      nBytes := Fread( nStdOUT, @cData, Len(cData) )
      ?? Left( cData, nBytes )

      ? "Reading errors       : "
      cData := Space( 1000 )
      nBytes := Fread( nStdERR, @cData, Len(cData) )
      ?? Left( cData, nBytes )

      ? "Waiting for end      :", HB_ProcessValue( nChild )

      FClose( nChild )
      FClose( nStdIN )
      FClose( nStdOUT )
      FClose( nStdERR )
   RETURN
```

Child process

```
// This example is the child process lauched by the first exemple.
// Both must be compiled and linked as separate EXE files.

   // definition of standard file handles available
   // in a child process:

   #define  FH_STDIN   0
   #define  FH_STDOUT  1
   #define  FH_STDERR  2
```

```
#include "Fileio.ch"

#define BUFF_SIZE   1024

PROCEDURE Main
   LOCAL nBytes, cBuffer, cData, nError

   cBuffer := Space( BUFF_SIZE )

   nBytes  := FRead( FH_STDIN, @cBuffer, Len( cBuffer ) )

   cBuffer := "Child process received "    + ;
                     LTrim( Str( nBytes) ) + ;
              " bytes [" + Trim( cBuffer ) + "]"

   nBytes  := FWrite( FH_STDOUT, cBuffer, Len(cBuffer) )

   nError  := FError()

   cBuffer := "Error code: " + LTrim( Str( nError ) )

   nBytes  := FWrite( FH_STDERR, cBuffer, Len(cBuffer) )
RETURN
```

HB_OsDriveSeparator()

Returns the operating specific drive separator character.

Syntax

```
HB_OsDriveSeparator() --> cDriveSeparator
```

Return

The function returns a character used as drive separator.

Info

See also:	HB_OsNewLine(), HB_OsPathDelimiters(), HB_OsPathListSeparator(), HB_OsPathSeparator()
Category:	Disks and Drives, Environment functions, xHarbour extensions
Source:	rtl\philes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows an operating system independent way of
// obtaining a fully qualified file name from a simple file name.

   PROCEDURE Main
      LOCAL cFile := "Test.prg"

      ? FullFileName( cFile )
   RETURN

   FUNCTION FullFileName( cFileName )
      LOCAL cPath := CurDrive()
      LOCAL cDelim:= IIF( "\" $ HB_OsPathDelimiters(), "\", "/" )

      cPath += HB_OsDriveSeparator()
      cPath += cDelim + CurDir() + cDelim
   RETURN  cPath + cFileName
```

HB_OsError()

Returns the operating specific error code of the last low-level file operation.

Syntax

```
HB_Oserror() --> nLastErrorCode
```

Return

The function returns a numeric value representing the operating system specific error code of the last low-level file operation.

Description

HB_OsError() is used in the same ways as FError(). The difference is that FError() returns Clipper compatible error codes, while HB_OsError() returns the error code that was actually issued by the operating system. The error codes of both functions may or may not differ for the same error condition.

Info

See also:	DosError(), FError()
Category:	Error functions, Low level file functions, xHarbour extensions
Source:	rtl\philes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_OsNewLine()

Returns the end-of-line character(s) to use with the current operating system.

Syntax

```
HB_OsNewLine() --> cEndOfLine
```

Return

The function returns a character string containing the end-of-line character(s).

Description

HB_OsNewLine() returns a character string containing the character(s) required to move the screen cursor or print head to the start of a new line. The string will hold either Chr(10) or Chr(13)+Chr(10).

Info

See also:	Chr(), Os(), OutStd(), OutErr(), SET EOL
Category:	Environment functions, xHarbour extensions
Source:	rtl\console.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_OsPathDelimiters()

Returns the operating specific characters for paths.

Syntax

```
HB_OsPathDelimiters() --> cDelimiters
```

Return

The function returns a character string holding all delimiting characters that may occur in a full qualified path name, including separators for the drive and directories.

Info

See also: HB_OsNewLine(), HB_OsDriveSeparator(), HB_OsPathListSeparator(), HB_OsPathSeparator()
Category: Environment functions, xHarbour extensions
Source: rtl\philes.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example implements a function that splits a full qualified
// file name into all components.

   PROCEDURE Main
      LOCAL cFile := "c:\xhb\samples\Test.prg"
      LOCAL aFile

      aFile := SplitFileName( cFile )
      AEval( aFile, {|c| QOut( c ) } )

      ** output
      // c
      // xhb
      // samples
      // test.prg
   RETURN

   FUNCTION SplitFileName( cFileName )
      LOCAL cDelims:= HB_OSPathDelimiters()
      LOCAL aPos   := {0}, nStart, nEnd
      LOCAL aFile  := {}
      LOCAL cDelim, i, imax, cStr

      FOR EACH cDelim IN cDelims
         nStart := 1
         DO WHILE ( nEnd := At( cDelim, cFileName, nStart ) ) > 0
            AAdd( aPos, nEnd )
            nStart := nEnd + 1
         ENDDO
      NEXT

      imax  := Len( aPos )
      AAdd( aPos, Len( cFileName ) + 1 )
      ASort( aPos )

      FOR i:=1 TO imax
         nStart   := aPos[i] + 1
```

```
      nEnd     := aPos[i+1]
      cStr     := SubStr( cFileName, nStart, nEnd-nStart )
      IF .NOT. cStr == ""
         AAdd( aFile, cStr )
      ENDIF
   NEXT
RETURN aFile
```

HB_OsPathListSeparator()

Returns the operating specific separator character for a path list.

Syntax

```
HB_OsPathListSeparator() --> cDDelimiter
```

Return

The function returns a character used as separator between directories in a path list.

Info

See also:	HB_OsNewLine(), HB_OsDriveSeparator(), HB_OsPathDelimiters(), HB_OsPathSeparator()
Category:	Environment functions, xHarbour extensions
Source:	rtl\philes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example implements a function that fills an array with all
// directories contained in an environment variable holding a path.

   PROCEDURE Main
      LOCAL aPath := AGetPath( "PATH" )

      ? "SET PATH="
      AEval( aPath, {|c| QOut( c ) } )

      ? "SET INCLUDE="
      aPath := AGetPath( "INCLUDE" )
      AEval( aPath, {|c| QOut( c ) } )
   RETURN


   FUNCTION AGetPath( cEnvVar )
      LOCAL cDelim := HB_OsPathListSeparator()
      LOCAL cPath  := GetEnv( cEnvVar )
      LOCAL aPath  := {}
      LOCAL nStart := 1
      LOCAL nEnd

      DO WHILE ( nEnd := At( cDelim, cPath, nStart ) ) > 0
         AAdd( aPath, SubStr( cPath, nStart, nEnd - nStart ) )
         nStart := nEnd + 1
      ENDDO
   RETURN aPath
```

HB_OsPathSeparator()

Returns the operating specific separator character for a path.

Syntax

```
HB_OsPathSeparator() --> cDelimiter
```

Return

The function returns a character used as separator between subdirectories in a full qualified file name.

Info

See also: HB_OsNewLine(), HB_OsDriveSeparator(), HB_OsPathDelimiters(), HB_OsPathListSeparator()
Category: Environment functions, xHarbour extensions
Source: rtl\philes.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example implements a function filling an array with the
// information of a full qualified file name. The first element holds
// the drive and root directory while the last element contains the file
// name. All elements in between hold names of subdirectories

   PROCEDURE Main
      LOCAL cFile := "c:\xhb\source\rtl\philes.c"
      LOCAL aFile := AGetFile( cFile )

      AEval( aFile, {|c| QOut(c) } )
      ** output
      // c:
      // xhb
      // source
      // rtl
      // philes.c
   RETURN

   FUNCTION AGetFile( cFileName )
      LOCAL cDelim := HB_OsPathSeparator()
      LOCAL aFile  := {}
      LOCAL nStart := 1
      LOCAL nEnd

      DO WHILE ( nEnd := At( cDelim, cFileName, nStart ) ) > 0
         AAdd( aFile, SubStr( cFileName, nStart, nEnd - nStart ) )
         nStart := nEnd + 1
      ENDDO
   RETURN aFile
```

HB_PCodeVer()

Retrieves the PCode version of the current xHarbour build.

Syntax

```
HB_PCodeVer() --> cPCodeVersion
```

Return

The function returns the PCode version of the current xHarbour build as a character string.

Info

See also:	HB_BuildDate(), HB_BuildInfo(), HB_Compiler(), Os(), Os_VersionInfo()
Category:	Environment functions, xHarbour extensions
Source:	rtl\version.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_Pointer2String()

Reads bytes from a pointer into a character string.

Syntax

```
HB_Pointer2String( <pPointer>, <nBytes> ) --> cString
```

Arguments

<pPointer>

This is a pointer value previously obtained from HB_String2Pointer().

<nBytes>

This is a numeric value specifying the number of bytes to read from the pointer.

Return

The function returns the a character string of *<nBytes>* length holding the bytes currently stored at the memory address *<pPointer>*.

Description

HB_Pointer2String() is the reverse function of HB_String2Pointer(). It reads *<nBytes>* bytes from memory and copies them into a character string, which is returned.

Warning: be cautious with reading bytes from a pointer. If the character string stored at the memory address *<pPointer>* has less than *<nBytes>* bytes, the result is unpredictable.

Info

See also:	C Structure class, HB_String2Pointer()
Category:	Pointer functions, xHarbour extensions
Source:	rtl\str2ptr.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example converts a string to a pointer and reads
// various numbers of bytes from memory.
// The last line is an error to demonstrate the
// unpredictable result when using HB_Pointer2String()
// incorrectly.

   PROCEDURE Main
      LOCAL cString := "123456789"
      LOCAL pString := HB_String2Pointer( cString )

      ? Valtype( cString ), cString
      ? Valtype( pString ), pString

      ? HB_Pointer2String( pString, 5 )

      ? HB_Pointer2String( pString, 9 )

      // This is an error!!
      ? HB_Pointer2String( pString, 15 )
   RETURN
```

HB_ProcessValue()

Retrieves the return value of a child process.

Syntax

```
HB_ProcessValue( <nProcessHandle>, ;
                 [<lWaitForTermination>] ) ) --> nErrorLevel
```

Arguments

<nProcessHandle>

This is the numeric process handle of the child process to retrieve the return code of. It is returned from HB_OpenProcess().

<lWaitForTermination>

This parameter defaults to .T. (true) so that the parent process waits until the child process has regularly ended with program execution. When .F. (false) is passed, the function returns immediately if the child process is still executing its program.

Return

The function obtaines the return code of the program running in the child process. This return code is set via ErrorLevel() when the child process runs an xHarbour application. If the child process has not finished program execution when HB_ProcessValue() returns, the return value is -1.

Description

HB_ProcessValue() can be used for two purposes: it can check if a child process started with HB_OpenProcess() is still executing its program, or it can halt the parent process until the child process is complete with program execution.

To "kill" a program running in a child process, call HB_CloseProcess() and pass .F. (false) as second paremeter.

Info

See also:	HB_CloseProcess(), HB_OpenProcess()
Category:	Process functions, xHarbour extensions
Source:	rtl\philes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_QSelf()

Retrieves the *self* object during method execution.

Syntax

```
HB_QSelf() --> self
```

Return

The function returns the *self* object while methods are being executed. Outside the context of methods, the return value is NIL.

Description

Function HB_QSelf() returns the *self* object in the context of method execution. This is only rquired when methods are implemented with a FUNCTION or PROCEDURE statement rather than a METHOD statement.

Functions or procedures can serve as method implementation when a method of a class is redefined using EXTEND CLASS...WITH METHOD.

Note: when HB_QSelf() is called outside the body of a method implementation, the return value is always NIL. Code blocks cannot retrieve the *self* object with HB_QSelf(). To access the *self* object within a code block, *self* must either be embedded in the code block, or it must be passed as a code block parameter.

Info

See also:	CLASS, EXTEND CLASS...WITH METHOD, HB_QWith()
Category:	Object functions, xHarbour extensions
Source:	vm\hvm.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_QWith()

Retrieves the *with* object during WITH OBJECT execution.

Syntax

```
HB_QWith() --> oObject | NIL
```

Return

The function returns the *with* object while the WITH OBJECT statement is executed. Outside this statement, the return value is NIL.

Description

Function HB_QWith() returns the *with* object in the context of the WITH OBJECT statement. All abbreviated messages within the WITH OBJECT .. END block are addressed to the *with* object.

Info

See also:	HB_QSelf(), HB_SetWith(), WITH OBJECT
Category:	Object functions, xHarbour extensions
Source:	vm\hvm.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_Random()

Generates a random number between two boundaries.

Syntax

```
HB_Random( [<nMaxLimit>] ) --> nRandomNumber
or
HB_Random( <nMinLimit>, <nMaxLimit> ) --> nRandomNumber
```

Arguments

`<nMaxLimit>`

<nMaxLimit> is a numeric value specifying the upper boundary of the random number generator. It defaults to 1.

`<nMinLimit>`

<nMinLimit> is a numeric value specifying the lower boundary of the random number generator. It defaults to 0. Note that the first parameter is only interpreted as lower boundary when two parameters are passed.

Return

The function returns a random numeric value with two decimal digits, limited by the two boundaries.

Description

This function generates a random number between two boundaries. The boundary values are **excluded** from the range. When only one boundary is passed to the function, it is considered the maximum boundary. The minimum boundary, in that case, is zero. When no boundaries are passed, the function returns a number between 0.00 and 1.00.

Note Use function HB_RandomInt() to generate random integer values.

Info

See also:	HB_RandomInt(), HB_RandomSeed()
Category:	Numeric functions, Random generator, xHarbour extensions
Source:	rtl\hbrandom.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates the three possibilities of generating
// random numbers with HB_Random().

   PROCEDURE Main
      // Random number between 0.01 and 0.99
      ? HB_Random()

      // Random number between 0.01 and 9.99
      ? HB_Random(10)

      // Random number between 8.01 and 9.99
      ? HB_Random(8,10)
   RETURN
```

HB_RandomInt()

Generates a random integer number between two boundaries.

Syntax

```
HB_RandomInt( [<nMaxLimit>] ) --> nRandomInteger
or
HB_RandomInt( <nMinLimit>, <nMaxLimit> ) --> nRandomInteger
```

Arguments

<nMaxLimit>

<nMaxLimit> is a numeric value specifying the upper boundary of the random number generator. It defaults to 1.

<nMinLimit>

<nMinLimit> is a numeric value specifying the lower boundary of the random number generator. It defaults to 0. Note that the first parameter is only interpreted as lower boundary when two parameters are passed.

Return

The function returns a random integer value, limited by the two boundaries.

Description

This function generates a random number between two boundaries. The boundary values are **included** in the range. When only one boundary is passed to the function, it is considered the maximum boundary. The minimum boundary, in that case, is zero. When no boundaries are passed, the function returns 0 or 1.

Note Use function HB_Random() to generate random numeric values with two decimals.

Info

See also: HB_Random(), HB_RandomSeed()
Category: Numeric functions, Random generator, xHarbour extensions
Source: rtl\hbrandom.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates the three possibilities of generating
// random numbers with HB_RandomInt().

   PROCEDURE Main
      // Random creation of 0 and 1
      ? HB_RandomInt()

      // Random number between 0 and 10
      ? HB_RandomInt(10)

      // Random number between 8 and 10
      ? HB_RandomInt(8,10)
   RETURN
```

HB_RandomSeed()

Sets the seed for generating random numbers.

Syntax

```
HB_RandomSeed( [<nSeed>] ) --> NIL
```

Arguments

<nSeed>

A numeric value used as seed for generating the random numbers returned by HB_Random() and HB_RandomInt().

Return

The function returns always NIL.

Description

This function sets the seed value for generating random numbers with HB_Random() and HB_RandomInt(). When a seed value is specified for the random number generator, it produces the same series of random numbers with each program invocation. This is useful for debugging purposes.

Calling the function without an argument resets the seed so that HB_Random() and HB_RandomInt() generate different series of random numbers each time a program is executed.

Info

See also:	HB_Random(), HB_RandomInt()
Category:	Numeric functions, Random generator, xHarbour extensions
Source:	rtl\hbrandom.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example generates two stings holding random upper case characters
// from A to Z. The first string is the same, each time the example is
// executed, while the second differs between program invocations.

   PROCEDURE Main
      LOCAL i, cStr1 := Space(10), cStr2 := Space(10)

      HB_RandomSeed(7)

      FOR i:=1 TO Len( cStr1 )
         cStr1[i] := HB_RandomInt(65,90)
      NEXT

      ? HB_RandomSeed()

      FOR i:=1 TO Len( cStr2 )
         cStr2[i] := HB_RandomInt(65,90)
      NEXT

      ? cStr1
      ? cStr2
   RETURN
```

HB_ReadIni()

Reads an INI file from disk.

Syntax

```
HB_ReadIni( <cFileName>  , ;
           [<lCaseSens>] , ;
           [<cDelimiter>], ;
           [<lAutoMain>]   ) --> hIniData
```

Arguments

`<cFileName>`

This is a character string holding the name of the INI file to open. It must include path and file extension. If the path is omitted from *<cFileName>*, the file is searched in the current directory.

`<lCaseSens>`

This parameter defaults to .T. (true) causing the function to return a case sensitive Hash(). When set to .F. (false), the returned hash is case insensitive.

`<cDelimiter>`

This is a character string holding the delimiter(s) recognized as separating character between key and value within the INI file. Bydefault, the characters ":", "=" and "|" are recognized.

`<lAutoMain>`

This parameter defaults to .T. (true) so that a [MAIN] section is automatically added to the resulting hash. All key/value pairs appearing at the beginning of the INI file outside a [section] are assigned to the hash key "MAIN".

Return

The function returns two dimensional hash holding sections and key/value pairs of each section inthe INI file. When the file does not exist, the return value is NIL.

Description

Function HB_ReadIni() reads an INI file and loads the data into a two dimensional Hash, which is returned. INI files are commonly used for configuration data of applications. They are comfortable to use since they can be created and/or changed with a regular ASCII editor.

The basic feature of an INI file is that it organizes data in key/value pairs which can be divided into sections:

```
; comment line: this is the auto-MAIN section

mainKey1=mainvalue1 # inline comment
mainKey2=mainvalue2

[SECTION1]
sectionKey1=valueA
sectionKey2=valueB

[SECTION2]
sectionKey1=valueC
sectionKey2=valueD
```

INI file sections are named. The names are enclosed in square brackets in the INI file and are used as keys in the Hash returnd by HB_ReadIni(). The value of each [section] key is, again, a hash, making the return value a two-dimensional hash. The hashes in the second dimension contain the key=value pairs of each [section] of the INI file.

The example above shows two key=value entries at the beginning of the INI file which are not part of a named [section]. By default, HB_ReadIni() assigns such entries to the hash key named "MAIN".

Info

See also: FOpen(), FRead(), Hash(), HB_SetIniComment(), HB_WriteIni()
Category: File functions, xHarbour extensions
Source: rtl\hbini.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example uses the INI file discussed in the function
// description and creates from it a case sensitive Hash.
// The key values of different sections are displayed.

   PROCEDURE Main
      LOCAL hIniData := HB_ReadIni( "test.ini" )

      ? Valtype( hIniData["MAIN"] )         // result: H
      ? Valtype( hIniData["SECTION1"] )     // result: H

      ? hIniData["MAIN"]["mainKey2"]        // result: mainvalue2

      ? hIniData["SECTION1"]["sectionKey1"] // result: valueA
      ? hIniData["SECTION1"]["sectionKey2"] // result: valueB

      ? hIniData["SECTION2"]["sectionKey1"] // result: valueC
      ? hIniData["SECTION2"]["sectionKey2"] // result: valueD

   RETURN
```

HB_ReadLine()

Scans a formatted character string or memo field for text lines.

Syntax

```
HB_ReadLine( <cText>            , ;
            [<cDelim>|<aDelim>], ;
             <nLineLen>         , ;
            [<nTabWidth>]       , ;
             <lWrap>            , ;
            [<nStartOffset>]    , ;
            @<lFound>           , ;
            @<lEOF>             , ;
            @<nEndOffSet>       , ;
            @<nEndOfLine>         ) --> NIL
```

Arguments

<cText>

This is a character string or memo field to extract a text line from. It can be a formatted text string that includes Tab and Hard/Soft carriage return characters.

<cDelim>|<aDelim>

This is either a character string holding the end-of-line character(s) to search in *<cText>*, or an array holding individual end-of-line character(s). The default value is determined by the SET EOL setting.

<nLineLen>

This is a numeric value specifying the number of characters that fit into one extracted line.

<nTabWidth>

A numeric value specifying the number of blank spaces the Tab character should be expanded to. It defaults to 4 blank spaces.

<lWrap>

A logical value indicating if word wrapping should be applied to *<cText>* when lines are extracted. The default value is .T. (true), resulting in extracted text lines that contain whole words only. When a word does not fit entirely to the end of the extracted text line, it is wrapped to the next text line. Passing .F. (false) for this parameter turns word wrapping off so that only lines ending with a hard carriage return are extracted.

<nStartOffset>

A numeric value specifying the first character in *<cText>* to begin finding the next text line. It defaults to 1.

@<lFound>

This parameter must be passed by reference. It receives a logical value: when *<lFound>* is .T. (true), the text line ends with *<cDelim>*, otherwise it does not include the end-of-line character(s).

@<lEOF>

This parameter must be passed by reference. It receives a logical value: when *<lEOF>* is .T. (true), no more text lines can be extracted from *<cText>*.

@<nEndOffSet>

This parameter must be passed by reference. It receives a numeric value indicating the end position of the found text line **without** the end-of-line character(s).

@<nEndOfLine>

This parameter must be passed by reference. It receives a numeric value indicating the end position of the found text line **including** the end-of-line character(s).

Return

The return value is always NIL.

Description

HB_ReadLine() is a fast text scanning function used to extract single text lines from a formatted text string that may include Tab characters (Chr(9)), Soft carriage returns (Chr(141)) or Hard carriage returns (Chr(13)). Text is usually scanned for single lines within a DO WHILE loop until <*lEOF*> is .T. (true).

Info

See also:	FLineCount(), HB_FReadLine(), MemoEdit(), MemoLine(), MemoRead(), MemoWrit(), MLCount(), MLPos()
Category:	Character functions, Memo functions, xHarbour extensions
Source:	rtl\txtline.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates how an entire text file can be transferred
// line by line into an array.

   PROCEDURE Main
      LOCAL cText        := MemoRead( "HB_ReadLine.prg" )
      LOCAL nLineLen     := 60
      LOCAL lWrap        := .T.
      LOCAL nEndOfLine   := 0
      LOCAL nStartOffset := 1
      LOCAL nEndOffSet   := 0
      LOCAL lFound       := .F.
      LOCAL lEof         := .F.
      LOCAL aLines       := {}
      LOCAL cLine

      CLS

      DO WHILE .NOT. lEof
         HB_ReadLine( cText        , ;
                      NIL          , ;
                      nLineLen     , ;
                      NIL          , ;
                      lWrap        , ;
                      nStartOffset, ;
                      @lFound      , ;
                      @lEof        , ;
                      @nEndOffset  , ;
                      @nEndOfLine    )

         cLine := SubStr( cText, nStartOffset, nEndOffset-nStartOffset+1 )
```

```
      IF cLine == ""
         cLine := " "  // empty line for AChoice()
      ENDIF

      AAdd( aLines, cLine )

      nStartOffSet := nEndOfLine
   ENDDO

   AChoice( ,,,, aLines )
RETURN
```

HB_RegEx()

Searches a string using a regular expression

Syntax

```
HB_RegEx( <cRegEx>          , ;
          <cString>         , ;
         [<lCaseSensitive>], ;
         [<lNewLine>]        ) --> aMatches
```

Arguments

`<cRegEx>`

This is a character string holding the search pattern as a literal regular expression. Alternatively, the return value of HB_RegExComp() can be used.

`<cString>`

This is the character string being searched for a match.

`<lCaseSensitive>`

This parameter defaults to .T. (true) so that a case sensitive search is performed. Passing .F. (false) results in a case insensitive search.

`<lNewLine>`

This parameter defaults to .F. (false).

Return

The function returns an array filled with the found substrings matching the regular expression. If no match is found, the return value is NIL.

Description

Function HB_RegEx() searches a character string using a regular expression and collects matching substrings in an array, which is returned. The first element of the returned array contains the whole match, while subsequent elements contain the portions of the whole match that comply with the regular expression. If the result in the first element does not match further, the returned array has one element only.

Note: the xHarbour Regular Expressions (RegEx) engine is using the PCRE implementation http://www.pcre.org/. Regular Expressions is a very powerful language specifically designed to analyze and match text patterns. The subject of RegEx is well beyond the scope of this documentation, and has been covered by many dedicated books. For a comprehensive description of RegEx in general, and PCRE in specific, please review http://www.pcre.org/.

A good introduction and tutorial on RegEx can also be found at http://www.regular-expressions.info/.

Info

See also: HAS, HB_AtX(), HB_RegExAll(), HB_RegExAtX(), HB_RegExComp(), HB_RegExSplit(), LIKE
Category: Character functions, Regular expressions, xHarbour extensions
Source: rtl\regex.c
LIB: xhb.lib
DLL: xhbdll.dll

Examples

```
// The example extracts a time from a text string by searching
// two pairs of digits separated by a colon.

   PROCEDURE Main
      LOCAL cRegEx := "[0-9]{2}:[0-9]{2}"
      LOCAL cText  := "Departure is at 18:45h from London airport"
      LOCAL aResult

      aResult := HB_RegEx( cRegEx, cText )

      AEval( aResult, {|c| QOut(c) } )
      ** Output:
      // 18:45
   RETURN

// The example matches an ISO formatted date string and displays
// the whole match and sub-matches.

   PROCEDURE Main
      LOCAL cRegEx := "([0-9]{4})[-/]([0-9]{1,2})[-/]([0-9]{1,2})"
      LOCAL aDates := { "2006-9-28", ;
                        "2006/10/1", ;
                        "2006-123-1" }
      LOCAL cDate, aMatch

      FOR EACH cDate IN aDates
          ?
          ? "Matching:   ", cDate

          aMatch := HB_RegEx( cRegEx, cDate )

          IF aMatch == NIL
             ? "No match"
          ELSEIF Len( aMatch ) == 1
             ? "Matched:", aMatch[1]

          ELSEIF Len( aMatch ) > 1
             ? "Whole match:", aMatch[1]
             ? "Sub matches: "
             AEVal( aMatch,{|c| QQOut( '"'+ c +'" ') }, 2 )
          ENDIF
      NEXT

      ** Output:
      // Matching:    2006-9-28
      // Whole match: 2006-9-28
      // Sub matches: "2006" "9" "28"
      //
      // Matching:    2006/10/1
      // Whole match: 2006/10/1
      // Sub matches: "2006" "10" "1"
```

```
    //
    // Matching:    2006-123-1
    // No match
RETURN
```

HB_RegExAll()

Parses a string and fills an array with parsing information.

Syntax

```
HB_RegExAll( <cRegEx>          , ;
             <cString>         , ;
            [<lCaseSensitive>], ;
            [<lNewLine>]       , ;
            [<nMaxMatches>]    , ;
            [<nWichMatch>]     , ;
            [<lMatchOnly>]       ) --> aAllRegexMatches
```

Arguments

<cRegEx>

This is a character string holding the search pattern as a literal regular expression. Alternatively, the return value of HB_RegExComp() can be used.

<cString>

This is the character string being searched for a match.

<lCaseSensitive>

This parameter defaults to .T. (true) so that the resulting compiled regular expression performs a case sensitive search. Passing .F. (false) results in a case insensitive search.

<lNewLine>

This parameter defaults to .F. (false).

<nMaxMatches>

This is a numeric value indicating the maximum number of matches to return. The default value is zero which returns all matches.

<nWichMatch>

This parameter is numeric and specifies the type of match to return. It defaults to zero which means that the function returns the whole match and sub-matches, if any. The value 1 instructs the function to return only whole matches, 2 means the first sub-match in the whole match, 3 is the second sub-match in the whole match, and so forth.

<lMatchOnly>

This parameter defaults to .T. (true) so that only matched substrings are included in the returned array. When passing .F. (false) the array includes the numeric start and end positions of the substrings found in *<cString>*.

Return

The function returns an array filled with the found substrings matching the regular expression. If no match is found, the return value is NIL. The array is two dimensional when *<lMatchOnly>* is .T. (default), and three dimensional when *<lMatchOnly>* is set to .F. (false).

Description

Function HB_RegExAll() searches a character string using a regular expression and collects matching substrings in an array, which is returned. The array is two dimensional when *<lMatchOnly>* is .T. (true), which is the default. The first column of the array contains whole matches, while subsequent columns contain sub-matches found in the whole match.

When <*lMatchOnly*> is .F. (false), the numeric start and end positions of the matched substrings are determined in addition and the result is a three dimensional array of the following structure:

```
aMatch := ;
{ ;
  { ;
    { <cWholeMatch1>, <nStart>, <nEnd> }, ;
    { <cSubMatch11> , <nStart>, <nEnd> }, ;
    { <cSubMatch1N> , <nStart>, <nEnd> }, ;
  } , ;
  { ;
    { <cWholeMatchN>, <nStart>, <nEnd> }, ;
    { <cSubMatchN1> , <nStart>, <nEnd> }, ;
    { <cSubMatchNN> , <nStart>, <nEnd> }, ;
  } ;
}
```

The example program outlines the possible return values of HB_RegExAll() in detail.

Info

See also:	HAS, HB_AtX(), HB_RegEx(), HB_RegExComp(), HB_RegExReplace(), HB_RegExSplit(), LIKE
Category:	Character functions, Regular expressions, xHarbour extensions
Source:	rtl\regex.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Examples

```
// The example demonstrates the return values of HB_RegExAll() by
// matching a text that contains 4 ISO formatted dates.

   PROCEDURE Main
      LOCAL cRegEx := "([0-9]{4})[-/]([0-9]{1,2})[-/]([0-9]{1,2})"
      LOCAL cText  := "Our shop is closed from 2006/10/25 to 2006/11/1 " + ;
                      "and 2006/12/24 to 2007/1/3"
      LOCAL aMatch, i, j

      CLS
      ? "Text to match:"
      ? cText
      ?
      ? "Matching without positions"
      ?
      ? "All matches and sub-matches"
      aMatch := HB_RegExAll( cRegEx, cText )
      AEval( aMatch, {|a| QOut(ValToPrg(a)) } )
      WAIT
      CLS

      ? "Text to match:"
      ? cText
      ?
      ? "Matching with positions"
      ?
      ? "One whole match"
      aMatch := HB_RegExAll( cRegEx, cText, .T., .T., 1, 1, .F. )
      AEval( aMatch, {|a| QOut(ValToPrg(a)) } )
      ?
      ? "First sub-match of one whole match"
      aMatch := HB_RegExAll( cRegEx, cText, .T., .T., 1, 2, .F. )
```

```
      AEval( aMatch, {|a| QOut(ValToPrg(a)) } )
      ?
      ? "Second sub-match of one whole match"
      aMatch := HB_RegExAll( cRegEx, cText, .T., .T., 1, 3, .F. )
      AEval( aMatch, {|a| QOut(ValToPrg(a)) } )
      ?
      ? "Third sub-match of one whole match"
      aMatch := HB_RegExAll( cRegEx, cText, .T., .T., 1, 4, .F. )
      AEval( aMatch, {|a| QOut(ValToPrg(a)) } )
      ?
      ? "Two whole matches"
      aMatch := HB_RegExAll( cRegEx, cText, .T., .T., 2, 1, .F. )
      AEval( aMatch, {|a| QOut(ValToPrg(a)) } )
      ?
      ? "First sub-match of two whole matches"
      aMatch := HB_RegExAll( cRegEx, cText, .T., .T., 2, 2, .F. )
      AEval( aMatch, {|a| QOut(ValToPrg(a)) } )
      ?
      ? "Second sub-match of two whole matches"
      aMatch := HB_RegExAll( cRegEx, cText, .T., .T., 2, 3, .F. )
      AEval( aMatch, {|a| QOut(ValToPrg(a)) } )
      ?
      ? "Third sub-match of two whole matches"
      aMatch := HB_RegExAll( cRegEx, cText, .T., .T., 2, 4, .F. )
      AEval( aMatch, {|a| QOut(ValToPrg(a)) } )
      ?
      ? "All whole matches"
      aMatch := HB_RegExAll( cRegEx, cText, .T., .T., 0, 1, .F. )
      AEval( aMatch, {|a| QOut(ValToPrg(a)) } )
      ?
      ? "All first sub-matches"
      aMatch := HB_RegExAll( cRegEx, cText, .T., .T., 0, 2, .F. )
      AEval( aMatch, {|a| QOut(ValToPrg(a)) } )

      WAIT
      CLS
      ? "Text to match:"
      ? cText
      ?
      ? "All whole matches including sub-matches"
      aMatch := HB_RegExAll( cRegEx, cText,,,,, .F. )

      FOR i:=1 TO Len( aMatch )
          ?
          FOR j:=1 TO Len( aMatch[i] )
              IF j == 1
                 ? "Whole match"
              ELSE
                 ? "- sub-match"
              ENDIF
              ?? " found:", aMatch[i,j,1]
              ?? " Start:", aMatch[i,j,2]
              ?? " End:"  , aMatch[i,j,3]
          NEXT
      NEXT
   RETURN

// The example locates the opening and closing tags in an HTML or XML
// file and displays them along with their positions. Note that the
// end position of opening tags is found with the At() function since
// opening HTML and XML tags may include attributes.
```

```
PROCEDURE Main
   LOCAL cText := Memoread( "Test.htm" )
   LOCAL cRegEx := "<[A-Z][A-Z0-9]*|</[A-Z][A-Z0-9]*>"
   LOCAL aMatch, i, nStart, nEnd

   aMatch := HB_RegExAll( cRegEx, cText, .F., .T., 0, 0, .F. )

   FOR i:=1 TO Len( aMatch )
      IF Left( aMatch[i, 1, 1], 2 ) == "</"
         ? "Closing tag:", aMatch[i, 1, 1], ;
                "Start:", aMatch[i, 1, 2], ;
                  "End:", aMatch[i, 1, 3]
      ELSE
         nStart := aMatch[i, 1, 2]
         nEnd   := At( ">", cText, aMatch[i, 1, 3] )

         ? "Opening tag:", SubStr( cText, nStart, nEnd-nStart+1 ), ;
                "Start:", nStart, ;
                 "End:" , nEnd
      ENDIF
   NEXT
RETURN
```

HB_RegExAtX()

Parses a string and fills an array with parsing information.

Syntax

```
HB_RegExAtX( <cRegEx>          , ;
             <cString>         , ;
            [<lCaseSensitive>], ;
            [<lNewLine>]        ) --> aMatches
```

Arguments

`<cRegEx>`

This is a character string holding the search pattern as a literal regular expression. Alternatively, the return value of HB_RegExComp() can be used.

`<cString>`

This is the character string being searched for a match.

`<lCaseSensitive>`

This parameter defaults to .T. (true) so that a case sensitive search is performed. Passing .F. (false) results in a case insensitive search.

`<lNewLine>`

This parameter defaults to .F. (false).

Return

The function returns a two dimensional array. Each element of the array holds an array of three elements holding the found substring plus its start and end position. If no match is found, the return value is NIL.

Description

Function HB_RegExAtX() searches a character string using a regular expression and collects matching substrings along with their starting and ending position in a two dimensional array, which is returned. The first element of the returned array contains the whole match, while subsequent elements contain the portions of the whole match that comply with the regular expression. If the result in the first element does not match further, the returned array has one element only.

Info

See also:	HAS, HB_AtX(), HB_RegEx(), HB_RegExAll(), HB_RegExComp(), HB_RegExSplit(), LIKE
Category:	Character functions, Regular expressions, xHarbour extensions
Source:	rtl\regex.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example matches an ISO formatted date string and displays
// the whole match and sub-matches along with theri positoins in
// the searched strings.

   PROCEDURE Main
      LOCAL cRegEx := "([0-9]{4})[-/]([0-9]{1,2})[-/]([0-9]{1,2})"
      LOCAL aDates := { "2006-9-28", ;
                        "2006/10/1", ;
```

```
                  "2006-123-1" }
   LOCAL cDate, aMatch, i

   FOR EACH cDate IN aDates
       ?
       ? "Matching:   ", cDate

       aMatch := HB_RegExAtX( cRegEx, cDate )

       IF aMatch == NIL
          ? "No match"
       ELSE
          ? "Whole match:", PadL(aMatch[1,1],10), ;
                  "Start:", PadL(aMatch[1,2], 2), ;
                    "End:", PadL(aMatch[1,3], 2)

          FOR i:=2 TO Len( aMatch )
             ? "  Sub match:", PadL(aMatch[i,1],10), ;
                     "Start:", PadL(aMatch[i,2], 2), ;
                       "End:", PadL(aMatch[i,3], 2)
          NEXT
       ENDIF
   NEXT

   ** Output:
   // Matching:    2006-9-28
   // Whole match:  2006-9-28 Start:  1 End:  9
   //   Sub match:       2006 Start:  1 End:  4
   //   Sub match:          9 Start:  6 End:  6
   //   Sub match:         28 Start:  8 End:  9
   //
   // Matching:    2006/10/1
   // Whole match:  2006/10/1 Start:  1 End:  9
   //   Sub match:       2006 Start:  1 End:  4
   //   Sub match:         10 Start:  6 End:  7
   //   Sub match:          1 Start:  9 End:  9
   //
   // Matching:    2006-123-1
   // No match
RETURN
```

HB_RegExComp()

Compiles a regular expression into a binary search pattern.

Syntax

```
HB_RegExComp( <cRegEx>          , ;
             [<lCaseSensitive>], ;
             [<lNewLine>]        ) --> cBinaryRegEx
```

Arguments

<cRegEx>

This is a character string holding a literal regular expression.

<lCaseSensitive>

This parameter defaults to .T. (true) so that the resulting compiled regular expression performs a case sensitive search. Passing .F. (false) results in a case insensitive search.

<lNewLine>

This parameter defaults to .F. (false).

Return

The function returns a character string holding *<cRegEx>* as a compiled binary regular expression. If *<cRegEx>* is not a valid regular expression, a runtime error is generated.

Description

All regular expression functions and operators can process character strings containing literal regular expressions or compiled regular expressions. A literal regular expression is compiled with HB_RegExComp() to speed up pattern matching. This is advantageous when the same regular expression is used very often in a search routine.

Note: function HB_IsRegExString() can be used to test if a variable holds a compiled regular expression.

Info

See also:	HAS, HB_AtX(), HB_IsRegExString(), HB_RegEx(), HB_RegExAll(), HB_RegExSplit(), LIKE
Category:	Character functions, Regular expressions, xHarbour extensions
Source:	rtl\regex.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_RegExMatch()

Tests if a string contains a substring using a regular expression

Syntax

```
HB_RegExMatch( <cRegEx>          , ;
                <cString>        , ;
               [<lCaseSensitive>], ;
               [<lNewLine>]        ) --> lFound
```

Arguments

`<cRegEx>`

This is a character string holding the search pattern as a literal regular expression. Alternatively, the return value of HB_RegExComp() can be used.

`<cString>`

This is the character string being searched for a match.

`<lCaseSensitive>`

This parameter defaults to .T. (true) so that a case sensitive match is performed. Passing .F. (false) results in a case insensitive match.

`<lNewLine>`

This parameter defaults to .F. (false).

Return

The function returns .T. (true), when *<cString>* contains a substring matching the regular expression *<cRegEx>*, otherwise .F. (false) is returned.

Description

HB_RegExMatch() is a simple regular expression function that tests if a text string contains a substring matching the regular expression. It is similar to the HAS operator.

Info

See also:	HAS, HB_AtX(), HB_RegEx(), HB_RegExAll(), HB_RegExComp(), HB_RegExSplit(), LIKE
Category:	Character functions, Regular expressions, xHarbour extensions
Source:	rtl\regex.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example tests if a text string contains an eMail address.
// The search is case insensitive although the RegEx defines
// character classes only with upper case letters.

   PROCEDURE Main
      LOCAL cRegEx := "[A-Z0-9._%-]+@[A-Z0-9.-]+\.[A-Z]{2,4}"
      LOCAL cText  := "Send your request to info@xharbour.com " + ;
                      "for more information"

      IF HB_RegExMatch( cRegEx, cText, .F. )
         ? "Text contains eMail address"
      ELSE
```

```
      ? "Text contains no eMail address"
   ENDIF
RETURN
```

HB_RegExReplace()

Searches and replaces characters within a character string using a regular expression.

Syntax

```
HB_RegExReplace(<cRegEx>         , ;
                 <cString>        , ;
                 <cReplace>       , ;
                [<lCaseSensitive>], ;
                [<lNewLine>]      , ;
                [<nMaxMatches>]   , ;
                [<nWichMatch>]      ) --> cNewString
```

Arguments

<cRegEx>

This is a character string holding the search pattern as a literal regular expression. Alternatively, the return value of HB_RegExComp() can be used.

<cString>

This is the character string being searched for a match.

<cReplace>

A character string that replaces all matched substrings in *<cString>*.

<lCaseSensitive>

This parameter defaults to .T. (true) so that the resulting compiled regular expression performs a case sensitive search. Passing .F. (false) results in a case insensitive search.

<lNewLine>

This parameter defaults to .F. (false).

<nMaxMatches>

This is a numeric value indicating the maximum number of replacements. The default value is zero which replaces all matches.

<nWichMatch>

This parameter is numeric and specifies the type of matches to replace. It defaults to zero which means that the function replaces the whole match and sub-matches, if any. The value 1 instructs the function to replace only whole matches, 2 means the first sub-match in the whole match, 3 is the second sub-match in the whole match, and so forth.

Return

The function returns a copy of *<String>* where substrings matching the regular expression are replaced with *<cReplace>*.

Description

Function HB_RegExReplace() searches a character string using a regular expression and replaces matching substrings with *<cReplace>*. It is similar to the function StrTran() but more powerful since the search condition is a regular expression.

Info

See also:	HAS, HB_AtX(), HB_RegEx(), HB_RegExAll(), HB_RegExComp(), HB_RegExSplit(), LIKE, StrTran(), Stuff()
Category:	Character functions, Regular expressions, xHarbour extensions
Source:	rtl\regexrpl.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_RegExSplit()

Parses a string using a regular expression and fills an array.

Syntax

```
HB_RegExSplit( <cRegEx>         , ;
               <cString>        , ;
              [<lCaseSensitive>, ;
              [<lNewLine>]      , ;
              [<nMaxMatches>]     ) --> aSplitString
```

Arguments

<cRegEx>

This is a character string holding the search pattern as a literal regular expression. Alternatively, the return value of HB_RegExComp() can be used.

<cString>

This is the character string being searched for a match.

<lCaseSensitive>

This parameter defaults to .T. (true) so that a case sensitive search is performed. Passing .F. (false) results in a case insensitive search.

<lNewLine>

This parameter defaults to .F. (false).

<nMaxMatches>

This is a numeric value indicating the maximum number of matches to return. The default value is zero which returns all matches.

Return

The function returns an array whose elements hold the parsing result. If no match is found, the return value is NIL.

Description

Function HB_RegExSplit() scans a character string for substrings matching a regular expression. All matched substrings are removed while the remaining strings are collected and returned in an array.

Info

See also:	HAS, HB_AtX(), HB_RegEx(), HB_RegExAll(), HB_RegExComp(), HB_RegExReplace(), LIKE
Category:	Character functions, Regular expressions, xHarbour extensions
Source:	rtl\regex.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates the result of parsing a text string
// using different regular expressions.

   PROCEDURE Main
      LOCAL cRegEx := "[A-Z0-9._%-]+@[A-Z0-9.-]+\.[A-Z]{2,4}"
      LOCAL cText  := "Send your request to info@xharbour.com " + ;
                      "for more information"
```

```
    LOCAL aResult, cEmail

    CLS

    ? "----- words in text -----"
    aResult := HB_RegExSplit( " ", cText )
    AEval( aResult, {|c| QOut(c) } )

    cEmail := HB_AtX( cRegEx, cText, .F. )
    ?
    ? "----- email address -----"
    ? cEMail

    aResult := HB_RegExSplit( "[.@]", cEMail )

    ?
    ? "----- email components --"
    AEval( aResult, {|c| QOut(c) } )

    ** Output
    // ----- words in text -----
    // Send
    // your
    // request
    // to
    // info@xharbour.com
    // for
    // more
    // information
    //
    // ----- email address -----
    // info@xharbour.com
    //
    // ----- email components --
    // info
    // xharbour
    // com
 RETURN
```

HB_ResetWith()

Replaces the *with* object during WITH OBJECT execution.

Syntax

```
HB_ResetWith( <oObject> ) --> NIL
```

Arguments

`<oObject>`

This is an object which becomes the new *with* object in the current WITH OBJECT .. END context.

Return

The return value is always NIL.

Description

Function HB_ResetWith() replaces the current *with* object in the context of the WITH OBJECT statement. All abbreviated messages within the WITH OBJECT .. END block are addressed to this new *with* object.

In contrast to the HB_SetWith() function, HB_ResetWith() does not open a new WITH OBJECT context. As a result, the nesting level of WITH OBJECT and the return value of HB_WithObjectCounter() remain unchanged.

Info

See also:	HB_SetWith(), HB_QWith(), HB_WithObjectCounter(), WITH OBJECT
Category:	Object functions, xHarbour extensions
Source:	vm\hvm.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows how to change the current "with" object
// without changing the nesting level of WITH OBJECT

   PROCEDURE Main()
      LOCAL oSavedWith

      WITH OBJECT ErrorNew()
         ? :className()            // result: ERROR
         ? HB_WithObjectCounter()    // result: 1

         oSavedWith := HB_QWith()

         HB_ResetWith( GetNew() )

         ? :className()            // result: GET
         ? HB_WithObjectCounter()    // result: 1

         HB_ResetWith( oSavedWith )

         ? :className()            // result: ERROR
         ? HB_WithObjectCounter()    // result: 1
      END
```

```
RETURN
```

HB_RestoreBlock()

Converts binary information back to a code block.

Syntax

```
HB_RestoreBlock( <aCodeblockInfo> ) --> bCodeblock
```

Arguments

<aCodeblockInfo>

This is the array returned from HB_SaveBlock().

Return

The function returns the code block previously converted with HB_SaveBlock().

Description

Function HB_RestoreBlock() is a utility function for HB_DeSerialize(). It converts the information stored in the return value of HB_SaveBlock() to a code block. Refer to function HB_Serialize() for comprehensive information about (de)serialization of code blocks.

Info

See also:	HB_DeSerialize(), HB_SaveBlock(), HB_Serialize(), RESTORE, SAVE
Category:	Code block functions, Conversion functions, xHarbour extensions
Source:	vm\hvm.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_SaveBlock()

Utility function for code block serialization.

Syntax

```
HB_SaveBlock( <bCodeblock> ) --> aCodeBlockInfo
```

Arguments

<bCodeblock>

This is the codeblock to convert to a binary form.

Return

The function returns an array holding information about the serialized code block.

Description

HB_SaveBlock() converts a code block to its binary representation and collects this information in an array, which is returned. The array can be passed to HB_RestoreBlock() to rebuild the code block. The function is a utility function internally used by HB_Serialize(). Refer to this function for limitations on code block serialization.

Info

See also:	HB_DeSerialize(), HB_RestoreBlock(), HB_Serialize(), RESTORE, SAVE
Category:	Code block functions, Conversion functions, xHarbour extensions
Source:	vm\hvm.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_Serialize()

Converts an arbitrary value to a binary string.

Syntax

```
HB_Serialize( <xValue> ) --> cBinary
```

Arguments

<xValue>

This is an arbitrary value to convert to a binary string. The value can be of any data type, except Pointer.

Return

The function returns the binary representation of *<xValue>* as a character string.

Description

Function HB_Serialize() converts an arbitrary value to its binary string representation. The binary string can then be treated like a regular character string and written to a file, for example. The original value can be obtained by passing the binary string to HB_DeSerialize().

Although values of any data type can be passed to the function, certain limitations must be complied with when values are (de)serialized:

Simple data types

There is no limit for the serialization of values having the data type Character, Date, Numeric, Logic or NIL

Arrays and Hashes

No limitation exists for arrays and hashes when their elements contain values of simple data types. Nested arrays and hashes can be serialized as long as they do not reference themselves in their elements. When complex data types are stored in arrays or hashes, the same limitations as for the respective data type exist for arrays and hashes.

Code blocks

As a general rule, only code blocks that access items having a symbolic name at runtime can be serialized. This excludes code blocks referencing GLOBAL, LOCAL or STATIC variables and/or STATIC FUNCTIONs and PROCEDUREs from conversion, since their symbolic names do not exist at runtime. Passing such a code block to HB_Serialize() generates a runtime error.

The code block can only be deserialized in the exact same binary that converted the code block. That means, only executables that are created with the same PCode version of xHarbour and have the same symbol table can deserialize a serialized code block.

Objects

The same limits as for code blocks exist for objects. It is recommended to subclass from the HBPersistent() class when objects should be serialized.

Pointers

Pointers cannot be serialized since they refer to memory locations at runtime. HB_Serialize() returns always NIL when a pointer is passed.

Info

See also: HB_Deserialize(), HB_SerializeSimple(), HB_SerialNext(), RESTORE, SAVE
Category: Conversion functions, Serialization functions, xHarbour extensions
Source: rtl\hbserial.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// In this example, a code block is created and stored in a MEM file.
// The second invocation of the example code loads the stored code block
// and evaluates it. Note that the code block referenes a PRIVATE variable
// that is restored along with the code block.

   PROCEDURE Main
      LOCAL bBlock := {|| Alert( pcString ) }

      IF .NOT. File( "Codeblock.mem" )

         PRIVATE pcString := "code block restored from file"
         PRIVATE pcBlock  := HB_Serialize( bBlock )

         SAVE TO Codeblock.mem ALL LIKE pc*
         ? "Restart program to load code block"

      ELSE
         RESTORE FROM Codeblock.mem

         bBlock := HB_DeSerialize( pcBlock )

         Eval( bBlock )
      ENDIF

   RETURN
```

HB_SerializeSimple()

Serializes values of simple data types.

Syntax

```
HB_SerializeSimple( <xValue> ) --> cBinaryData
```

Arguments

<xValue>

This is a value of simple data type, i.e. Valtype() must be C, D, L , M, N or U.

Return

The function returns the binary representation of the passed value as a character string.

Description

Function HB_SerializeSimple() is a utility function of xHarbour's serialization system, optimized for serializing simple data types only. These are Character, Date, Logic, Memo, Numeric and undefined (NIL). Complex data types (Array, Code block, Hash and Object) must be serialized using the functions HB_Serialize().

Note: refer to the file source\rtl\hbserial.prg for a usage example of HB_SerializeSimple().

Info

See also:	HB_Deserialize(), HB_DeserialBegin(), HB_Serialize()
Category:	Serialization functions, xHarbour extensions
Source:	rtl\hbsrlraw.c, rtl\hbserial.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_SerialNext()

Returns the position of the next chunk of binary data to retrieve.

Syntax

```
HB_SerialNext( <cBinaryData> ) --> nPos
```

Arguments

<cBinaryData>

This is a character string obtained from one or multiple calls to HB_Serialize(). It holds the binary data of one ore more previously serialized variable(s).

Return

The function returns the starting position of the next value encoded in *<cBinaryData>*. This value can be passed to SubStr().

Description

Function HB_SerialNext() is a utility function of xHarbour's serialization system. It finds the position of the next chunk of binary data in the serialization string.

Note: refer to the file source\rtl\hbserial.prg for a usage example of HB_SerialNext().

Info

See also:	HB_Deserialize(), HB_GetLen8(), HB_Serialize()
Category:	Serialization functions, xHarbour extensions
Source:	rtl\hbsrlraw.c, rtl\hbserial.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_SetCodePage()

Queries or changes the current code page.

Syntax

```
HB_SetCodePage( [<cCodePageID>] ) --> cOldCodePageID
```

Arguments

<cCodePageID>

This is a character string identifying the code page to select as current (see table below).

Return

The function returns a character string identifying the code page selected before the function is called.

Description

Function HB_SetCodePage() queries and optionally changes the current code page used for character strings. Code pages differ by the character set (ANSI/OEM) and national language. They define the "sorting weight" of characters in the national alphabet.

By default, code page 437 for the English language is selected. In order to change the code page to a different character set, the corresponding module must be linked. That is, the symbolic names of the code pages to be available at runtime must be REQUESTed.

When the code page module is linked, the code page can be changed at runtime by passing the code page identifier *<cCodePageID>* to HB_SetCodePage(). The following table lists the REQUEST symbols and code page identifiers available in xHarbour.

Code page support in xHarbour

Language name	Code page	REQUEST symbol	*<cCodePageID>*	Source file
Bulgarian	Windows-1251	HB_CODEPAGE_BG1251	"BG1251"	cpbgwin.c
Bulgarian	MIK	HB_CODEPAGE_BGMIK	"BGMIK"	cpbgmik.c
Croatien	1250	HB_CODEPAGE_HR1250	"HR1250"	cphr1250.c
Croatien	437	HB_CODEPAGE_HR437	"HR437"	cphr437.c
Croatien	852	HB_CODEPAGE_HR852	"HR852"	cphr852.c
English	437	none	"EN"	cp_tpl.c
French	850	HB_CODEPAGE_FR	"FR"	cpfrdos.c
German	850	HB_CODEPAGE_DE	"DE"	cpgedos.c
German	ISO-8859-1	HB_CODEPAGE_DEWIN	"DEWIN"	cpgewin.c
Greek (Dos)	737	HB_CODEPAGE_EL	"EL"	cpeldos.c
Greek WIN	ANSI (1253)	HB_CODEPAGE_ELWIN	"ELWIN"	cpelwin.c
Hungarian	852	HB_CODEPAGE_HU852	"HU852"	cphu852.c
Hungarian	Windows-1250	HB_CODEPAGE_HUWIN	"HUWIN"	cphuwin.c
Italian	437	HB_CODEPAGE_IT437	"IT437"	cpit437.c
Italian	850	HB_CODEPAGE_IT850	"IT850"	cpit850.c
Italian	ISO-8859-1	HB_CODEPAGE_ITISO	"ITISO"	cpitiso.c
Italian	ISO-8859-1b (with BOX chars)	HB_CODEPAGE_ITISB	"ITISB"	cpitisb.c
Lithuanian	Windows-1257	HB_CODEPAGE_LT	"LT"	cpltwin.c
Polish	852	HB_CODEPAGE_PL852	"PL852"	cppl852.c
Polish	ISO-8859-2	HB_CODEPAGE_PLISO	"PLISO"	cppliso.c
Polish	Mazovia	HB_CODEPAGE_PLMAZ	"PLMAZ"	cpplmaz.c
Polish	Windows-1250	HB_CODEPAGE_PLWIN	"PLWIN"	cpplwin.c
Portuguese	850	HB_CODEPAGE_PT850	"PT850"	cppt850.c
Portuguese	ISO-8859-1	HB_CODEPAGE_PTISO	"PTISO"	cpptiso.c

Russian	Windows-1251	HB_CODEPAGE_RU1251	"RU1251"	cpruwin.c
Russian	866	HB_CODEPAGE_RU866	"RU866"	cpru866.c
Russian	KOI-8	HB_CODEPAGE_RUKOI8	"RUKOI8"	cprukoi.c
Serbian	Windows-1251	HB_CODEPAGE_SRWIN	"SRWIN"	cpsrwin.c
Slovenian	852	HB_CODEPAGE_SL852	"SL852"	cpsl852.c
Slovenian	ISO-8859-2	HB_CODEPAGE_SLISO	"SLISO"	cpsliso.c
Slovenian	Windows-1250	HB_CODEPAGE_SLWIN	"SLWIN"	cpslwin.c
Spanish	850	HB_CODEPAGE_ES	"ES"	cpesdos.c
Spanish (Modern)	ISO-8859-1	HB_CODEPAGE_ESMWIN	"ESMWIN"	cpesmwin.c
Spanish	ISO-8859-1	HB_CODEPAGE_ESWIN	"ESWIN"	cpeswin.c
Ukrainian	Windows-1251	HB_CODEPAGE_UA1251	"UA1251"	cpuawin.c
Ukrainian	866	HB_CODEPAGE_UA866	"UA866"	cpua866.c
Ukrainian	KOI-8U	HB_CODEPAGE_UAKOI8	"UAKOI8"	cpuakoi.c

Info

See also: HB_AnsiToOem(), HB_LangSelect(), HB_OemToAnsi(), HB_Translate()
Category: Language specific, xHarbour extensions
Source: rtl\cdpapi.c, codepage\cp*.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates the influence of the selected code page
// on character comparison. It uses German umlauts and shows their
// sorting weight in the English and German alphabet.

   REQUEST HB_CODEPAGE_DEWIN

   PROCEDURE Main
      LOCAL aChr := { "A", "B", "O", "P", "U", "V", "-", "Ö", "Ü" }
      LOCAL cDE  := "", cEN := ""

      ? HB_SetCodePage()              // result: EN
      ? "-" > "B"                     // result: .T.

      ASort( aChr )
      ? "Sorted: "
      cEN := ""
      AEval( aChr, {|c| cEN += c } )
      ? HB_AnsiToOem( cEN )           // result: ABOPUV-ÖÜ

      HB_SetCodePage( "DEWIN" )

      ? HB_SetCodePage()              // result: DEWIN
      ? "-" > "B"                     // result: .F.

      ASort( aChr )
      ? "Sorted: "
      AEval( aChr, {|c| cDE += c } )
      ? HB_AnsiToOem( cDE )           // result: A-BOÖPUÜV
   RETURN
```

HB_SetIniComment()

Defines the delimiting characters for comments and inline comments.

Syntax

```
HB_SetIniComment( <cLineComment>, <cInlineComment> ) --> NIL
```

Arguments

<cLineComment>

This is a single character used to indicate comment lines. The default is ";".

<cInlineComment>

This is a single character used to indicate inline comments. The default is "#".

Return

The return value is always NIL.

Description

INI files may contain comment lines. They are recognized by a single character which can be changed with function HB_SetIniComment().

Info

See also:	HB_ReadIni(), HB_WriteIni()
Category:	File functions, xHarbour extensions
Source:	rtl\hbini.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// This example demonstrates the two possibilities for
// comments in an INI file.

   ; this is a line comment
   [sect1]
   key1 = value1 # this is an inline comment
```

HB_SetKeyArray()

Associates a code block with multiple keys.

Syntax

```
HB_SetKeyArray( <aInkey>, [<bCodeblock>] ) --> NIL
```

Arguments

<aInkey>

This is a one dimensional array. It contains the numeric codes of the keys to associate *<bCodeblock>* with. The key code is returned by the Inkey() function. To specify values for the array elements use #define constants from the file INKEY.CH.

<bNewCodeblock>

This parameter optionally specifies a code block to be executed when any key specified with *<aInkey>* is pressed. Passing NIL explicitly for *<bCodeblock>* releases a previously assigned code block for all keys in *<aInkey>*.

Return

The function returns always NIL.

Description

The function iterates the array *<aInkey>* and associates the code block *<bCodeblock>* with each Inkey() code contained in the array. This is equivalent to the following expression:

```
bBlock := <a code block or NIL>
AEval( aInkey, {|nKey| SetKey( nKey, bBlock ) } )
```

Info

See also:	AEval(), Inkey(), RestSetkey(), SaveSetkey(), SetKey()
Category:	Environment functions, Keyboard functions, xHarbour extensions
Source:	rtl\setkey.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_SetKeyCheck()

Evaluates a code block associated with a key.

Syntax

```
HB_SetKeyCheck( <nInkey> , ;
               [<param1>], ;
               [<param2>], ;
               [<param3>]  ) --> lIsSetkeyBlock
```

Arguments

<nInkey>

This is a numeric value representing the key code of the key associated with a code block. The key code is returned by the Inkey() function. To specify values for this parameter use #define constants from the file INKEY.CH.

<param1>..<param3>

Optionally, up to three arbitrary values can be specified. They are passed on to the code block associated with <*nInkey*>.

Return

The function returns .T. (true) when the key <*nInkey*> is associated with a code block and the code block is evaluated, otherwise .F. (false) is returned.

Description

The function tests if a key is associated with a code block. Key/code block associations are defined with the SetKey() function. When the key <*nInkey*> is associated with a code block, HB_SetKeyCheck() evaluates this code block and optionally passes up to three parameters on to it.

Info

See also:	Eval(), Inkey(), RestSetkey(), SaveSetkey(), SetKey()
Category:	Environment functions, Keyboard functions, xHarbour extensions
Source:	rtl\setkey.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_SetKeyGet()

Retrieves code blocks associated with a key.

Syntax

```
HB_SetKeyGet( <nInkeyCode> , ;
             [@<bCondition>] ) --> bCodeblock
```

Arguments

`<nInkeyCode>`

This is a numeric value representing the key code of the key to retrieve associated code blocks for. The key code is returned by the Inkey() function. To specify values for this parameter use #define constants from the file INKEY.CH.

`@<bCondition>`

If specified, this parameter must be passed by reference. It gets assigned the code block defined with the third parameter of the SetKey() function.

Return

The function returns the code block associated with *<nInkeyCode>* using the SetKey() function. The optional second SetKey() code block is retrieved when *<bCondition>* is passed by reference.

Info

See also:	Inkey(), RestSetkey(), SaveSetkey(), SetKey()
Category:	Environment functions, Keyboard functions, xHarbour extensions
Source:	rtl\setkey.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_SetKeySave()

Queries or changes all SetKey() code blocks.

Syntax

```
HB_SetKeySave( [<aNewKeyBlock>] ) --> aOldKeyBlock
```

Arguments

<aNewKeyBlock>

This is an optional two dimensional array with three columns. Each array element is an array holding three values. They are used as parameters for the SetKey() function. If <*aNewKeyBlock*> is an array, new key/code block associations are defined.

When NIL is explicitly passed for <*aNewKeyBlock*>, all active SetKey() code blocks are voided.

Return

The function returns a two dimensional array holding the previous key/code block associations.

Description

HB_SetKeySave() queries all active key/code block associations defined with the SetKey() function. Optionally, new SetKey() code blocks can be defined with <*aNewKeyBlock*> or they can be voided when NIL is passed to the function.

When the return value is passed to HB_SetKeySave(), all previously defined SetKey() code blocks become active again.

Info

See also:	Inkey(), RestSetkey(), SaveSetkey(), SetKey()
Category:	Environment functions, Keyboard functions, xHarbour extensions
Source:	rtl\setkey.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example defines SetKey() code blocks which branch to
// sub-routines while READ is executed. To suppress recursive
// calls within sub-routines, all key/code block associations
// are voided and restored with HB_SetKeySave().

   #include "Inkey.ch"

   PROCEDURE Main
      LOCAL cString := "Testing HB_SetKeySave()"

      CLS
      SET KEY K_F2 TO F2_Key
      SET KEY K_F3 TO F3_Key

      @ 10,10 SAY "Press F1, F2 or F3" GET cString
      READ

   RETURN
```

```
PROCEDURE Help( cProcName, nProcLine, cReadVar )
   LOCAL cScreen := SaveScreen()
   LOCAL aSaved  := HB_SetKeySave( NIL ) // no recursive calls
   CLS
   ? "Help routine"
   ? "Called from:", cProcName, nProcLine, cReadVar
   WAIT
   HB_SetKeySave( aSaved )
   Restscreen(,,,, cScreen )
RETURN


PROCEDURE F2_Key( cProcName, nProcLine, cReadVar )
   LOCAL cScreen := SaveScreen()
   LOCAL aSaved  := HB_SetKeySave( NIL ) // no recursive calls
   CLS
   ? "F2 key pressed"
   ? "Called from:", cProcName, nProcLine, cReadVar
   WAIT
   HB_SetKeySave( aSaved )
   Restscreen(,,,, cScreen )
RETURN


PROCEDURE F3_Key( cProcName, nProcLine, cReadVar )
   LOCAL cScreen := SaveScreen()
   LOCAL aSaved  := HB_SetKeySave( NIL ) // no recursive calls
   CLS
   ? "F3 key pressed"
   ? "Called from:", cProcName, nProcLine, cReadVar
   WAIT
   HB_SetKeySave( aSaved )
   Restscreen(,,,, cScreen )
RETURN
```

HB_SetMacro()

Enables or disables runtime behavior of the macro compiler.

Syntax

```
HB_SetMacro( <nMode>, [<lOnOff>] ) --> lOldSetting
```

Arguments

<nMode>

This is a numeric value used to enable/disable features of the macro compiler. #define constants listed in Hbmacro.ch are used to query or toggle individual features.

Features of the macro compiler

Constant	Value	Default	Description
HB_SM_HARBOUR	1	ON	Support inline operators
HB_SM_XBASE	2	ON	Support comma separated expression lists
HB_SM_PREPROC	4	OFF	Support commands preprocessing
HB_SM_SHORTCUTS	8	ON	Support shortcuts for logical operators

<lOnOff>

A logical value. .T. (true) enables the feature specified with *<nMode>*, and .F. (false) disables it.

Return

The function returns the previous setting, active before HB_SetMacro() is called.

Description

This function enables or disables some features of the macro compiler. xHarbour is extending the macro features compared to an original set available in Clipper. Enabling/disabling some of them allows to keep strict Clipper compatibility.

Info

See also: & (macro operator)
Category: Indirect execution, xHarbour extensions
Source: vm\macro.c
LIB: xhb.lib
DLL: xhbdll.dll

HB_SetWith()

Changes the *with* object during WITH OBJECT execution.

Syntax

```
HB_SetWith( [<oNewObject>] ) --> oOldObject
```

Arguments

<oNewObject>

If specified, *<oNewObject>* becomes the new *with* object, while *oOldObject* is pushed on the WITH OBJECT execution stack. If *<oNewObject>* is omitted, the last object from the WITH OBJECT execution stack becomes the current *with* object.

Return

The function returns the previous *with* object active in the WITH OBJECT statement.

Description

Function HB_SetWith() changes the current *with* object in the context of the WITH OBJECT statement. All abbreviated messages within the WITH OBJECT .. END block are addressed to the current *with* object.

Passing an object to HB_SetWith() opens a new WITH .. OBJECT context, and pushes the current *with* object to the WITH OBJECT execution stack. As a result, function HB_WithObjectCounter() returns a value increased by 1.

The new WITH OBJECT context is closed by calling HB_SetWith() without parameter, so that a previous *with* object is removed from the WITH OBJECT execution stack, and HB_WithObjectCounter() returns a value decreased by 1.

Info

See also:	HB_QSelf(), HB_ResetWith(), HB_QWith(), HB_WithObjectCounter(), WITH OBJECT
Category:	Object functions, xHarbour extensions
Source:	include\hbexprb.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example outlines the effect of calling HB_SetWith()
// with and without parameter. Passing an object to the
// function opens a new WITH OBJECT context. Note that the
// new "with" object must be obtained with function HB_QWith()
// when the new context is opened with the HB_SetWith() function
// instead of the WITH OBJECT statement.


   PROCEDURE Main()

      WITH OBJECT ErrorNew()
         ? :className()                // result: ERROR
         ? HB_WithObjectCounter()      // result: 1

         // opens new WITH OBJECT context
         HB_SetWith( GetNew() )

            ? HB_QWith():className()    // result: GET
```

```
         ? HB_WithObjectCounter()    // result: 2

      // closes new WITH OBJECT context
      HB_SetWith()

      ? :className()                 // result: ERROR
      ? HB_WithObjectCounter()       // result: 1
   END

RETURN
```

HB_Shadow()

Displays a shadow around a rectangular area on the screen.

Syntax

```
HB_Shadow( <nTop>, <nLeft>, <nBottom>, <nRight>, ;
          [<nColorAttr>] ) --> NIL
```

Arguments

`<nTop> and <nLeft>`

Numeric values indicating the screen coordinates for the upper left corner of the shadow.

`<nBottom> and <nRight>`

Numeric values indicating the screen coordinates for the lower right corner of the shadow.

`<nColorAttr>`

This is an optional numeric color attribute in the range between 0 and 255 for drawing a shadow. The default attribute is 7, which corresponds to a color value of "W/N". Refer to ColorToN() for obtaining a numeric color attribute from a color string.

Return

The function HB_Shadow() displays a shadow around a rectangular area on the screen and returns always NIL.

Description

Info

See also: @...BOX, @...CLEAR, @...TO, HB_ClrArea(), Scroll(), SetColor()
Category: Screen functions, xHarbour extensions
Source: rtl\shadow.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates how to display a shadow around a box.

   #include "Box.ch"

   PROCEDURE Main
      SET COLOR TO "W+/B"
      CLS

      DispBox( 10,10,20,50, B_DOUBLE + Space(1), "W+/R" )

      HB_Shadow( 10,10,20,50 )

      DispOut( "Shadowed box" )
   RETURN
```

HB_SizeofCStructure()

Calculates the amount of memory required to store a C structure.

Syntax

```
HB_SizeofCStructure( <aTypes>, [<nAlign>] ) --> nBytes
```

Arguments

<aTypes>

The C-data types for each structure member must be specified as a one dimensional array of the same number of elements as there are structure members. #define constants must be used for *<aTypes>*. The following constants are available in the Cstruct.ch file:

Constants for C-data types

Constant	C-data type
CTYPE_CHAR	char
CTYPE_UNSIGNED_CHAR	unsigned char
CTYPE_CHAR_PTR	char *
CTYPE_UNSIGNED_CHAR_PTR	unsigned char*
CTYPE_SHORT	short
CTYPE_UNSIGNED_SHORT	unsigned short
CTYPE_SHORT_PTR	short *
CTYPE_UNSIGNED_SHORT_PTR	unsigned short *
CTYPE_INT	int
CTYPE_UNSIGNED_INT	unsigned int
CTYPE_INT_PTR	int *
CTYPE_UNSIGNED_INT_PTR	unsigned int *
CTYPE_LONG	long
CTYPE_UNSIGNED_LONG	unsigned long
CTYPE_LONG_PTR	long *
CTYPE_UNSIGNED_LONG_PTR	unsigned long *
CTYPE_FLOAT	float
CTYPE_FLOAT_PTR	float *
CTYPE_DOUBLE	double
CTYPE_DOUBLE_PTR	double *
CTYPE_VOID_PTR	void *
CTYPE_STRUCTURE	struct
CTYPE_STRUCTURE_PTR	struct *

<nAlign>

Optionally, the byte alignment for C-structure members can be specified. By default, the members are aligned at an eight byte boundary. Note that Windows API functions require a four byte alignment for structures.

Return

The function returns the memory requirement for a structure as a numeric value in bytes.

Description

Function HB_SizeOfCStructure() is used to calculate the size of a C structure in bytes. This depends on the number of bytes each C data type of structure members occupies in memory, and the byte alignment. The returned value is then used to create a character string like Replicate(Chr(0), nBytes). This character string can then be passed by reference to an API function via DllCall(). When DllCall() returns, the changed C structure string is decoded with function HB_StructureToArray().

An easier way of obtaining the size of a C structure is to declare a C structure class with typedef struct and call method :sizeOf() of a C structure object.

Info

See also: C Structure class, HB_ArrayToStructure(), HB_StructureToArray(), pragma pack()
Category: C Structure support, xHarbour extensions
Header: cstruct.ch
Source: vm\arrayshb.c
LIB: xhb.lib
DLL: xhbdll.dll

HB_String2Pointer()

Obtains the pointer for a character string.

Syntax

```
HB_String2Pointer( <cString> ) --> pPointer
```

Arguments

<cString>

This is a variable holding the character string to retrieve the pointer for.

Return

The function returns the pointer (memory address) of the passed string as a Poiner value (Valtype()=="P").

Description

The function obtains the memory address of the character string <*cString*> and returns it as a Pointer value.

Info

See also: C Structure class, HB_Pointer2String()
Category: Pointer functions, xHarbour extensions
Source: rtl\str2ptr.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example converts a string to a pointer, voids the string
// variable, and reads the string back from the pointer.

   PROCEDURE Main
      LOCAL cString := "Hello World"
      LOCAL pString := HB_String2Pointer( cString )
      LOCAL nBytes

      ? Valtype( cString ), cString  // result: C Hello World
      ? Valtype( pString ), pString  // result: P 004b100a

      nBytes  := Len( cString )
      cString := NIL

      ? Valtype( cString ), cString  // result: U NIL
      ? Valtype( pString ), pString  // result: P 004b100a

      cString := HB_Pointer2String( pString, nBytes )

      ? Valtype( cString ), cString  // result: C Hello World
      ? Valtype( pString ), pString  // result: P 004b100a
   RETURN
```

HB_StructureToArray()

Converts values contained in a binary C structure string to an array.

Syntax

```
HB_StructureToArray( <cBinary>   , ;
                     <aTypes>    , ;
                    [<nAlign>]   , ;
                    [<lRecurse>], ;
                     <aStructure> ) -->  aStructure
```

Arguments

<cBinary>

This is a character string holding the values of structure members in binary form. It is returned from HB_ArrayToStructure().

<aTypes>

The C-data types for each structure member must be specified as a one dimensional array of the same number of elements as there are structure members. #define constants must be used for *<aTypes>*. The following constants are available in the Cstruct.ch file:

Constants for C-data types

Constant	C-data type
CTYPE_CHAR	char
CTYPE_UNSIGNED_CHAR	unsigned char
CTYPE_CHAR_PTR	char *
CTYPE_UNSIGNED_CHAR_PTR	unsigned char*
CTYPE_SHORT	short
CTYPE_UNSIGNED_SHORT	unsigned short
CTYPE_SHORT_PTR	short *
CTYPE_UNSIGNED_SHORT_PTR	unsigned short *
CTYPE_INT	int
CTYPE_UNSIGNED_INT	unsigned int
CTYPE_INT_PTR	int *
CTYPE_UNSIGNED_INT_PTR	unsigned int *
CTYPE_LONG	long
CTYPE_UNSIGNED_LONG	unsigned long
CTYPE_LONG_PTR	long *
CTYPE_UNSIGNED_LONG_PTR	unsigned long *
CTYPE_FLOAT	float
CTYPE_FLOAT_PTR	float *
CTYPE_DOUBLE	double
CTYPE_DOUBLE_PTR	double *
CTYPE_VOID_PTR	void *
CTYPE_STRUCTURE	struct
CTYPE_STRUCTURE_PTR	struct *

<nAlign>

Optionally, the byte alignment for C-structure members can be specified. By default, the members are aligned at an eight byte boundary. Note that Windows API functions require a four byte alignment for structures.

`<lRecurse>`

This parameter defaults to .F. (false). When set to .T. (true) nested structures are decoded as well.

`<aStructure>`

An array of Len(*<aTypes>*) elements must be passed to receive in its elements the decoded values of memober variables.

Return

The function returns the array *<aStructure>*.

Description

HB_StructureToArray() is the reverse function of HB_ArrayToStructure(). It accepts a binary character string holding structure members and decodes them from their binary representation into native xHarbour data types. The resulting values are assigned to the elements of *<aStructure>*.

Note: conversion of structure data to/from binary is most comfortly done with declaring a C structure class using typedef struct and using the resulting C structure objects.

Info

See also:	C Structure class, HB_ArrayToStructure(), pragma pack(), (struct), typedef struct
Category:	C Structure support, xHarbour extensions
Header:	cstruct.ch
Source:	vm\arrayshb.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_SysLogClose()

Closes the log file of the operating system.

Syntax

```
HB_SysLogClose( <cAppName> ) --> lSuccess
```

Arguments

<cAppName>

This is a character string holding the name of the application which opened the system log file. It must be the same name as passed to HB_SysLogOpen().

Return

The function returns .T. (true) when the system log file is successfully closed, otherwise .F. (false) is returned.

Info

See also:	CLOSE LOG, HB_SysLogMessage(), HB_SysLogOpen()
Category:	Log functions, xHarbour extensions
Source:	rtl\hbSysLog.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_SysLogMessage()

Adds a new entry to the log file of the operating system.

Syntax

```
HB_SysLogMessage( <cMessage> , ;
                  <nPriority>, ;
                  <nSysID>     ) --> lSuccess
```

Arguments

<cMessage>

This is a character string holding the message to add to the new entry of the system log file.

<nPriority>

#define constants are available in the file HbLogDef.ch that can be used for the priority of log messages.

Log channel priorities

Constant	Value	Description
HB_LOG_CRITICAL	1	Critical log messages
HB_LOG_ERROR	2	Error log messages
HB_LOG_WARNING	3	Warning messages
HB_LOG_INFO	4	Informational messages
HB_LOG_DEBUG	5	Debug messages

<nSysID>

A numeric value specifying the numeric ID of the log nessage.

Return

The function returns .T. (true) when the log message is successfully added to the log file of the operating system, otherwise .F. (false) is returned.

Info

See also: LOG, HB_SysLogClose(), HB_SysLogOpen()
Category: Log functions, xHarbour extensions
Header: hblog.ch
Source: rtl\hbSysLog.c
LIB: xhb.lib
DLL: xhbdll.dll

HB_SysLogOpen()

Opens the log file of the operating system

Syntax

```
HB_SysLogOpen( <cAppName> ) --> lSuccess
```

Arguments

<cAppName>

This is a character string holding the name of the application which opens the system log file.

Return

The function returns .T. (true) when the system log file is successfully opened, otherwise .F. (false) is returned.

Description

HB_SysLogOpen() is part of xHarbour's log system. It is recommended to use the INIT LOG command with the SYSLOG() option for accessing the log file of the operating system.

Info

See also:	INIT LOG, HB_SysLogMessage(), HB_SysLogOpen()
Category:	Log functions, xHarbour extensions
Source:	rtl\hbSysLog.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_ThisArray()

Retrieves an array or object from its pointer.

Syntax

```
HB_ThisArray( <pID> ) --> aArray|oObject
```

Arguments

<pID>

This is a pointer to an array or object as it is returned from function HB_ArrayID().

Return

The function returns the array or object pointer to by *<pID>*.

Description

Function HB_ThisArray() is the reverse function of HB_ArrayID(). It accepts the pointer returned from HB_ArrayID() (Valtype()=="P") and returns the corresponding array (Valtype()=="A") or object (Valtype()=="P").

Info

See also:	Array(), HBObject(), HB_ArrayID()
Category:	Array functions, Debug functions, Object functions, xHarbour extensions
Source:	vm\arrayshb.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_Translate()

Converts a character string from one code page to another one.

Syntax

```
HB_Translate( <cString>        , ;
              <cCodePageID_Src>, ;
              <cCodePageID_Tgt>  ) --> cConvertedString
```

Arguments

<cString>

This is a character string to be translated from *<cCodePageID_Src>* to *<cCodePageID_Tgt>*.

<cCodePageID_Src>

This is a character string identifying the code page ID, *<cString>* is encoded with (refer to HB_SetCodePage() for code page IDs.

<cCodePageID_Tgt>

This is a character string identifying the code page ID, *<cString>* must be translated to.

Return

The function returns the converted character string. If either code page is not available, the return value is a null string ("").

Info

See also:	HB_LangSelect(), HB_SetCodePage()
Category:	Language specific, xHarbour extensions
Source:	rtl\cdpapi.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_Uncompress()

Uncompresses a compressed character string (ZIP).

Syntax

```
HB_Uncompress( <nBytes>, <cCompressed> ) --> cUncompressed

or

HB_Uncompress( <nBytes>     , ;
               <cCompressed>, ;
               <nComprLen>  , ;
              @<cString>      ) --> nError
```

Arguments

<nBytes>

This numeric value indicates the number of bytes the uncompressed string is going to have.

<cCompressed>

This is a character string holding the copressed data. It is obtained from function HB_Compress().

<nComprLen>

This numeric value indicates the number of bytes of the input string to uncompress. Use the expression Len(*<cCompressed>*) to uncompress the entire input string.

@<cString>

This is a pre-allocated character string. It must be passed by reference and receives the uncompressed data.

Return

The function returns either the uncompressed character string, or a numeric error code indicating success of the uncompression operation. See the description below.

Description

HB_Uncompress() is the reverse function of HB_Compress() and uncompressed character string holding ZIP compressed data. It is implemented in two "flavours" allowing for simple and advanced uncompression of data.

The easiest way of uncompressing a character string is by passing the number of bytes for the result string and the ZIP compressed string to HB_Uncompress(). In this case, the function returns the uncompressed data as a character string.

Alternatively, the number od bytes to use from *<cCompressed>* can be specified as *<nComprLen>*. This requires *<cString>* be passed by reference as fourth parameter, since the function returns a numeric error code in this case. The value zero indicates a successful operation. Values other than zero can be passed to HB_CompressErrorDesc() to obtain a descriptive error message.

Info

See also: HB_Compress(), HB_CreateLen8(), HB_CompressError(), HB_GetLen8()
Category: ZIP compression, xHarbour extensions
Header: HbCompress.ch
Source: rtl\hbcomprs.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example implements two functions for easiest (un)compression
// since the buffer size required for uncompression is stored in
// the compressed string using HB_CreateLen8().

   PROCEDURE Main
      LOCAL cText := MemoRead( "LargeFile.txt" )
      LOCAL cCompressed, cUncompressed

      cCompressed   := ZipCompress( cText )

      cUnCompressed := ZipUncompress( cCompressed )

      ? "Original    :", Len( cText )
      ? "Compressed  :", Len( cCompressed )
      ? "Uncompressed:", Len( cUncompressed )
      ? cUncompressed == cText
   RETURN


   // The function adds 8 bytes holding the length of the original
   // character string to the compressed string
   FUNCTION ZipCompress( cString )
      LOCAL cCompressed := HB_Compress( cString )

   RETURN HB_CreateLen8( Len( cString ) ) + cCompressed


   // The function extracts the first 8 bytes holding the length
   // of the uncompressed character string from the compressed string
   FUNCTION ZipUncompress( cCompressed )
      LOCAL nBytes := HB_GetLen8( cCompressed )

   RETURN HB_Uncompress( nBytes, SubStr( cCompressed, 9 ) )
```

HB_UUDecode()

Decodes a UUEncoded character string.

Syntax

```
HB_UUDecode( <cUUEncode> ) --> cString
```

Arguments

<cUUEncode>

This is an encoded character string previously returned from HB_UUEncode().

Return

The function returns the decoded character string.

Description

HB_UUDecode() decodes a UUEncoded character string in memory and returns the original data. Use function HB_UUDecodeFile() to decode an entire file.

Info

See also:	HB_UUDecodeFile(), HB_UUEncode(), HB_UUEncodeFile()
Category:	Encoding/Decoding, xHarbour extensions
Source:	tip\encoding\UUEncode.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_UUDecodeFile()

Decodes a UUEncoded file.

Syntax

```
HB_UUDecodeFile( ( <cUUEFile>, <cTargetFile> ) --> NIL
```

Arguments

`<cUUEFile>`

This is a character string holding the name of a UUEncoded file. It can be created by HB_UUEncodeFile().

`<cTargetFile>`

This is a character string holding the name of the file where the decoded data is written to.

Return

The return value is always NIL.

Description

HB_UUDecodeFile() reads an entire UUEncoded file, decodes it, and writes the original data into a second file.

Info

See also:	HB_UUDecode(), HB_UUEncode(), HB_UUEncodeFile()
Category:	Encoding/Decoding, xHarbour extensions
Source:	tip\encoding\UUEncode.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_UUEncode()

UUEncodes a character string.

Syntax

```
HB_UUEncode( <cString>, <nBytes> ) --> cUUEncoded
```

Arguments

`<cString>`

This is a character string to UUEncode.

`<nBytes>`

The number of bytes to encode from <*cString*> must be passed as second parameter. Use the expression Len(<*cString*>) to encode the entire string.

Return

The function returns a UUEncoded character string.

Description

HB_UUEncode() uses the UUEncode algorithm for encoding a character string.

Pass the resulting string to function HB_UUDecode() to obtain the original data.

Info

See also:	HB_UUDecode(), HB_UUDecodeFile(), HB_UUEncodeFile()
Category:	Encoding/Decoding, xHarbour extensions
Source:	tip\encoding\UUEncode.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example outlines UUEncoding and decoding.

   PROCEDURE Main
      LOCAL cString   := "xHarbour"
      LOCAL cUUEncode := HB_UUEncode( cString, Len(cString ) )

      ? cUUEncode                 // result: (>$AA<F)O=7(`rg

      ? HB_UUDecode( cUUEncode )  // result: xHarbour
   RETURN
```

HB_UUEncodeFile()

UUEncodes a file.

Syntax

```
HB_UUEncodeFile( <cFilename>, <cUUEFile> ) --> NIL
```

Arguments

`<cFilename>`

This is a character string holding the name of the file to encode.

`<cUUEFile>`

This is a character string holding the name of the file where UUEncoded data is written to.

Return

The return value is always NIL.

Description

HB_UUEncodeFile() reads an entire file, UUEncodes it, and writes the encoded data into a second file.

Info

See also:	HB_UUDecode(), HB_UUDecodeFile(), HB_UUEncode()
Category:	Encoding/Decoding, xHarbour extensions
Source:	tip\encoding\UUEncode.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example outlines UUEncoding and decoding of files.
// A PRG file is encoded in a second file which is decoded into
// a third file. The contents of the original and third file are
// identical.

   PROCEDURE Main
      LOCAL cFileIn  := "HB_UUEncode.prg"
      LOCAL cFileOut := "HB_UUEncode.prg.uue"

      HB_UUEncodeFile( cFileIn, cFileOut )

      HB_UUDecodeFile( cFileOut, "Test.txt" )

      ? Memoread( cFileIn ) == MemoRead( "Test.txt" ) // Result: .T.

   RETURN
```

HB_ValToStr()

Converts values of simple data types to character string.

Syntax

```
HB_ValToStr( <xValue> ) --> cString
```

Arguments

<xValue>

This is a value of simple data type (C, D, L, M, N, P, U).

Return

The function returns the converted value as a character string.

Description

Function HB_ValToStr() accepts a value of simple data type and converts it into a character string representing this value. When a value of complex data type is passed (Array, Code block, Hash or Object) the function returns a null string ("").

Info

See also:	CStr(), HB_Serialize(), ValToPrg(), ValToPrgExp(), Valtype()
Category:	Conversion functions, xHarbour extensions
Source:	rtl\valtostr.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_VMExecute()

Executes a PCode string.

Syntax

```
HB_VMExecute( <cPCode> ) --> xValue
```

Arguments

<cPCode>

This is a character string holding the PCode sequence(s) to execute. It must be created with function HB_MacroCompile().

Return

The function returns the result of the last PCode instruction of *<cPCode>*.

Description

Function HB_VMExecute() passes a character string of PCode instructions to xHarbour's virtual machine which executes the PCode. The string *<cPCode>* must be created with HB_MacroCompile(). The return value depends on the last PCode instruction within *<cPCode>*.

Info

See also:	& (macro operator), Eval(), HB_MacroCompile()
Category:	Indirect execution, xHarbour extensions
Source:	vm\hvm.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_VMMode()

Indicates the creation mode of the xHarbour virtual machine.

Syntax

```
HB_VMMode() --> nCreationMode
```

Return

The function returns a numeric value.

Description

Function HB_VMMode() exists for informational purposes only. It returns a numeric value indicating the creation mode of xHarbour's virtual machine. The following return values are possible:

Modes of the Virtual Machine

Value	Description
0 *)	Regular creation
1	Optimized for console applications
2	Optimized for GUI applications
**) default*	

Info

See also:	HB_BuildInfo(), HB_Compiler(), Version()
Category:	Environment functions, xHarbour extensions
Source:	vm\hvm.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HB_WithObjectCounter()

Determines the nesting level of WITH OBJECT statements.

Syntax

```
HB_WithObjectCounter() --> nNestingLevel
```

Return

The function returns a numeric value indicating the nesting level of WITH OBJECT statements.

Description

Function HB_WithObjectCounter() is used to determine the nesting level of the current WITH OBJECT context. A context can be opened with the WITH OBJECT statement or the HB_SetWith() function.

Info

See also: HB_SetWith(), HB_QWith(), WITH OBJECT
Category: Object functions, xHarbour extensions
Source: vm\hvm.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example opens two WITH OBJECT contexts and displays
// the result of HB_WithObjectCounter().

   PROCEDURE Main()

      WITH OBJECT ErrorNew()
         ? :className()              // result: ERROR
         ? HB_WithObjectCounter()    // result: 1


         WITH OBJECT GetNew()

            ? :className()           // result: GET
            ? HB_WithObjectCounter() // result: 2

         END

         ? :className()              // result: ERROR
         ? HB_WithObjectCounter()    // result: 1
      END

      ? HB_WithObjectCounter()       // result: 0
   RETURN
```

HB_WriteIni()

Creates an INI file and writes hash data to it.

Syntax

```
HB_WriteIni( <cFileName>      , ;
             <hIniData>       , ;
             [<cCommentBegin>], ;
             [<cCommentEnd>]  , ;
             [<lAutoMain>]      ) --> lSuccess
```

Arguments

<cFileName>

This is a character string holding the name of the INI file to create. It must include path and file extension. If the path is omitted from *<cFileName>*, the file is created in the current directory.

<hIniData>

This is a two-dimensional Hash() holding the section data and key/value pairs to store in the INI file.

<cCommentBegin>

This is an optional a character string which is written as a comment at the beginning of the INI file.

<cCommentEnd>

This is an optional a character string which is written as a comment at the end of the INI file.

<lAutoMain>

The parameter defaults to .T. (true), causing all entries in the "MAIN" hash key of *<cIniData>* be written at the beginning of the INI file without a [section] entry. When set to .F. (false), all data of *<hIniData>* are divided into [sections], including a possible the [MAIN] section.

Return

The function returns a logical value indicating a successful operation.

Description

Function HB_WriteIni() writes the data held in *<hIniData>* to the INI file *<cFileName>*. If the file exists already, it is overwritten. When the file cannot be created, the return value is .F. (false) and function FError() can be called to identify the cause of failure.

INI files provide a convenient way for storing configuration data of an application. They are divided into named [sections] holding key/value pairs. An example for an INI file is this:

```
; Start comment
Desc = Application program
Ver = Version 2.2

[Files]
DB1=Customer.dbf
DB2=Sales.dbf

[Path]
CONFIG=\apps\conf\
DATA=\apps\data\
; End comment
```

This INI file contains six key/value pairs, divided into three [sections]. The first section is unnamed. It has the default name "MAIN".

Info

See also: FCreate(), FWrite(), Hash(), HB_SetIniComment(), HB_ReadIni()
Category: File functions, xHarbour extensions
Source: rtl\hbini.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates how an INI file can be created
// programmatically by creating a two-dimensional Hash. The
// INI file shown in the function description is created by
// the example.

   PROCEDURE Main
      LOCAL hIni := Hash()

      hIni["MAIN" ]            := Hash()
      hIni["MAIN" ]["Desc" ]   := "Application program"
      hIni["MAIN" ]["Ver"  ]   := "Version 2.2"

      hIni["Path" ]            := Hash()
      hIni["Path" ]["DATA"]    := "\apps\data\"
      hIni["Path" ]["CONFIG"]  := "\apps\conf\"

      hIni["Files"]            := Hash()
      hIni["Files"]["DB1"]     := "Customer.dbf"
      hIni["Files"]["DB2"]     := "Sales.dbf"

      HB_WriteIni( "Application.ini", hIni, ;
                   "; Start comment", "; End comment" )

   RETURN
```

HB_XmlErrorDesc()

Retrieves a textual error description for XML file parsing errors.

Syntax

```
HB_XmlErrorDesc( <nErrorCode> ) --> cErrorMessage
```

Arguments

<nErrorCode>

This is the numeric error code of a TXmlDocument() object after reading an XML file. The error code is stored in the :nError instance variable.

Return

The function returns a character string holding the error message.

Description

HB_XmlErrorDesc() is a helper function provided for debugging purposes. It retrieves a textual description for an error that may occur when a TXmlDocument() processes XML data.

Info

See also:	TXmlDocument()
Category:	Debug functions, xHarbour extensions
Source:	rtl\hbxml.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HClone()

Creates an entire copy of a hash.

Syntax

```
HClone( <hHash> ) --> hClone
```

Arguments

<hHash>

A variable referencing the hash to copy.

Return

The function returns a new Hash reference containing the same key/value pairs as <*hHash*>.

Description

This function creates a new Hash value and populates it with key/value pairs of the original hash. The new hash receives copies of the original values when they are of simple data types Character, Date, Logic, Numeric and NIL.

When data types are represented by references, the reference is copied, so that the new hash holds the same (identical) value as the original. This applies to data types Array, Code block, Hash and Object.

The attributes for case sensitivity and automatic element creation of <*hHash*> is transferred to the new hash as well.

Important: <*hHash*> must not reference itself in any of its key/value pairs.

Info

See also:	Hash(), HCopy(), HGet(), HGetAutoAdd(), HDel(), HEval(), HSet(), HSetCaseMatch()
Category:	Hash functions, xHarbour extensions
Source:	vm\hash.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates how the values of key/value pairs
// of a hash are copied to a new hash

   PROCEDURE Main
      LOCAL hHash := {=>}
      LOCAL hClone

      ? HGetCaseMatch( hHash )         // result: .T.

      HSetCaseMatch( hHash, .F. )

      // create a hash with all data types
      hHash:A := { 1, 2 }
      hHash:B := {|x| QOut(x) }
      hHash:C := "Hello World"
      hHash:D := Date()
      hHash:H := Hash( "OPT1", 10, "OPT2", 20 )
      hHash:L := .T.
      hHash:N := 12345
      hHash:O := GetNew()
```

```
   hClone  := HCLone( hHash )

   // clone is not identical with original
   ? hClone == hHash                // result: .F.
   ? HGetCaseMatch( hClone )        // result: .T.

   // clone holds same array reference
   ? ValToPrg( hClone:A )           // result: { 1, 2 }
   AAdd( hHash:A, 3 )
   ? ValToPrg( hClone:A )           // result: { 1, 2, 3 }

   // All values are identical
   ? "A", hClone:A == hHash:A       // result: .T.
   ? "B", hClone:B == hHash:B       // result: .T.
   ? "C", hClone:C == hHash:C       // result: .T.
   ? "D", hClone:D == hHash:D       // result: .T.
   ? "H", hClone:H == hHash:H       // result: .T.
   ? "L", hClone:L == hHash:L       // result: .T.
   ? "N", hClone:N == hHash:N       // result: .T.
   ? "O", hClone:O == hHash:O       // result: .T.
RETURN
```

HCopy()

Copies key/value pairs from a hash into another hash.

Syntax

```
HCopy( <hSource>, ;
        <hTarget>, ;
       [<nStart>], ;
       [<nCount>], ;
       [<xMode>]   ) --> hTarget
```

Arguments

`<hSource>`

A variable referencing the hash to copy.

`<hTarget>`

A variable referencing the hash that receives key/value pairs from *<hSource>*.

`<nStart>`

A numeric value indicating the first key/value pair of *<hSource>* to copy to *<hTarget>*. It defaults to 1.

`<nCount>`

A numeric value specifying the number of key/value pairs to copy, starting from position *<nStart>*. If *<nCount>* is omitted, all elements of *<hSource>* starting at *<nStart>* are copied.

`<xMode>`

<xMode> is either a numeric value or a code block. It specifies how to treat duplicate keys in both hashes. See description below. The default value is 0.

Return

The function returns a reference to *<hTarget>*.

Description

The function copies key/value pairs from a source to a target hash. By default, all key/value pairs are copied. The first key/value pair and the number of them can optionally be specified with *<nStart>* and *<nCount>*.

Since keys in a hash must be unique, the *<xMode>* parameter defines how to treat keys that are present in both hashes. The following numeric values are valid for *<xMode>*.

Copy modes for HCopy()

Mode	Value	Description
Normal copy *)	0	When a key of *<hSource>* is present in *<hTarget>*, the value of *<hSource>* is copied to *<hTarget>*. Otherwise, the key/value pair of *<hSource>* is added to *<hTarget>*.
XOR copy	1	When a key of *<hSource>* is present in *<hTarget>*, the key/value pair is removed from *<hTarge>*, otherwise it is added to *<hTarget>*.
AND copy	2	First, all keys in *<hTarget>* are removed when they are not present in *<hSource>*. Then, only values of matching keys in *<hSource>* are copied to *<hTarget>*. Other elements of *<hSource>* are not copied.
NOT copy	3	All keys in *<hTarget>* are removed when they are present in *<hSource>*.

No other key/value pair is added or changed.

) default mode

As an alternative, a code block can be specified for <*xMode*> which allows to define even more sophisticated copy rules. The code block receives three parameters: the key, the value and the numeric ordinal position of the current key/value pair in <*hSource*>. The code block must return a logical value. When it returns .T. (true), the key/value pair is copied to <*hTarget*> in Normal copy mode. Otherwise, the key/value pair is ignored.

Notes: when the built in copy rules do not meet your requirements, use a code block for <*xMode*> that returns always .F. (false), and alter <*hTarget*> within the code block.

HCopy() does not return a new copy of the <*hTarget*> hash, but alters it. If you need to preserve the original target hash, create a new reference with HClone() first.

Info

See also: Hash(), HClone(), HDel(), HEval(), HGet(), HMerge(), HSet()
Category: Hash functions, xHarbour extensions
Source: vm\hash.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates various copy modes of HCopy()

   PROCEDURE Main
      LOCAL hHash := Hash( "b", 20, "e", 30 )
      LOCAL hSrc, hTgt

      hSrc := Hash( "a", 1, "b", 2, "c", 3, "d", 4 )
      hTgt := HClone( hHash )

      HCopy( hSrc, hTgt )
      ? ValToPrg( hTgt )
      // result: { "a" => 1, "b" => 2, "c" => 3, "d" => 4, "e" => 30 }

      hTgt := HClone( hHash )
      HCopy( hSrc, hTgt,,, 1 )
      ? ValToPrg( hTgt )
      // result: { "a" => 1, "c" => 3, "d" => 4 }

      hTgt := HClone( hHash )
      HCopy( hSrc, hTgt,,, 2 )
      ? ValToPrg( hTgt )
      // result: { "b" => 2 }

      hTgt := HClone( hHash )
      HCopy( hSrc, hTgt,,, 3 )
      ? ValToPrg( hTgt )
      // result: { "e" => 30 }

      hTgt := HClone( hHash )
      HCopy( hSrc, hTgt,,, {|k,v,i| i > 1 .AND. v % 2 == 0 } )
      ? ValToPrg( hTgt )
      // result: { "b" => 2, "d" => 4, "e" => 30 }
   RETURN
```

HDel()

Removes a key/value pair from the hash by its key.

Syntax

```
HDel( <hHash>, <xKey> ) --> NIL
```

Arguments

<hHash>

A variable referencing the hash to remove a key/value pair from.

<xKey>

<*xKey*> is the key to delete from the hash.

Return

The function returns always NIL.

Description

The function removes a key/value pair from the hash <*hHash*>. If the key <*xKey*> is not found in the hash, a runtime error is raised. Use function HHasKey() to determine the existence of a key.

Note: deleting a key is the only way to change a key in a hash. If the value must be preserved, use function HGet() to retrieve the value, then remove the key, and finally add the new key with the preserved value.

Info

See also:	Hash(), HDelAt(), HGet(), HHasKey(), HSet()
Category:	Hash functions, xHarbour extensions
Source:	vm\hash.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The function deletes a key/value pair from a hash by key.

   PROCEDURE Main
      LOCAL hHash := Hash( "A",10, "B",20 , "C",30 , "D",40 )

      HDel( hHash, "C" )

      ? ValToPrg( hHash )
      // result: { "A" => 10, "B" => 20, "D" => 40 }
   RETURN
```

HDelAt()

Removes a key/value pair from the hash by its ordinal position.

Syntax

```
HDelAt( <hHash>, <nPos> ) --> NIL
```

Arguments

<hHash>

A variable referencing the hash to remove a key/value pair from.

<nPos>

A numeric value specifying the ordinal position of the key/value pair to remove. It must be in the range between 1 and Len(*<hHash>*).

Return

The function returns always NIL.

Description

The function removes a key/value pair from the hash *<hHash>* by its ordinal position. If *<nPos>* is outside the valid range, a runtime error is raised. Use function HGetPos() to determine the ordinal position of a key.

Info

See also:	Hash(), HDel(), HGet(), HGetPos(), HSet()
Category:	Hash functions, xHarbour extensions
Source:	vm\hash.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The function deletes a key/value pair from a hash
// by ordinal position.

   PROCEDURE Main
      LOCAL hHash := Hash( "A",10, "B",20 , "C",30 , "D",40 )

      HDelAt( hHash, 3 )

      ? ValToPrg( hHash )
      // result: { "A" => 10, "B" => 20, "D" => 40 }
   RETURN
```

Header()

Returns the size of the database file header.

Syntax

```
Header() --> nBytes
```

Return

The function returns the number of bytes occupied by the file header of a database open in a work area, or zero if the work area is unused.

Description

The database function Header() is used in conjunction with LastRec() and RecSize() to compute the size of a database file.

Info

See also: DbInfo(), DiskSpace(), LastRec(), RecSize()
Category: Database functions
Source: rdd\dbcmd.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example implements a user-defined function that returns the
// file size of a database.

   PROCEDURE Main

      USE Customer NEW
      ? DbfSize()
      USE
   RETURN

   FUNCTION DbfSize()
   RETURN ( (RecSize() * LastRec()) + Header() + 1 )
```

HEval()

Evaluates a code block with each hash element.

Syntax

```
HEval( <hHash>, <bBlock>, [<nStart>], [<nCount>] ) --> hHash
```

Arguments

<hHash>

A variable referencing the hash to process.

<bBlock>

This is the codeblock that is evaluated with each key/value pair of *<hHash>*

<nStart>

A numeric value indicating the first key/value pair of *<hHash>* to evaluate the code block with. It defaults to 1.

<nCount>

<nCount> is the nubmer of key/value pairs that are passed to *<bBlock>*. If omitted, *<nCount>* is calculated as 1+Len(*<hHash>*)-*<nStart>*.

Return

The function returns a reference to *<hHash>*.

Description

The function evaluates the code block *<bBlock>* with the key/value pairs of *<hHash>*. The code block receives three parameters: the key, the value and the numeric ordinal position of the current key/value pair in *<hHash>*. The return value of the code block is ignored.

Info

See also:	AEval(), Hash(), HGetKeys(), HGetValues(), HScan()
Category:	Code block functions, Hash functions, xHarbour extensions
Source:	vm\hash.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The function illustrates the parameters passed to the
// HEval() code block.

   PROCEDURE Main
      LOCAL hHash := Hash( "A",10, "B",20 , "C",30 , "D",40 )

      HEval( hHash, {|k,v,i| QOut( k, v , i ) } )

      // Output:
      // A         10          1
      // B         20          2
      // C         30          3
      // D         40          4
   RETURN
```

HexToNum()

Converts a Hex string to a numeric value.

Syntax

```
HexToNum( <cHexString> ) --> nNumber
```

Arguments

<cHexString>

This is a character string holding an integer number in hexadecimal format.

Return

The function returns the decoded Hex string as a numeric value.

Description

HexToNum() converts a Hex formatted character string to a numeric value. It converts only characters in the range from "0-9" and "a-f" or "A-F". If *<cHexString>* contains any other character, the conversion is stopped. The reverse function is NumToHex().

Info

See also: CStr(), HexToStr(), NumToHex(), StrToHex(), StrZero(), Transform()
Category: Numeric functions, Conversion functions, xHarbour extensions
Source: rtl\hbhex2n.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates return values of HexToNum()

   PROCEDURE Main
      ? HexToNum( "1" )                 // result:         1

      ? HexToNum( "FF" )                // result:       255
      ? HexToNum( "FFFF" )              // result:     65535

      ? HexToNum( "FFFFFFFFFFFFFFFF" )  // result:        -1
      ? HexToNum( "ffffffffffffff80" )  // result:      -128
   RETURN
```

HexToStr()

Converts a Hex encoded character string to an ASCII string.

Syntax

```
HexToStr( <cHexString> ) --> cASCIIstring
```

Arguments

<cHexString>

This is a character string holding ASCII values in hexadecimal format.

Return

The function returns the decoded Hex string as a character string in ASCII format.

Description

HexToStr() converts a Hex formatted character string to a character string in ASCII format. It converts only characters in the range from "0-9" and "a-f" or "A-F". If <*cHexString*> contains any other character, the conversion is stopped. The reverse function is StrToHex().

Info

See also: CStr(), HexToNum(), NumToHex(), StrToHex(), Transform()
Category: Character functions, Conversion functions, xHarbour extensions
Source: rtl\hbhex2n.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates return values of HexToStr()

    PROCEDURE Main
       LOCAL cString1 := "48656C6C6F"
       LOCAL cString2 := "576F726C64"

       ? HexToSTr( cString1 )   // result: Hello

       ? HexToSTr( cString2 )   // result: World
    RETURN
```

HFill()

Copies the same value into all key/value pairs.

Syntax

```
HFILL( <Hash>, <xValue> ) --> NIL
```

Arguments

<Hash>

A variable referencing the hash to fill with a value.

<xValue>

The value to copy to all key/value pairs of *<hHash>*

Return

The function returns always NIL.

Description

The function uses a single value *<xValue>* and copies it to all key/value pairs of a hash. This is useful when a unified state must be achieved for all items in the hash. Use only simple data types for *<xValue>* (C,D,L,N).

Info

See also:	Hash(), HSet(), HSetValueAt(), HScan()
Category:	Hash functions, xHarbour extensions
Source:	vm\hash.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows the effect of HFill()

   PROCEDURE Main
      LOCAL hHash := { "a"=> 10, "b" => 20, "c" => 30 }
      LOCAL aValues

      ? ValToPrg( HGetValues(hHash) )   // result: { 10, 20, 30 }

      ? HFill( hHash, "A" )             // result: NIL

      ? ValToPrg( HGetValues(hHash) )   // result: { "A", "A", "A" }
   RETURN
```

HGet()

Retrieves the value associated with a specified key.

Syntax

```
HGet( <hHash>, <xKey> ) --> xValue
```

Arguments

<hHash>

A variable referencing the hash to retrieve a value from.

<xKey>

<xKey> is the key to retrieve the associated value of.

Return

The function returns the value associated with the key *<xKey>*.

Description

The function retrieves the value *<xValue>* from the hash *<hHash>* associated with the key *<xKey>*.

If the key is not found in the hash, a runtime error is raised. Use function HHasKey() to determine the existence of a key.

Info

See also: Hash(), HGetKeyAt(), HGetPairAt(), HGetValueAt(), HHasKey(), HSet()
Category: Hash functions, xHarbour extensions
Source: vm\hash.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The function retrieves values by key.

   PROCEDURE Main
      LOCAL hHash := Hash( "A",10, "B",20 , "C",30 , "D",40 )

      ? HGet( hHash, "B" )             // result: 20

      ? HGet( hHash, "D" )             // result: 40

      ? HGet( hHash, "Z" )             // runtime error
   RETURN
```

HGetAACompatibility()

Checks if a hash is compatible with an associative array.

Syntax

```
HGetAACompatibility( <hHash> ) --> lIsAssocArray
```

Arguments

<hHash>

A variable referencing the hash to check for associative array compatibility.

Return

The function returns .T. (true) when the hash is compatible with an associative array, otherwise .F. (false).

Description

Function HGetAACompatibility() tests if a hash is compatible with an associative array. This is the case when the hash is passed to function HSetAACompatibility() and .T. (true) is specified as second parameter.

Info

See also:	{=>}, Hash(), HSetAACompatibility()
Category:	Associative arrays, Hash functions, xHarbour extensions
Source:	vm\hash.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HGetAutoAdd()

Retrieves the AutoAdd attribute of a hash.

Syntax

```
HGetAutoAdd( <hHash> ) --> lIsAutoAdd
```

Arguments

<hHash>

A variable referencing the hash to retrieve the attribute from.

Return

The function returns the AutoAdd attribute of *<hHash>* as a logical value.

Description

The function retrieves an attribute of a hash that identifies whether or not automatic creation of key/value pairs is permitted in the context of inline assignments or the HSet() function.

By default, this attribute is .T. (true) when a new hash is created. This causes the assignment of a value to a non-existent key to create a new key/value pair in the hash. This default behaviour can be switched off using function HSetAutoAdd().

Info

See also:	Hash(), HGetCaseMatch(), HSet(), HSetAutoAdd(), HSetCaseMatch()
Category:	Hash functions, xHarbour extensions
Source:	vm\hash.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates the automatic creation of key/value
// pairs when a value is assigned to a non-existent key, and how
// to prevent this.

   PROCEDURE Main
      // create an empty hash
      LOCAL hHash := {=>}

      // Populating the hash:
      hHash[ 'first key'  ] := 'first value'
      hHash[ 'second key' ] := 'second value'
      hHash[ 'third key'  ] := 'third value'

      ? Len( hHash )                          // result: 3

      HSetAutoAdd( hHash, .F. )
      hHash[ 'fourth key' ] := 'fourth value'    // runtime error
   RETURN
```

HGetCaseMatch()

Retrieves the case sensitivity attribute of a hash.

Syntax

```
HGetCaseMatch( <hHash> ) --> lIsCaseSensitive
```

Arguments

<hHash>

A variable referencing the hash to retrieve the attribute from.

Return

The function returns the case sensitivity attribute of *<hHash>* as a logical value.

Description

The function retrieves an attribute of a hash that identifies whether or not character string matching for keys is case sensitive. By default, this attribute is .T. (true) when a new hash is created. This causes keys consisting of character strings that differ only in case can be assigned different values. This default behaviour can be switched off using function HSetCaseMatch().

Info

See also:	Hash(), HGetAutoAdd(), HSetAutoAdd(), HSetCaseMatch()
Category:	Hash functions, xHarbour extensions
Source:	vm\hash.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates case sensitivity of keys in a
// new created hash and outlines the effect of changing
// the case sensitivity attribute in a populated hash.

   PROCEDURE Main
      // create an empty hash
      LOCAL hHash := {=>}

      ? HGetCaseMatch( hHash )          // result: .T.

      // Populating the hash:
      hHash[ 'KEY'  ] := 'UPPER CASE'
      hHash[ 'key'  ] := 'lower case'
      hHash[ 'Key'  ] := 'Mixed Case'

      ? Len( hHash )                    // result: 3

      ? hHash[ 'KEY' ]                  // result: UPPER CASE
      ? hHash[ 'key' ]                  // result: lower case
      ? hHash[ 'Key' ]                  // result: Mixed Case

      // Don't do this with a populated case sensitive hash
      HSetCaseMatch( hHash, .F. )

      ? hHash[ 'KEY' ]                  // result: Mixed Case
      ? hHash[ 'key' ]                  // result: Mixed Case
```

```
   ? hHash[ 'Key' ]                  // result: Mixed Case
RETURN
```

HGetKeyAt()

Retrieves the key from a hash by its ordinal position.

Syntax

```
HGetKeyAt( <hHash>, <nPos> ) --> xKey
```

Arguments

<hHash>

A variable referencing the hash to retrieve a key from.

<nPos>

A numeric value specifying the ordinal position of the key/value pair to query. It must be in the range between 1 and Len(*<hHash>*).

Return

The function returns the key at position *<nPos>* in the hash *<hHash>*.

Description

This function retrieves the key from the hash *<hHash>* at position *<nPos>*. If *<nPos>* is outside the valid range, a runtime error is raised. Use function HGetPos() to determine the ordinal position of a key.

The keys are inserted into the hash by their sorting order and cannot be moved or changed.

Info

See also: Hash(), HDelAt(), HGet(), HGetPairAt(), HGetKeys(), HGetPos(), HGetValueAt()
Category: Hash functions, xHarbour extensions
Source: vm\hash.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The function retrieves different keys by ordinal position.
// Note that the creation order of key/value pairs does not
// affect their insertion order (which is A B C D).

   PROCEDURE Main
      LOCAL hHash := Hash( "C", 10, "D", 30, "A", 40, "B", 20 )

      ? HGetKeyAt( hHash, 3 )       // result: C

      ? HGetKeyAt( hHash, 1 )       // result: A

   RETURN
```

HGetKeys()

Collects all keys from a hash in an array.

Syntax

```
HGetKeys( <hHash> ) --> aKeys
```

Arguments

<hHash>

A variable referencing the hash to retrieve all keys from.

Return

The function returns a one-dimensional array holding in its elements copies of the keys of *<hHash>*.

Description

This function retrieves all keys from a hash and collects them in an array. The ordinal position of a key in the returned array is identical with the position in the hash.

Info

See also:	Hash(), HGetKeyAt(), HGetValues()
Category:	Hash functions, xHarbour extensions
Source:	vm\hash.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The function retrieves all keys of a hash.
// Note that the creation order of key/value pairs does not
// affect their insertion order (which is A B C D).

   PROCEDURE Main
      LOCAL hHash := Hash( "C", 10, "D", 30, "A", 40, "B", 20 )
      LOCAL aKeys := HGetKeys( hHash )

      ? aKeys[1]                    // result: A
      ? aKeys[3]                    // result: C
      ? HGetKeyAt( hHash, 1 )       // result: A
      ? HGetKeyAt( hHash, 3 )       // result: C
   RETURN
```

HGetPairAt()

Retrieves a key/value pair from a hash by its ordinal position.

Syntax

```
HGetPairAt( <hHash>, <nPos> ) --> { xKey, xValue }
```

Arguments

<hHash>

A variable referencing the hash to retrieve a key/value pair from.

<nPos>

A numeric value specifying the ordinal position of the key/value pair to retrieve. It must be in the range between 1 and Len(*<hHash>*).

Return

The function returns a two-element array with the key and value found at position *<nPos>* in the hash *<hHash>*.

Description

This function retrieves the key/value pair from the hash *<hHash>* at position *<nPos>*. If *<nPos>* is outside the valid range, a runtime error is raised. Use function HGetPos() to determine the ordinal position of a key.

Info

See also:	Hash(), HDelAt(), HGet(), HGetKeyAt(), HGetPos(), HGetValueAt()
Category:	Hash functions, xHarbour extensions
Source:	vm\hash.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The function retrieves key/value pairs by ordinal position.
// Note that the creation order of key/value pairs does not
// affect their insertion order (which is A B C D).

   PROCEDURE Main
      LOCAL hHash  := Hash( "C", 10, "D", 30, "A", 40, "B", 20 )
      LOCAL aPair1 := HGetPairAt( hHash, 2 )
      LOCAL aPair2 := HGetPairAt( hHash, 4 )

      ? aPair1[1], aPair1[2]          // result: B 20
      ? aPair2[1], aPair2[2]          // result: D 30
   RETURN
```

HGetPartition()

Checks if a hash is partitioned.

Syntax

```
HGetPartition( <hHash>     , ;
              [@<nPageSize>], ;
              [@<nLevel>]     ) --> lIsPartitioned
```

Arguments

`<hHash>`

A variable referencing the hash to check for partitioning.

`<nPageSize>`

If passed by reference, *<nPageSize>* is assigned a numeric value indicating the size of a partition.

`<nLevel>`

If passed by reference, *<nLevel>* is assigned a numeric value indicating how many levels deep a hash is partitioned.

Return

The function returns .T. (true) when a hash is partitioned, otherwise .F. (false) is returned.

Description

The function serves informational purposes only and checks if a hash is partitioned with function HSetPartition().

Info

See also:	Hash(), HSetPartition()
Category:	Hash functions, xHarbour extensions
Source:	vm\hash.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HGetPos()

Retrieves the ordinal position of a key in a hash.

Syntax

```
HGetPos( <hHash>, <xKey>) --> nPos
```

Arguments

<hHash>

A variable referencing the hash to search a key in.

<xKey>

This is the key to search for in *<hHash>*.

Return

The function returns the ordinal position of *<xKey>* in the hash *<hHash>*, or 0 if the key is not present in the hash.

Description

This function is mostly used to check if a key is present in a hash. If the return value is greater than zero, the key exists. The return value can then be passed on to function HGetValueAt() to retrieve the associated value. This is more efficient than using the key a second time for retrieving the value.

Info

See also:	Hash(), HDelAt(), HGetKeys(), HGetPairAt(), HGetValueAt(), HHasKey()
Category:	Hash functions, xHarbour extensions
Source:	vm\hash.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The function first checks the existence of a key and then
// retrieves the associated value by ordinal position.
// Note that the creation order of key/value pairs does not
// affect their insertion order (which is A B C D).

   PROCEDURE Main
      LOCAL hHash := Hash( "C", 10, "D", 30, "A", 40, "B", 20 )
      LOCAL aKeys := { "C", "D", "E" }
      LOCAL i, nPos

      FOR i:=1 TO Len( aKeys )
          nPos := HGetPos( hHash, aKeys[i] )
          IF nPos > 0
             ? aKey[i], HGetValuAt( hHash, i )
          ELSE
             ? aKey[i], "not present"
          ENDIF
      NEXT
      ** Output:
      // C         40
      // D         20
      // E not present
   RETURN
```

HGetVaaPos()

Retrieves the sort order of all keys in an associative array.

Syntax

```
HGetVaaPos( <hArray> ) --> aSortOrder
```

Arguments

<hArray>

A variable referencing an associative array.

Return

The function returns a one dimensional array holding in its elements the numeric sort order of each key in *<hArray>*.

Description

Function HGetVaaPos() iterates an associative array and collects the sort order of each key in a regular array, which is returned. This is equivalent to calling HaaGetRealPos() for all elements of *<hArray>*.

Info

See also:	Array(), HaaGetKeyAt(), HaaGetPos(), HaaGetRealPos(), HaaGetValueAt(), HaaSetValueAt(), Hash(), HSetAACompatibility()
Category:	Associative arrays, Hash functions, xHarbour extensions
Source:	vm\hash.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays the ordinal position and the
// sort order of keys in an associative array.

   PROCEDURE Main
      LOCAL hArray := Hash(), aSortOrder, i

      HSetAACompatibility( hArray, .T. )

      hArray[ "One"   ] := 10
      hArray[ "Two"   ] := 20
      hArray[ "Three"] := 30
      hArray[ "Four" ] := 40
      hArray[ "Five" ] := 50

      aSortOrder := HGetVAAPos( hArray )

      FOR i :=1 TO Len( aSortOrder )
         ? i, aSortOrder[i]
      NEXT

      ** Output:
      //      1       3
      //      2       5
      //      3       4
      //      4       2
      //      5       1
   RETURN
```

HGetValueAt()

Retrieves the value from a hash by its ordinal position.

Syntax

```
HGetValueAt( <hHash>, <nPos> ) --> xValue
```

Arguments

<hHash>

A variable referencing the hash to retrieve a value from.

<nPos>

A numeric value specifying the ordinal position of the value to retrieve. It must be in the range between 1 and Len(*<hHash>*).

Return

The function returns the value at position *<nPos>* in the hash *<hHash>*.

Description

This function retrieves the value from the hash *<hHash>* at position *<nPos>*. If *<nPos>* is outside the valid range, a runtime error is raised. Use function HGetPos() to determine the ordinal position of a key/value pair.

Values of key/value pairs can be changed by specifying the key for a hash and assigning a new value, or by using function HSetValueAt() which accepts the ordinal position of the value to change.

Info

See also:	Hash(), HDelAt(), HGet(), HGetKeyAt(), HGetPairAt(), HGetPos(), HSetValueAt()
Category:	Hash functions, xHarbour extensions
Source:	vm\hash.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The function retrieves different values by ordinal position.
// Note that the creation order of key/value pairs does not
// affect their insertion order (which is A B C D).

   PROCEDURE Main
      LOCAL hHash := Hash( "C", 10, "D", 30, "A", 40, "B", 20 )

      ? HGetValueAt( hHash, 3 )       // result: 10

      ? HGetValueAt( hHash, 1 )       // result: 40

      hHash[ "C" ] := 123
      ? HGetValueAt( hHash, 3 )       // result: 123
   RETURN
```

HGetValues()

Collects all values from a hash in an array.

Syntax

```
HGetValues( <hHash> ) --> aValues
```

Arguments

<hHash>

A variable referencing the hash to retrieve all values from.

Return

The function returns a one-dimensional array holding in its elements the values of *<hHash>*.

Description

This function retrieves all values from a hash and collects them in an array. The ordinal position of a value in the returned array is identical with the position in the hash.

Info

See also: Hash(), HGetKeys(), HGetValueAt()
Category: Hash functions, xHarbour extensions
Source: vm\hash.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The function retrieves all keys of a hash.
// Note that the creation order of key/value pairs does not
// affect their insertion order (which is A B C D).

   PROCEDURE Main
      LOCAL hHash := Hash( "C", 10, "D", 30, "A", 40, "B", 20 )
      LOCAL aValues := HGetValues( hHash )

      ? aValues[1]                   // result: 40
      ? aValues[3]                   // result: 10

      ? HGetValueAt( hHash, 1 )      // result: 40
      ? HGetValueAt( hHash, 3 )      // result: 10
   RETURN
```

HHasKey()

Determines if a key is present in a hash.

Syntax

```
HHasKey( <hHash>, <xKey> ) --> lExists
```

Arguments

<hHash>

<hHash> is identifier of the hash to test for the presence of a key.

<xKey>

This is the key to search for in *<hHash>*.

Return

The function returns .T. (true) if the *<xKey>* exists in *<hHash>*, otherwise .F. (false) is returned.

Description

This function tests if a key is present in a hash. HHasKey() is slightly more efficient than HGetPos(). It can be used for the same purpose but returns the numeric key position.

Info

See also:	Hash(), HGetKeys(), HGetPos(), HGetValues()
Category:	Hash functions, xHarbour extensions
Source:	vm\hash.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates how a hash can be created for language
// dependent versions of a program.

   PROCEDURE Main
      LOCAL hHash := Hash()
      LOCAL cLang := "German"

      hHash[ cLang     ] := NIL
      hHash[ "Message" ] := ""

      IF HHasKey( hHash, "English" )
         hHash[ "Message" ] := "Hello World"
      ELSEIF HHasKey( hHash, "German" )
         hHash[ "Message" ] := "Hallo Welt"
      ENDIF

      ? hHash[ "Message" ]
   RETURN
```

HMerge()

Merges the contents of an entire hash into another hash.

Syntax

```
HMerge( <hTarget>, <hSource>, [<xMode>]) --> hTarget
```

Arguments

`<hTarget>`

A variable referencing the hash that receives key/value pairs from <*hSource*>.

`<hSource>`

A variable referencing the hash to merge into <*hTarget*>.

`<xMode>`

<*xMode*> is either a numeric value or a code block. It specifies how to treat duplicate keys in both hashes. The default value is 0.

Return

The function returns a reference to <*hTarget*>.

Description

This function merges the entire contents of the hash <*hSource*> into the hash <*hTarget*>, depending on the value of <*xMode*>.

This function is a shortcut for:

```
HCopy( <hSource>, <hTarget>, 1, Len(<hSource>), <xMode> )
```

Note that <*hTarget*> and <*hSource*> are used in different order with HCopy() and HMerge().

For a description of parameter <*xMode*> refer to function HCopy().

Info

See also:	HCopy()
Category:	Hash functions, xHarbour extensions
Source:	vm\hash.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// See the example for HCopy()
```

Hour()

Extracts the hour from a DateTime value

Syntax

```
Hour( <dDateTime> ) --> nHour
```

Arguments

<dDateTime>

This is a DateTime() value.

Return

The function extracts the hour from a DateTime value and returns it as a numeric value.

Info

See also:	DateTime(), Day(), Minute(), Month(), Secs(), Year()
Category:	Conversion functions, Date and time, xHarbour extensions
Source:	rtl\dateshb.c
LIB:	xhb.lib
DLL:	xhbdll.dll

HScan()

Searches a value in a hash.

Syntax

```
HScan( <hHash>  , ;
       <xValue> , ;
      [<nStart>], ;
      [<nCount>], ;
      [<lExact>], ;
      [<lASCII>]  ) --> nPos
```

Arguments

<hHash>

A variable referencing the hash to search for a value.

<xValue>

This is either the value to search or a code block specifying the search condition.

<nStart>

A numeric value indicating the first key/value pair of *<hHash>* to begin the search with. It defaults to 1.

<nCount>

<nCount> is the nubmer of key/value pairs to be searched, beginning at *<nStart>*. If omitted, *<nCount>* is calculated as 1+Len(*<hHash>*)-*<nStart>*.

<lExact>

This is a logical value and defaults to .T. (true). It is only relevant when a character string is searched in a hash. By default, the string matching rules are applied as if SET EXACT is set to ON. Pass .F. (false) to search values as if SET EXACT is OFF.

<lASCII>

This parameter defaults to .F. (false). If .T. (true) is passed, single character strings are matched with a numeric *<xValue>* based on their ASCII value.

Return

The function returns the numeric ordinal position of *<xValue>* in *<hHash>*, or zero when the value is not found.

Description

The function is used to search a value in a hash. It accepts either the value to search, or a code block that specifies the search condition.

If *<xValue>* is a code block, it receives three parameters: the key, the value and the numeric ordinal position of the current key/value pair in *<hHash>*. The code block must return a logical value. When it returns .T. (true), the search condition is satisfied and HScan() returns the ordinal position of the current item in *<hHash>*.

Info

See also: Hash(), HEval(), HGetPos(), HGetValueAt()
Category: Hash functions, xHarbour extensions
Source: vm\hash.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates various results of HScan()
// using search values and search conditions (code blocks)

   PROCEDURE Main
      LOCAL hHash := {=>}

      hHash[ "a" ] := "Hello"
      hHash[ "b" ] := { 1, 2 }
      hHash[ "c" ] := Date()
      hHash[ "d" ] := 1
      hHash[ "e" ] := { 4, 5 }

      ? HScan( hHash, "Hello" )                  // result: 1

      ? HScan( hHash, 1       )                  // result: 4

      ? HScan( hHash, Date()  )                  // result: 3

      ? HScan( hHash, {|k,v,i| Valtype(v)=="A" .AND. v[1] == 1} )
                                                 // result: 2

      ? HScan( hHash, {|k,v,i| Valtype(v)=="A" .AND. v[1] == 4} )
                                                 // result: 5
   RETURN
```

HSet()

Associates a value with a key in a hash.

Syntax

```
HSet( <hHash>, <xKey>, <xValue> ) --> NIL
```

Arguments

<hHash>

A variable referencing the hash that receives the key/value pair.

<xKey>

The key of the key/value pair.

<xValue>

The value of the key/value pair.

Return

The function returns always NIL.

Description

HSet() is the functional equivalent of the hash access operator [ ] in conjunction with the inline assignment operator :=. It provides the same functionality for populating a hash or changing the value of existing key/value pais, but is slightly less efficient.

Info

See also: Hash(), HGet(), HGetKeyAt(), HGetPairAt(), HSetValueAt(), HHasKey()
Category: Hash functions, xHarbour extensions
Source: vm\hash.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates two posibilities of populating a hash
// and changing its values.

   PROCEDURE Main
      LOCAL hHash := {=>}

      hHash[ "A" ] := 10
      HSet( hHash, "B", 20 )

      ? ValToPrg( hHash )
      // result: { "A" => 10, "B" => 20 }

      hHash[ "B" ] := 100
      HSet( hHash, "A", 200 )

      ? ValToPrg( hHash )
      // result: { "A" => 200, "B" => 100 }
   RETURN
```

HSetAACompatibility()

Enables or disables associative array compatibility for an empty hash.

Syntax

```
HSetAACompatibility( <hHash>, <lToggle> ) --> lSuccess
```

Arguments

<hHash>

This is a variable referencing an empty hash. If the hash is not empty, a runtime error is raised.

<lToggle>

A logical value must be passed as second parameter. .T. (true) enables associative array compatibility, and .F. (false) disables it.

Return

The function returns a logical value indicating success (.T.) or failure (.F.) of the operation.

Description

The functions accepts an empty hash and toggles associative array compatibility. If this is switched on, values of a hash can be retrieved not only with the hash key, but also using their numeric ordinal position. Hash access can then be programmed using the array element operator and a numeric index.

Info

See also:	Array(), HaaGetKeyAt(), HaaGetPos(), HaaGetRealPos(), HaaGetValueAt(), HaaSetValueAt(), Hash(), HGetAACompatibility()
Category:	Associative arrays, Hash functions, xHarbour extensions
Source:	vm\hash.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example outlines the possibilities for accessing values
// of a hash when associative array compatibility is switched on.

   PROCEDURE Main
      LOCAL hArray := Hash(), aPos, nPos

      HSetAACompatibility( hArray, .T. )
      HSetCaseMatch( hArray, .F. )

      hArray[ "One"  ] := 10
      hArray[ "Two"  ] := 20
      hArray[ "Three"] := 30
      hArray[ "Four" ] := 40
      hArray[ "Five" ] := 50

      ? hArray[1]                   // result: 10
      ? hArray[2] += 100            // result: 120

      ? hArray:Three                // result: 30

      ? hArray["Four"] + hArray[5]  // result: 90
   RETURN
```

HSetAutoAdd()

Changes the AutoAdd attribute of a hash.

Syntax

```
HSetAutoAdd( <hHash>, <lOnOff> ) --> hHash
```

Arguments

<hHash>

A variable referencing the hash whose AutoAdd attribute is changed.

<lOnOff>

A logical value. .T. (true) enables automatic creation of key/value pairs, and .F. (false) disables it.

Return

The function returns a reference to *<hHash>*.

Description

The function changes an attribute of a hash that identifies whether or not automatic creation of key/value pairs is permitted in the context of inline assignments or the Hset() function.

By default, this attribute is .T. (true) when a new hash is created. This causes the assignment of a value to a non-existent key to create a new key/value pair in the hash. This default behaviour can be switched off to prevent automatic creation of new key/value pairs.

Use function HGetAutoAdd() to retrieve the current setting of this attribute.

Info

See also:	Hash(), HGetAutoAdd(), HGetCaseMatch(), HSet(), HSetCaseMatch()
Category:	Hash functions, xHarbour extensions
Source:	vm\hash.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates the automatic creation of key/value
// pairs when a value is assigned to a non-existent key, and how
// to prevent this.

   PROCEDURE Main
      // create an empty hash
      LOCAL hHash := {=>}

      // Populating the hash:
      hHash[ 'first key'  ] := 'first value'
      hHash[ 'second key' ] := 'second value'
      hHash[ 'third key'  ] := 'third value'

      ? Len( hHash )                           // result: 3

      HSetAutoAdd( hHash, .F. )
      hHash[ 'fourth key' ] := 'fourth value'    // runtime error
   RETURN
```

HSetCaseMatch()

Changes the case sensitivity attribute of a hash.

Syntax

```
HSetCaseMatch( <hHash>, <lOnOff> ) --> hHash
```

Arguments

<hHash>

A variable referencing the hash whose case sensitivity attribute is changed.

<lOnOff>

A logical value. .T. (true) enables case sensitivity of a hash, and .F. (false) disables it.

Return

The function returns a reference to *<hHash>*.

Description

The function changes an attribute of a hash that identifies whether or not character string matching for keys is case sensitive. By default, this attribute is .T. (true) when a new hash is created. This causes keys consisting of character strings that differ only in case can be assigned different values.

Use function HGetCaseMatch() to retrieve the current setting of this attribute.

Info

See also: Hash(), HGetAutoAdd(), HGetCaseMatch(), HSetAutoAdd()
Category: Hash functions, xHarbour extensions
Source: vm\hash.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates case sensitivity of keys in a
// new created hash and outlines the effect of changing
// the case sensitivity attribute in a populated hash.

   PROCEDURE Main
      // create an empty hash
      LOCAL hHash := {=>}

      ? HGetCaseMatch( hHash )          // result: .T.

      // Populating the hash:
      hHash[ 'KEY'  ] := 'UPPER CASE'
      hHash[ 'key'  ] := 'lower case'
      hHash[ 'Key'  ] := 'Mixed Case'

      ? Len( hHash )                    // result: 3

      ? hHash[ 'KEY' ]                  // result: UPPER CASE
      ? hHash[ 'key' ]                  // result: lower case
      ? hHash[ 'Key' ]                  // result: Mixed Case

      // Don't do this with a populated case sensitive hash
      HSetCaseMatch( hHash, .F. )
```

```
   ? hHash[ 'KEY' ]                    // result: Mixed Case
   ? hHash[ 'key' ]                    // result: Mixed Case
   ? hHash[ 'Key' ]                    // result: Mixed Case
RETURN
```

HSetPartition()

Partitions a linear hash for improved performance.

Syntax

```
HSetPartition( <hHash>, <nPageSize>, [<nLevel>] ) --> NIL
```

Arguments

<hHash>

A variable referencing the hash who is partitioned.

<nPageSize>

A numeric value specifying the page size of one hash partition.

<nLevel>

A numeric value indicating how many levels deep the hash should be partitioned.

Return

The function returns always NIL.

Description

The function partitions a linear hash into *<nPageSize>* numbers of items. By partitioning a linear hash, searching and inserting keys is improved. This is advantageous for hashes of more than 1000 key/value pairs. Smaller hashes do not need to be partitioned, unless they are accessed with a high frequency.

The partitioning of a hash results in a hash of hashes. Keys of a hash are distributed equally in sub-hashes so that searching or inserting a key becomes faster in a partitioned hash compared to a linear hash.

Important: this function may only be called on empty hashes. Do not pass a populated hash to it. To partition a filled hash, or repartition it, create an empty hash, set the desired partition scheme and merge the filled hash into the partitioned empty hash using HMerge().

Info

See also: Hash(), HGetPartition(), HMerge()
Category: Hash functions, xHarbour extensions
Source: vm\hash.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example employs a small test program that demonstrates
// the effect of partitioning a large hash. The key creation
// is done in a "worst case" scenario, where the key with highest
// sorting order is inserted first into the hash. The key with
// lowest sorting order is inserted last. You can vary the
// #define constants for the number of hash keys, page size
// and partition depth to see how each factor influences
// the entire performance of populating a large hash.

    #define HASH_KEYS   10000
    #define PAGE_SIZE      16   // 20 | 30 | 50
    #define PART_LEVEL      2   //  5 |  4 |  3

    PROCEDURE Main
```

```
   LOCAL hHash, cKey, nSize, nLevel
   LOCAL t1, t2, t3

   t1 := Seconds()

   // creating linear hash
   hHash := {=>}
   FOR i:=1 TO HASH_KEYS
      cKey :=  "K" + Padl( HASH_KEYS - i, 6, "0" )
      hHash[ cKey ] := 2 * i
   NEXT

   t2 := Seconds()

   // hash partitioned down to partitioning level
   hHash := {=>}
   HSetPartition( hHash, PAGE_SIZE, PART_LEVEL )

   FOR i:=1 TO HASH_KEYS
      cKey :=  "K" + Padl( HASH_KEYS - i, 6, "0" )
      hHash[ cKey ] := 2 * i
   NEXT

   t3 := Seconds()

   ? "Hash keys  :", Len( hHash )
   ? "linear hash:", 100*(t2-t1)/(t2-t1), "%"   // 100.00 %
   ? "partitioned:", 100*(t3-t2)/(t2-t1), "%"   //   1.80 %
   ? "IsPartition:", HGetPartition( hHash, @nSize, @nLevel )
   ? "Page size  :", nSize
   ? "Level      :", nLevel
RETURN
```

HSetValueAt()

Changes the value in a hash by its ordinal position.

Syntax

```
HSetValueAt( <hHash>, <nPos>, <xValue> ) --> NIL
```

Arguments

<hHash>

A variable referencing the hash to change a value in.

<nPos>

A numeric value specifying the ordinal position of the value to change. It must be in the range between 1 and Len(*<hHash>*).

<xValue>

<xValue> is the new value to assign at position *<nPos>* in hash *<hHash>*.

Return

The function returns always NIL.

Description

This function changes the value of the hash *<hHash>* at position *<nPos>*. If *<nPos>* is outside the valid range, a runtime error is raised. Use function HGetPos() to determine the ordinal position of a key/value pair.

Info

See also:	Hash(), HDelAt(), HGet(), HGetKeyAt(), HGetPairAt(), HGetPos(), HGetValueAt(), HSet()
Category:	Hash functions, xHarbour extensions
Source:	vm\hash.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The function changes values in a hash by ordinal position and by key.
// Note that the creation order of key/value pairs does not
// affect their insertion order (which is A B C D).

   PROCEDURE Main
      LOCAL hHash := Hash( "C", 10, "D", 30, "A", 40, "B", 20 )

      ? HGetValueAt( hHash, 3 )      // result: 10
      ? HGetValueAt( hHash, 1 )      // result: 40

      HSetValueAt( hHash, 3 , 123 )
      hHash[ "A" ] := 789

      ? HGetValueAt( hHash, 3 )      // result: 123
      ? HGetValueAt( hHash, 1 )      // result: 789
   RETURN
```

HS_Add()

Adds a text string entry to a HiPer-SEEK index file.

Syntax

```
HS_Add( <nHsxHandle>, <bText> ) --> nIndexEntry
```

Arguments

<nHsxHandle>

This is the numeric file handle of the HiPer-SEEK index file to add a new entry to. The file handle is returned from HS_Open() or HS_Create().

<bText>

The second parameter is a code block which returns the character string to add to the HiPer-SEEK index file.

Return

The function returns a numeric value. Positive values identify the ordinal number of the new index entry, while negative numbers are error codes:

Error codes of HS_Add()

Value	Description
-4	Error while attempting to seek during buffer flushing.
-6	Write error during buffer flushing.
-16	Invalid parameters are passed.
-18	Illegal HiPer-SEEK index file handle.

Description

Function HS_Add() is used to fill a HyPer-SEEK index file after it is created with HS_Create(), or opened with HS_Open(). The code block *<bText>* must return a character string. HS_Add() calculates an index key for this text string and adds it to the HiPer-SEEK index file.

Info

See also: HS_Delete(), HS_Replace()
Category: Database functions, HiPer-SEEK functions, Index functions, xHarbour extensions
Source: rdd\hsx\hsx.c
LIB: lib\xhb.lib
DLL: dll\xhbdll.dll

Example

```
// The example outlines the steps required to create a
// HiPer-SEEK index file and fill with information
// from a database file.

   PROCEDURE Main
      LOCAL bIndex, nHandle, nIndex

      CLS
      USE Customer
      SET DELETED OFF

      nHandle := HS_Create( "Customer.hsx", 8, 3, .F., 2 )
      bIndex  := {|| Trim(LastName)+" "+Trim(FirstName) }
```

```
   DO WHILE .NOT. Eof()
      nIndex := HS_Add( nHandle, bIndex )
      IF nIndex <> Recno()
         ? "Error adding index:", nIndex
      ENDIF
      SKIP
   ENDDO

   HS_Close( nHandle )
   USE
RETURN
```

HS_Close()

Closes a HiPer-SEEK index file.

Syntax

```
HS_Close( <nHsxHandle> ) --> nErrorCode
```

Arguments

<nHsxHandle>

This is the numeric file handle of the HiPer-SEEK index file to add a new entry to. The file handle is returned from HS_Open() or HS_Create().

Return

The function returns a numeric value. Positive 1 indicates that the file is successfully closed, while negative numbers identify an error condition.

Error codes of HS_Close()

Value	Description
-4	Error while attempting to seek during buffer flushing.
-6	Write error during buffer flushing.
-16	Invalid parameters are passed.

Description

HS_Close() closes a HiPer-SEEK index file previously opened with HS_Create() or HS_Open(). This ensures that all memory buffers used are flushed to an index file before program termination.

Info

See also: HS_Create(), HS_Open()
Category: Database functions, HiPer-SEEK functions, Index functions, xHarbour extensions
Source: rdd\hsx\hsx.c
LIB: lib\xhb.lib
DLL: dll\xhbdll.dll

HS_Create()

Creates a new HiPer-SEEK index file.

Syntax

```
HS_Create( <cFileName>        , ;
           <nBufferSize>      , ;
           <nKeySize>         , ;
           <lCaseInsensitive>, ;
           <nFilterSet>        ) --> nHsxHandle
```

Arguments

`<cFileName>`

This is a character string holding the name of the HiPer-SEEK index file to create. It must include path and file extension. If the path is omitted from *<cFileName>*, the file is created in the current directory. If no file extension is given, the extension .HSX is used.

`<nBufferSize>`

This is a numeric value specifying the memory buffer size in kB to be used by HS_*() functions for this file (the value 10 means 10240 bytes).

`<nKeySize>`

An numeric value of 1, 2 or 3 must be passed. It defines the size of a single index entry in the HiPer-SEEK index file. The larger *<nKeySize>* is, the more unique index entries can be stored, reducing the possibility of false positives when searching a HiPer-SEEK index file. The size of an index entry is 16 bytes (1), 32 bytes (2) or 64 bytes (3).

`<lCaseInsensitive>`

This is a logical value. When .T. (true) is passed, the index is case insensitive, otherwise it is case sensitive.

`<nFilterSet>`

This is a numeric value of 1 or 2 defining the filter set to use with the HiPer-SEEK index file. When *<nFilterSet>* is 1, all non-printable characters, such as Tab, or carriage return/line feed, are treated as blank spaces and the high-order bit is ignored for characters (7-bit character set).

Specifying 2 recognizes all characters by their ASCII value.

Return

The function returns a positive numeric value when the HiPer-SEEK index file is successfully created. This is the handle to the new file, which must be used with other HS_*() functions. A negative value indicates an error:

Error codes of HS_Create()

Value	Description
-1	File cannot be created by the operating system.
-2	Not enough memory or *<nBufferSize>* is too large.
-3	Write error for index file header.
-16	Invalid parameters are passed.

Description

HS_Create() creates a new HiPer-SEEK index file and writes the file header information. Index entries are not added by the function, which must be done afterwards using HS_Add(). Alternatively, a new HiPer-SEEK index file can be created and populated with function HS_Index().

HiPer-SEEK index files provide the means for fast full-text search routines, since they contain unique index keys (hash values) for text strings. The size of an index key can be choosen as 16, 32 or 64 bytes. Text strings that result in different index keys, are found very fast within a HiPer-SEEK index file.

After index keys are added to the HiPer-SEEK index file, it can be searched for records. The search is initiated by defining the search text with HS_Set(), followed by subsequent calls to HS_Next().

Note: it is possible that two different text strings can result in the same index key. For this reason, function HS_Verify() is available to verify if a found text actually matches the original text.

Info

See also: HS_Add(), HS_Index(), HS_Next(), HS_Open(), HS_Set()
Category: Database functions, HiPer-SEEK functions, Index functions, xHarbour extensions
Source: rdd\hsx\hsx.c
LIB: lib\xhb.lib
DLL: dll\xhbdll.dll

Example

```
// The example outlines the steps required to create a
// HiPer-SEEK index file and fill it with data from a
// database file.

   PROCEDURE Main
      LOCAL bIndex, nHandle, nIndex
      LOCAL nBufferSize := 8
      LOCAL nKeySize    := 3
      LOCAL lCaseInsens := .T.
      LOCAL nFilterSet  := 2

      CLS
      USE Customer
      SET INDEX TO Cust01, Cust02

      // Make sure records are in natural order and deleted
      // records are included in the HiPer-SEEK index file
      SET DELETED OFF
      SET ORDER TO 0

      nHandle := HS_Create( "Customer.hsx", ;
                            nBufferSize   , ;
                            nKeySize      , ;
                            lCaseInsens   , ;
                            nFilterSet      )
      bIndex  := {|| Trim(LastName)+" "+Trim(FirstName) }

      DO WHILE .NOT. Eof()
         nIndex := HS_Add( nHandle, bIndex )
         IF nIndex <> Recno()
            ? "Error adding index:", nIndex
         ENDIF
         SKIP
      ENDDO

      HS_Close( nHandle )
```

```
USE
RETURN
```

HS_Delete()

Marks an index entry as deleted in a HiPer-SEEK index file.

Syntax

```
HS_Delete( <nHsxHandle>, <nIndexEntry> ) --> nErrorCode
```

Arguments

`<nHsxHandle>`

This is the numeric file handle of the HiPer-SEEK index file to mark an entry as deleted. The file handle is returned from HS_Open() or HS_Create().

`<nIndexEntry>`

The ordinal position of the index entry to delete must be specified as a numeric value.

Return

The function returns 1 when the index entry is successfully marked as deleted, or a negative numeber indicating an error condition:

Error codes of HS_Delete()

Value	Description
-4	Error while attempting to seek during buffer flushing.
-5	Read error while reading.
-6	Write error during buffer flushing.
-7	*<nIndexEntry>* is out of bounds.
-8	*<nIndexEntry>* is already marked for deletion.
-16	Invalid parameters are passed.
-18	Illegal HiPer-SEEK index file handle.

Description

HS_Delete() marks an index entry as deleted, but actually does not remove the entry from the file. The deletion mark can be removed with HS_Undelete(). This is similar to DbDelete() and DbRecall().

Info

See also:	DbDelete(), HS_IfDel(), HS_Replace(), HS_Undelete()
Category:	Database functions, HiPer-SEEK functions, Index functions, xHarbour extensions
Source:	rdd\hsx\hsx.c
LIB:	lib\xhb.lib
DLL:	dll\xhbdll.dll

Example

```
// The example creates a new HiPer-SEEK index file and
// transfers the Deleted() flag from the database to
// the index.

   PROCEDURE Main
      LOCAL bIndex, nHandle, nIndex

      CLS
      USE Customer
      SET DELETED OFF

      nHandle := HS_Create( "Customer.hsx", 8, 3, .F., 2 )
```

```
    bIndex  := {|| Trim(LastName)+" "+Trim(FirstName) }

    DO WHILE .NOT. Eof()
       nIndex := HS_Add( nHandle, bIndex )
       IF Deleted()
          HS_Delete( nHandle, Recno() )
       ENDIF
       SKIP
    ENDDO

    HS_Close( nHandle )
    USE
 RETURN
```

HS_Filter()

Uses a HiPer-SEEK index file as filter for a work area

Syntax

```
HS_Filter( <cFileName>|<nHsxHandle>, ;
           <cFilterToken>, ;
          [<cExpression>], ;
          [<nBufferSize>], ;
          [<nOpenMode>]    ) --> nRecCount
```

Arguments

<cFileName>

This is a character string holding the name of the HiPer-SEEK index file to use as filter. It must include path and file extension. If the path is omitted from *<cFileName>*, the file is searched in the current directory. If no file extension is given, the extension .HSX is used.

<cFilterToken>

This is a character string holding the tokens to search for in the HiPer-SEEK index file. Multiple tokens are separated with blank spaces.

<cExpression>

A character string holding the index key expression can optionally be specified.

<nBufferSize>

This is an optional numeric value specifying the memory buffer size in kB to be used by HS_*() functions. The default is 8 (=8*1024 bytes).

<nOpenMode>

This parameter defaults to 2, which opens the HiPer-SEEK index file READ ONLY and SHARED. Possible values for *<nOpenMode>* are:

Open modes for HiPer-SEEK index files

Value	Description
0	READ-WRITE + SHARED
1	READ-WRITE + EXCLUSIVE
2	READ-ONLY + SHARED
3	READ-ONLY + EXCLUSIVE4

Return

The function returns a numeric value greater or equal to zero when the HiPer-SEEK index file can be used as a filter. If this is successful, the return value indicates the number of records matching the filter tokens.

A negative value indicates an error condition:

Error codes of HS_Filter()

Value	Description
-16	Invalid parameters are passed.
-18	Illegal HiPer-SEEK index file handle.
-24	Work area not in use.
-25	RDD does not support bitmap filters.

Description

HS_Filter() allows to define a filter for a database open in a work area based on a search condition. The search is performed in a HiPer-SEEK index file which results in an extremely fast filter operation. The function filters all records matching the tokens specified with <*cFilterToken*>.

Note: the database must be open with an RDD that supports bitmap filters for HB_Filter() to work properly (RMDBFCDX, for example).

Info

See also: HS_Add(), HS_Next(), SET FILTER
Category: Database functions, HiPer-SEEK functions, Index functions, xHarbour extensions
Source: rdd\hsx\hsx.c
LIB: lib\xhb.lib
DLL: dll\xhbdll.dll

Example

```
// The example demonstrates filtering of records using
// a HiPer-SEEK index file

   REQUEST RMDBFCDX

   PROCEDURE Main
      LOCAL cToken := "ller ston"    // finds "Miller, Shiller, Jonston"
      LOCAL cFields:= "Cust->Lastname + Cust->Firstname"
      LOCAL nRecCount
      CLS
      USE Customer ALIAS Cust VIA "RMDBFCDX"

      nRecCount := HS_Filter( "Customer", cToken, cFields )

      ? nRecCount,"records matched",cToken

      WAIT

      GO TOP
      Browse()

      USE
   RETURN
```

HS_IfDel()

Checks if a HiPer-SEEK index entry is marked as deleted

Syntax

```
HS_IfDel( <nHsxHandle>, <nIndexEntry> ) --> nErrorCode
```

Arguments

<nHsxHandle>

This is the numeric file handle of the HiPer-SEEK index file to check. The file handle is returned from HS_Open() or HS_Create() !ELINK.

<nIndexEntry>

This is a numeric value specifying the ordinal position of the index entry to check.

Return

The function returns 1 when the index entry *<nIndexEntry>* is marked as deleted, zero when the deletion flag is not set, or a negative numeber indicating an error condition:

Error codes of HS_IfDel()

Value	Description
-4	Error while attempting to seek during buffer flushing.
-5	Read error while reading.
-6	Write error during buffer flushing.
-7	*<nIndexEntry>* is out of bounds.
-16	Invalid parameters are passed.
-18	Illegal HiPer-SEEK index file handle.

Description

HS_IfDel() checks if the deletion flag is set for an index entry in a HiPer-SEEK index file. This is analogous to the Deleted() function which tests if the current record is marked for deletion in a database file. The deletion flag in the HiPer-SEEK index is set with HS_Delete() and can be removed with HS_Undelete().

Info

See also:	Deleted(), HS_Delete(), HS_Undelete()
Category:	Database functions, HiPer-SEEK functions, Index functions, xHarbour extensions
Source:	rdd\hsx\hsx.c
LIB:	lib\xhb.lib
DLL:	dll\xhbdll.dll

HS_Index()

Creates a new HiPer-SEEK index file and fills it with index entries.

Syntax

```
HS_Index( <cFileName>   , ;
           <cExpression> , ;
          [<nKeySize>]   , ;
          [<nOpenMode>]  , ;
          [<nBufferSize>], ;
          [<lCaseInsens>], ;
          [<nFilterSet>]   ) --> nErrorCode
```

Arguments

`<cFileName>`

This is a character string holding the name of the HiPer-SEEK index file to create. It must include path and file extension. If the path is omitted from *<cFileName>*, the file is created in the current directory. If no file extension is given, the extension .HSX is used.

`<cExpression>`

This is a character string holding the index key expression.

`<nKeySize>`

An numeric value of 1, 2 or 3 can be passed. It defines the size of a single index entry in the HiPer-SEEK index file. The larger *<nKeySize>* is, the more unique index entries can be stored, reducing the possibility of false positives when searching a HiPer-SEEK index file. The size of an index entry is 16 bytes (1), 32 bytes (2) or 64 bytes (3). The default value is 2

`<nOpenMode>`

This parameter defaults to 1, which opens the HiPer-SEEK index file READ-WRITE and EXCLUSIVE after creation. Possible values for *<nOpenMode>* are:

Open modes for HiPer-SEEK index files

Value	Description
0	READ-WRITE + SHARED
1	READ-WRITE + EXCLUSIVE
2	READ-ONLY + SHARED
3	READ-ONLY + EXCLUSIVE

`<nBufferSize>`

This is a numeric value specifying the memory buffer size in kB to be used by HS_*() functions for this file. The default is 8 kB (8*1024 bytes).

`<lCaseInsens>`

This is an optional logical value. It defauls to .T. (true) which creates a case insensitive index. Passing .F. (false) creates a case sensitive index.

`<nFilterSet>`

This is a numeric value of 1 or 2 defining the filter set to use with the HiPer-SEEK index file. When *<nFilterSet>* is 1, which is the default, all non-printable characters, such as Tab, or carriage return/line feed, are treated as blank spaces and the high-order bit is ignored for characters (7-bit character set).

Specifying 2 recognizes all characters by their ASCII value.

Return

The function returns a numeric value greater or equal to zero when the HiPer-SEEK index file is successfully created and populated. This is the handle to the new file, which must be used with other HS_*() functions. A negative value indicates an error condition:

Error codes of HS_Index()

Value	Description
-4	Error while attempting to seek during buffer flushing.
-6	Write error during buffer flushing.
-16	Invalid parameters are passed.

Description

HS_Index() creates a new HiPer-SEEK index file and populates it with index entries according to the index key expression <*cExpression*>. The index expression must result in a character, or text string.

HiPer-SEEK index files provide the means for fast full-text search routines, since they contain unique index keys (hash values) for text strings. The size of an index key can be choosen as 16, 32 or 64 bytes. Text strings that result in different index keys, are found very fast within a HiPer-SEEK index file.

After the HiPer-SEEK index file is created, it can be searched for records. The search is initiated by defining the search text with HS_Set(), followed by subsequent calls to HS_Next().

Note: it is possible that two different text strings can result in the same index key. For this reason, function HS_Verify() is available to verify if a found text actually matches the original text.

Info

See also: HS_Add(), HS_Create(), HS_Next(), HS_Open(), HS_Set()
Category: Database functions, HiPer-SEEK functions, Index functions, xHarbour extensions
Source: rdd\hsx\hsx.c
LIB: lib\xhb.lib
DLL: dll\xhbdll.dll

Example

```
// The example creates a new, populated HiPER-SEEK index for
// a customer database.

   PROCEDURE Main
      LOCAL cIndex, nHandle

      CLS
      USE Customer ALIAS Cust

      cIndex := 'Trim(Cust->LastName) +'
      cIndex +=  '" "'
      cIndex += '+ Trim(Cust->FirstName)'

      nHandle := HS_Index( "Customer.hsx", cIndex, 3, 0, 16 )

      IF nHandle >= 0
         ? "HiPer-SEEK index successfully created with"
         ?? HS_KeyCount( nHandle), "index entries"
         HS_Close( nHandle )
      ELSE
         ? "HiPer-SEEK index creation failed with error:", nHandle
      ENDIF
```

```
   USE
RETURN
```

HS_KeyCount()

Returns the number of index entries in a HiPer-SEEK index file.

Syntax

```
HS_KeyCount( <nHsxHandle> ) --> nKeyCount
```

Arguments

<nHsxHandle>

This is the numeric file handle of the HiPer-SEEK index file to count the index entries in. The file handle is returned from HS_Open() or HS_Create().

Return

The function returns a numeric value greater or equal to zero representing the number of index entries contained in the HiPer-SEEK index. A negative value indicates an error condition:

Error codes of HS_KeyCount()

Value	Description
-16	Invalid parameter is passed.
-18	Illegal HiPer-SEEK index file handle.

Description

HS_KeyCount() is used to determine the total number of index entries available in a HiPer-SEEK index file.

Info

See also: Hs_Filter(), OrdKeyCount()
Category: Database functions, HiPer-SEEK functions, Index functions, xHarbour extensions
Source: rdd\hsx\hsx.c
LIB: lib\xhb.lib
DLL: dll\xhbdll.dll

HS_Next()

Searches a HiPer-SEEK index file for a matching index entry.

Syntax

```
HS_Next( <nHsxHandle> ) --> nRecno
```

Arguments

<nHsxHandle>

This is the numeric file handle of the HiPer-SEEK index file to use for a search. The file handle is returned from HS_Open() or HS_Create().

Return

The function returns a numeric value. When it is greater than zero, it represents the ordinal position of the index entry found in the file. The value zero is returned when no match is found, and a negative numeber indicates an error condition:

Error codes of HS_Next()

Value	Description
-5	Read error while reading.
-16	Invalid parameter is passed.
-18	Illegal HiPer-SEEK index file handle.

Description

HS_Next() initiates a search in a HiPer-SEEK index file after a new search string is defined with function HS_Set(), or searches the next index entry matching the current search string. The return value is a positive number as long as a matching index entry is found. When no more matches exist, the return value is zero.

Info

See also: HS_Set(), HS_Verify()
Category: Database functions, HiPer-SEEK functions, Index functions, xHarbour extensions
Source: rdd\hsx\hsx.c
LIB: lib\xhb.lib
DLL: dll\xhbdll.dll

Example

```
// The example outlines a typical HS_Set()/HS_Next() pattern used to
// find all records matching a search string.

   PROCEDURE Main
      LOCAL cIndex, nHandle
      LOCAL nRecno, cSearch := "ller"

      CLS
      USE Customer ALIAS Cust

      cIndex := 'Trim(Cust->LastName) +'
      cIndex +=  '" "'
      cIndex += '+ Trim(Cust->FirstName)'

      nHandle := HS_Index( "Customer.hsx", cIndex, 3, 0, 16 )
```

```
   IF nHandle < 0
      ? "HiPer-SEEK index creation failed with error:", nHandle
      QUIT
   ENDIF

   // define search string and find first index entry
   HS_Set( nHandle, cSearch )
   nRecno := HS_Next( nHandle )

   DO WHILE nRecno > 0
      DbGoto( nRecno )

      IF HS_Verify( cIndex, cSearch )
         ? "Found:", &(cIndex)
      ELSE
         ? "Not found:", nRecno
      ENDIF

      // find next index entry
      nRecno := HS_Next( nHandle )
   ENDDO

   HS_Close( nHandle )
   USE
RETURN
```

HS_Open()

Opens a HiPer-SEEK index file.

Syntax

```
HS_Open( <cFileName>  , ;
         <nBufferSize>, ;
         <nOpenMode>    ) --> nHsxHandle
```

Arguments

<cFileName>

This is a character string holding the name of the HiPer-SEEK index file to open. It must include path and file extension. If the path is omitted from *<cFileName>*, the file is searched in the current directory. If no file extension is given, the extension .HSX is used.

<nBufferSize>

This is a numeric value specifying the memory buffer size in kB to be used by other HS_*() functions for this file (the value 10 means 10240 bytes).

<nOpenMode>

This numeric parameter specifies how the HiPer-SEEK index file is opened. Possible values for *<nOpenMode>* are:

Open modes for HiPer-SEEK index files

Value	Description
0	READ-WRITE + SHARED
1	READ-WRITE + EXCLUSIVE
2	READ-ONLY + SHARED
3	READ-ONLY + EXCLUSIVE4

Return

The function returns a numeric value greater or equal to zero when the HiPer-SEEK index file can be opened. This value is a handle to the HiPer-SEEK index file and must be used with other HS_*() functions.

A negative return value indicates an error condition:

Error codes of HS_Open()

Value	Description
-2	Not enough memory or *<nBufferSize>* is too large.
-5	Read error while reading.
-10	File cannot be opened.
-16	Invalid parameters are passed.
-17	No more file handles available.
-20	File cannot be locked.
-21	Lock table exhausted.
-22	File cannot be unlocked.

Description

HS_Open() opens an existing HiPer-SEEK index file and returns a file handle to it. This handle must be preserved and passed to HS_Add() when new index entries should be added, or to HS_Set() and HS_Next() to perform a search.

Info

See also: HS_Add(), HS_Close(), HS_Create(), HS_Next(), HS_Set()
Category: Database functions, HiPer-SEEK functions, Index functions, xHarbour extensions
Source: rdd\hsx\hsx.c
LIB: lib\xhb.lib
DLL: dll\xhbdll.dll

HS_Replace()

Changes a HiPer-SEEK index entry.

Syntax

```
HS_Replace( <nHsxHandle>, ;
            <cNewIndex> , ;
            <nIndexEntry> ) --> nErrorCode
```

Arguments

`<nHsxHandle>`

This is the numeric file handle of the HiPer-SEEK index file to replace an index entry in. The file handle is returned from HS_Open() or HS_Create().

`<cNewIndex>`

This is a character string holding the new value for the index entry specified with *<nIndexEntry>*.

`<nIndexEntry>`

The ordinal position of the index entry to change must be specified as a numeric value.

Return

The function returns numeric 1 whan the index entry is successfully changed, or a negative number as error code:

Error codes of HS_Replace()

Value	Description
-4	Error while attempting to seek during buffer flushing.
-6	Write error during buffer flushing.
-7	*<nIndexEntry>* is out of bounds.
-16	Invalid parameters are passed.
-18	Illegal HiPer-SEEK index file handle.
-20	File cannot be locked.
-21	Lock table exhausted.
-22	File cannot be unlocked.

Description

HS_Replace() changes the value of the index entry *<nIndexEntry>* in a HiPer-SEEK index file. This is necessary, for example, when a database record is changed with REPLACE and the changes must be reflected in the index file. HiPer-SEEK index files are not automatically updated with REPLACE operations, but must be updated programmatically.

Info

See also: HS_Add(), HS_Delete(), REPLACE
Category: Database functions, HiPer-SEEK functions, Index functions, xHarbour extensions
Source: rdd\hsx\hsx.c
LIB: lib\xhb.lib
DLL: dll\xhbdll.dll

Example

```
// The example demonstrates how a HiPER seek index is updated after
// a database record is changed.
```

```
PROCEDURE Main
   LOCAL cIndex := "Cust->Lastname + Cust->Firstname + Cust->City"
   LOCAL nHandle, nRet

   USE Customer ALIAS Cust EXCLUSIVE

   nHandle := HS_Index( "Customer.hsx", cIndex, 3, 1 )

   GOTO 10
   REPLACE Cust->City WITH "New York"
   COMMIT

   nRet := HS_Replace( nHandle, &(cIndex), Recno() )
   IF nRet < 1
      ? "Error on updating index:", nRet
   ELSE
      ? "Index updated"
   ENDIF

   HS_Close( nHandle )
   USE

RETURN
```

HS_Set()

Defines a search string for subsequent HS_Next() calls.

Syntax

```
HS_Set( <nHsxHandle>, <cSearch> ) --> nIndexEntry
```

Arguments

<nHsxHandle>

This is the numeric file handle of the HiPer-SEEK index file to find *<cSearch>* in. The file handle is returned from HS_Open() or HS_Create().

<cSearch>

This is a character string holding the search string for subsequent HS_Next() calls.

Return

The function returns numeric 1 whan the search string is successfully defined, or a negative number as error code:

Error codes of HS_Set()

Value	Description
-4	Error while attempting to seek during buffer flushing.
-6	Write error during buffer flushing.
-13	HiPer-SEEK index file has no index entries.
-16	Invalid parameters are passed.
-18	Illegal HiPer-SEEK index file handle.

Description

HS_Set() defines the search string for a new Find operation in a HiPer-SEEK index file. The function prepares all internal parameters but dos not initiate a search. This must be accomplished with subsequent calls to the HS_Next() function which can be called repeatedly until no more matches are found.

Info

See also:	HS_Next()
Category:	Database functions, HiPer-SEEK functions, Index functions, xHarbour extensions
Source:	rdd\hsx\hsx.c
LIB:	lib\xhb.lib
DLL:	dll\xhbdll.dll

HS_Undelete()

Removes the deletion mark from an index entry in a HiPer-SEEK index file.

Syntax

```
HS_Undelete( <nHsxHandle>, <nIndexEntry> ) --> nErrorCode
```

Arguments

`<nHsxHandle>`

This is the numeric file handle of the HiPer-SEEK index file containing index entries marked as deleted. The file handle is returned from HS_Open() or HS_Create().

`<nIndexEntry>`

The ordinal position of the index entry to mark as undeleted must be specified as a numeric value.

Return

The function returns 1 when the deletion mark is successfully removed from the index entry, or a negative numeber indicating an error condition:

Error codes of HS_Undelete()

Value	Description
-4	Error while attempting to seek during buffer flushing.
-5	Read error while reading.
-6	Write error during buffer flushing.
-7	*<nIndexEntry>* is out of bounds.
-9	*<nIndexEntry>* is not marked for deletion.
-16	Invalid parameters are passed.
-18	Illegal HiPer-SEEK index file handle.

Description

HS_Undelete() removes the deletion mark from an index entry previously set with HS_Delete(). This is similar to DbDelete() and DbRecall().

Info

See also:	DbRecall(), HS_Delete(), HS_IfDel(), HS_Replace()
Category:	Database functions, HiPer-SEEK functions, Index functions, xHarbour extensions
Source:	rdd\hsx\hsx.c
LIB:	lib\xhb.lib
DLL:	dll\xhbdll.dll

HS_Verify()

Verifies a HS_Next() match against the index key.

Syntax

```
HS_Verify( <xIndexKey>, <cSearch> ) --> lMatched
```

Arguments

<xIndexKey>

This is either a code block holding the index key, or a character string holding the index key expression. A code block is recommended, since it does not need to be macro compiled.

<cSearch>

The searched value must be passed as a character string.

Return

The function returns .T. (true) when *<cSearch>* is contained anywhere in the result of the code block, or index value, otherwise .F. (false) is returned.

Description

HS_Verify() is used to make sure that the value found in a HiPer-SEEK index file actually matches the data in the corresponding database file. The HiPer-SEEK algorithm calculates fixed size hash keys from text strings. This makes it possible that two different text strings result in the same hash key so that a search may result in false positives. HS_Verify() prevents from finding false positives by comparing the search string against the data stored in the database.

Info

See also: HS_Next()
Category: Database functions, HiPer-SEEK functions, Index functions, xHarbour extensions
Source: rdd\hsx\hsx.c
LIB: lib\xhb.lib
DLL: dll\xhbdll.dll

Example

```
// The example demonstrates how HiPer-SEEK search operations
// are verified against a database.

   PROCEDURE Main
      LOCAL cIndex, bIndex, nHandle
      LOCAL nRecno, cSearch := "ller"

      CLS
      USE Customer ALIAS Cust

      cIndex := 'Trim(Cust->LastName) +'
      cIndex +=  '" "'
      cIndex += '+ Trim(Cust->FirstName)'
      bIndex := &( "{||" + cIndex + "}" )

      nHandle := HS_Index( "Customer.hsx", cIndex, 3, 0, 16 )

      IF nHandle < 0
         ? "HiPer-SEEK index creation failed with error:", nHandle
         QUIT
```

```
    ENDIF

    // define search string and find first index entry
    HS_Set( nHandle, cSearch )
    nRecno := HS_Next( nHandle )

    DO WHILE nRecno > 0
       DbGoto( nRecno )

       IF HS_Verify( bIndex, cSearch )
          ? "Matched :", Eval( bIndex )
       ELSE
          ? "No match:", Eval( bIndex )
       ENDIF

       // find next index entry
       nRecno := HS_Next( nHandle )
    ENDDO

    HS_Close( nHandle )
    USE
 RETURN
```

HS_Version()

Returns version information for HiPer-SEEK functions.

Syntax

```
HS_Version() -->  cVersionInfo
```

Return

The function returns a character string holding version information of HiPer-SEEK functions.

Info

See also:	HS_Create(), HS_Index(), Version()
Category:	Database functions, HiPer-SEEK functions, Index functions, xHarbour extensions
Source:	rdd\hsx\hsx.c
LIB:	lib\xhb.lib
DLL:	dll\xhbdll.dll

HtmlToAnsi()

Converts an HTML formatted text string to the ANSI character set.

Syntax

```
HtmlToAnsi( <cHtmlString> ) --> cAnsiString
```

Arguments

`<cHtmlString>`

This is a HTML formatted text string.

Return

The function returns a text string converted to the ANSI character set.

Description

Function HtmlToAnsi() removes HTML character entities from *<cHtmlString>* and returns the text string converted to the ANSI character set. Note that the function processes only HTML character entities (e.g. >, < or &) that are interchangeable in the OEM and ANSI character sets. Refer to AnsiToHtml() for a complete list of HTML character entities recognized by the function.

Info

See also:	AnsiToHtml(), HtmlToOem(), OemToHtml(), THtmlDocument()
Category:	Conversion functions, HTML functions, xHarbour extensions
Source:	tip\thtml.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows a result of HtmlToAnsi()

   PROCEDURE Main
      LOCAL cHtml := 'IF cChr &lt;&gt; "&amp;"'

      ? HtmlToAnsi( cHtml )  // result: IF cChr <> "&"

   RETURN
```

HtmlToOem()

Converts an HTML formatted text string to the OEM character set

Syntax

```
HtmlToOem( <cHtmlString> ) --> cOemString
```

Arguments

```
<cHtmlString>
```

This is a HTML formatted text string.

Return

The function returns a text string converted to the ANSI character set.

Description

Function HtmlToOem() removes HTML character entities from *<cHtmlString>* and returns the text string converted to the OEM character set. Note that the function processes only HTML character entities (e.g. >, < or &) that are interchangeable in the OEM and ANSI character sets. Refer to AnsiToHtml() for a complete list of HTML character entities recognized by the function.

Info

See also:	AnsiToHtml(), HtmlToAnsi(), OemToHtml(), THtmlDocument()
Category:	Conversion functions, HTML functions, xHarbour extensions
Source:	tip\thtml.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows a result of HtmlToOem()

   PROCEDURE Main
      LOCAL cHtml := 'IF cChr &lt;&gt; "&amp;"'

      ? HtmlToOem( cHtml )  // result: IF cChr <> "&"
   RETURN
```

I2Bin()

Converts a numeric value to a signed short binary integer (2 bytes).

Syntax

```
I2Bin( <nNumber> ) --> cInteger
```

Arguments

<nNumber>

A numeric value in the range of -(2^15) to +(2^15) - 1.

Return

The function returns a two-byte character string representing a 16-bit signed short binary integer.

Description

I2Bin() is a binary conversion function that converts a numeric value (Valtype()=="N") to a two byte binary number (Valtype()=="C").

The range for the numeric return value is determined by a signed short integer. If *<nNumber>* is outside this range, a runtime error is raised.

I2Bin() is the inverse function of Bin2I().

Info

See also:	Asc(), Bin2I(), Bin2L(), Bin2U(), Bin2W(), Chr(), L2Bin(), U2Bin(), W2Bin()
Category:	Binary functions, Conversion functions
Source:	rtl\binnum.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates return values of I2Bin().

   PROCEDURE Main
      LOCAL cInt

      cInt := I2Bin(0)
      ? Asc( cInt[1] ), Asc( cInt[2] )  // result:   0    0

      cInt := I2Bin(1)
      ? Asc( cInt[1] ), Asc( cInt[2] )  // result:   1    0

      cInt := I2Bin(-1)
      ? Asc( cInt[1] ), Asc( cInt[2] )  // result: 255  255

      cInt := I2Bin(256)
      ? Asc( cInt[1] ), Asc( cInt[2] )  // result:   0    1

      cInt := I2Bin(-256)
      ? Asc( cInt[1] ), Asc( cInt[2] )  // result:   0  255

      cInt := I2Bin(32767)
      ? Asc( cInt[1] ), Asc( cInt[2] )  // result: 255  127

      cInt := I2Bin(-32768)
      ? Asc( cInt[1] ), Asc( cInt[2] )  // result:   0  128
```

```
RETURN
```

If() | IIf()

Returns the result of an expression based on a logical expression

Syntax

```
If( <lCondition>, <exprTrue>, <exprFalse> ) --> xValue
or
IIf(<lCondition>, <exprTrue>, <exprFalse> ) --> xValue
```

Arguments

<lCondition>

This is a logical expression which determines the return value of If().

<exprTrue>

An expression resulting in any data type. It is evaluated when *<lCondition>* is .T. (true).

<exprFalse>

An expression resulting in any data type. It is evaluated when *<lCondition>* is .F. (false).

Return

The function returns the value of *<exprTrue>* if *<lCondition>* is .T. (true), otherwise the value of *<exprFalse>* is returned.

Description

Function If(), and its syntactical synonym IIf(), is a conditional function providing two possible return values. It first evluates the expression *<lCondition>*. Based on this result, it returns the value of either *<exprTrue>* or *<exprFalse>*.

Info

See also:	DO CASE, IF, SWITCH
Category:	Conversion functions, Logical functions
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example sorts a two column array. IIf() compares values
// in the second column when adjacent array elements in the
// first column contain identical values.


   PROCEDURE Main
      LOCAL aArray := { ;
            { "B", 3 }, ;
            { "A", 2 }, ;
            { "B", 1 }, ;
            { "C", 4 }, ;
            { "A", 1 }  }

      ASort( aArray,,, {|a,b| IIf( a[1] == b[1], ;
                                   a[2] <  b[2], ;
                                   a[1] <  b[1]  ) } )

      AEval( aArray, {|a| Qout( ValToPrg(a) ) } )
      // result:
      // { "A", 1 }
```

```
   // { "A", 2 }
   // { "B", 1 }
   // { "B", 3 }
   // { "C", 4 }
RETURN
```

IndexExt()

Retrieves the default index file extension in a work area.

Syntax

```
IndexExt() --> cFileExtension
```

Return

The function returns the default file extension for index files in a work area. If a work area is unused, the return value is an empty string ("").

Description

The default extension for index files depends on the RDD used to open a database file. IndexExt() returns the default extension used by the RDD if an index file should be created and the file name is specified without extension. IndexExt() does not return the file extension of an index file in use.

Note: The function exists for compatibility reasons. It is recommended to use OrdBagExt() instead.

Info

See also:	DbInfo(), IndexKey(), IndexOrd(), OrdBagExt()
Category:	Database functions, Index functions
Source:	rdd\rddord.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example checks the existence of an index file using
// the default file extension for indexes.

   PROCEDURE Main
      USE Customer NEW EXCLUSIVE

      IF .NOT. File( "Cust01" + IndexExt() )
         INDEX ON Upper(LastName+FirstName) TO Cust01
      ENDIF

      USE
   RETURN
```

IndexKey()

Returns the index expression of an open index.

Syntax

```
IndexKey( <nOrder> ) --> cIndexKey
```

Arguments

<nOrder>

A numeric value specifying the ordinal position of the index open in a work area. Indexes are numbered in the sequence of opening, beginning with 1. The value zero identifies the controlling index.

Return

The function returns the index key expression of the specified index as a character string. If the work area is unused, an empty string ("") is returned.

Description

The function is used to obtain the index key expression of an index open in a work area. The returned character string can be evaluated using the macro operator (&) to obtain the index value of the current database record. This is useful, for example, when data is edited by the user and written back to the database file. If this operation changes the index value of the controlling index, a screen update may be required, especially in browse views.

Note: The function exists for compatibility reasons. It is recommended to use OrdKey() instead.

Info

See also: INDEX, IndexExt(), IndexOrd(), OrdCreate(), OrdKey(), OrdListAdd(), OrdNumber(), SET INDEX, SET ORDER, USE
Category: Database functions, Index functions
Source: rdd\rddord.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example creates two indexes and displays their
// key expressions.

   REQUEST DBFCDX

   PROCEDURE Main
      RddSetDefault( "DBFCDX" )
      USE Customer

      INDEX ON CustID                     TAG ID   TO Cust01
      INDEX ON Upper(LastName+FirstName)  TAG Name TO Cust01

      ? IndexOrd()                // result: 2

      ? IndexKey(0)               // result: Upper(LastName+FirstName)
      ? IndexKey(1)               // result: CustID
      ? IndexKey(2)               // result: Upper(LastName+FirstName)

      USE
   RETURN
```

IndexOrd()

Returns the ordinal position of the controlling index in a work area.

Syntax

```
IndexOrd() --> nOrder
```

Return

The function returns the ordinal position of the controlling index as a numeric value. If no index or database is open in the work area, the return value is zero.

Description

The function is used to retrieve the ordinal position of the controlling index in the list of open indexes. Indexes are numbered in the sequence of opening, beginning with 1.

IndexOrd() is mainly used to save the position of the controlling index, temporarily change the controlling index, and restore it using DbSetOrder() or OrdSetFocus() when a database operation is complete.

Info

See also:	INDEX, DbSetOrder(), IndexKey(), OrdKey(), OrdListAdd(), OrdNumber(), OrdSetFocus(), SET INDEX, SET ORDER, USE
Category:	Database functions, Index functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example illustrates return values of IndexOrd().

   REQUEST DBFCDX

   PROCEDURE Main
      RddSetDefault( "DBFCDX" )
      USE Customer

      INDEX ON CustNO                    TAG ID   TO Cust01
      INDEX ON Upper(LastName+Firstname)  TAG Name TO Cust01

      ? IndexOrd()              // result: 2
      ? IndexKey(0)             // result: Upper(LASTNAME+FIRSTNAME)

      DbSetOrder(1)

      ? IndexOrd()              // result: 1
      ? IndexKey(0)             // result: CustID

      USE
   RETURN
```

INetAccept()

Waits for an incoming connection on a server side socket.

Syntax

```
INetAccept( <pServerSocket> ) --> pClientSocket
```

Arguments

<pSocket>

This is a pointer to a socket as returned by INetServer().

Return

The function returns a socket to a connected client process.

Description

Function INetAccept() is the only function available to establish a connection between the current process (server side) and a remote process (client side) that wants to connect to the server process. The socket *<pServerSocket>* must be created with function INetServer(). When passed to INetAccept(), the function does not return until a remote client process requests a connection. For this reason, it is recommended to set a timeout value with INetSetTimeout() in order to stop listening on the server socket periodically and react to user input in the server process.

An alternative is to use a separate thread that calls INetAccept() while the main thread is processing user input. Note that INetAccept() returns when *<pServerSocket>* is closed with INetClose(). That is, if INetAccept() is called in a separate thread, this thread is blocked until a connection request from outside is detected. If the main thread closes the server socket while a second thread waits for a connection request, INectAccept() returns with an socket error, and the second thread can terminate.

The socket returned from INetAccept() can be used to exchange data between the current process and a remote process using INetRecv() and INetSend().

An example for a complete client/server communication is provided with function INetServer().

Info

See also:	INetConnect(), INetInit(), INetRecv(), INetSend(), INetServer(), INetSetTimeout()
Category:	Internet functions, Sockets functions, xHarbour extensions
Source:	vm\INet.c
LIB:	xhb.lib
DLL:	xhbdll.dll

INetAddress()

Determines the internet address of a remote station.

Syntax

```
INetAddress( <pSocket> ) --> cInternetAddress
```

Arguments

<pSocket>

This is a pointer to a socket.

Return

The function returns a character string holding the internet address of the remote station in quad dot notation (e.g. "192.168.1.145").

Description

Function INetAddress() accepts a socket and determines the internet address of the internet connection.

Info

See also:	INetAccept(), INetConnect(), INetDGram(), INetPort(), INetGetHosts()
Category:	Internet functions, Sockets functions, xHarbour extensions
Source:	vm\inet.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays the internet address of www.xHarbour.com

   PROCEDURE Main
      LOCAL pSocket

      INetInit()

      pSocket := INetConnect( "www.xharbour.com", 80 )

      ? INetAddress( pSocket )   // result: 193.239.210.10

      INetClose( pSocket )
      INetCleanUp()
   RETURN
```

INetCleanup()

Releases memory resources for sockets.

Syntax

```
INetCleanup() --> NIL
```

Return

The return value is always NIL.

Description

INetCleanup() should be called at the end of any program using sockets functions, or when sockets are no longer required. The function releases all memory resources allocated with INetInit().

Info

See also:	INetAccept(), INetInit(), INetConnect()
Category:	Internet functions, Sockets functions, xHarbour extensions
Source:	vm\inet.c
LIB:	xhb.lib
DLL:	xhbdll.dll

INetClearError()

Resets the last error code of a socket.

Syntax

```
INetClearError( <pSocket> ) --> NIL
```

Arguments

`<pSocket>`

This is a pointer to a socket.

Return

The return value is always NIL.

Description

Function INetClearError() resets the last error code on a socket to zero and voids the last error description. This makes sure that an error can be detected when a call to a next sockets function fails.

Info

See also:	INetAccept(), INetConnect(), INetDGram(), INetErrorCode(), INetErrorDesc()
Category:	Internet functions, Sockets functions, xHarbour extensions
Source:	vm\inet.c
LIB:	xhb.lib
DLL:	xhbdll.dll

INetClearPeriodCallback()

Voids a callback function for a socket.

Syntax

```
INetClearPeriodCallback( <pSocket> ) --> NIL
```

Arguments

<pSocket>

This is a pointer to a socket as returned by INetConnect() or INetAccept().

Return

The return value is always NIL.

Description

Function INetClearPeriodCallback() is used to remove callback information set with INetSetPeriodCallback() from a socket.

Info

See also:	INetClearTimeout(), INetCreate(), INetSetPeriodCallback(), INetSetTimeout()
Category:	Internet functions, Sockets functions, xHarbour extensions
Source:	vm\INet.c
LIB:	xhb.lib
DLL:	xhbdll.dll

INetClearTimeout()

Voids a timout value for a socket.

Syntax

```
INetClearTimeout( <pSocket> ) --> NIL
```

Arguments

<pSocket>

This is a pointer to a socket as returned by INetConnect() or INetAccept().

Return

The return value is always NIL.

Description

Function INetClearTimeout() is used to remove a timeout value set with INetSetTimeout() from a socket.

Info

See also:	INetClearPeriodCallback(), INetCreate(), INetSetPeriodCallback(), INetSetTimeout()
Category:	Internet functions, Sockets functions, xHarbour extensions
Source:	vm\INet.c
LIB:	xhb.lib
DLL:	xhbdll.dll

INetClose()

Closes a connection on a socket.

Syntax

```
INetClose( <pSocket> ) --> nError
```

Arguments

<pSocket>

This is a pointer to a socket.

Return

The function returns zero on success, or -1 on failure.

Description

INetClose() closes a socket and notifies both ends of the connection. If threads are executing a blocking sockets function with *<pSocket>*, they will terminate their wait state and a socket error is set ("socket closed"). This causes all threads waiting on a blocking sockets function to resume so they can report a sockets error.

If the return value is not zero, use function INetErrorCode() to obtain information about failure.

Info

See also: INetAccept(), INetConnect(), INetDGram(), INetErrorCode()
Category: Internet functions, Sockets functions, xHarbour extensions
Source: vm\inet.c
LIB: xhb.lib
DLL: xhbdll.dll

INetConnect()

Establishes a sockets connection to a server.

Syntax

```
INetConnect( <cURL>, <nPort> ) --> pSocket

or

INetConnect( <cURL>, <nPort>, <pSocket> ) --> NIL
```

Arguments

<cURL>

This is a character string holding either the URL (DNS name) of the server to connect to, or its IP address in dotted notation (e.g. "192.168.1.145").

<nPort>

This is the numeric port number on the local computer to use for the socket connection. The default internet port is 80.

<pSocket>

Optionally, a socket created with INetCreate() or previously closed with INetClose() can be passed to establish a connection.

Return

If called with two parameters, the return value is a pointer to a socket. If a socket is passed as third parameter, the return value is NIL.

Description

Before INetConnect() can be called, the sockets system must be initialized with INetInit(). INetConnect() is a client side sockets function that establishes a connection to a server process. The server's internet address can be passed as the DNS name of the remote station or in dottet notation. If the connection is successfully established, the socket can be used for sending and receiving data to/from the server. If the connection fails, the cause of failure can be detected with INetErrorCode().

Note: the function is not thread safe! It is the responsibility of the programmer to make sure that InetConnect() is not called simultaneously in two ore more threads, or that INetGetHosts() is not called at the same time as InetConnect().

Use INetConnectIP() for building thread safe connections, or protect a call to INetConnect() with a Mutex.

Info

See also:	INetAccept(), INetCreate(), INetConnectIP(), INetDGram(), INetInit(), INetRecv(), INetSend(), INetServer()
Category:	Internet functions, Sockets functions, xHarbour extensions
Source:	vm\inet.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example outlines how a simple internet connection can be
// established for downloading a HTML page using the HTTP protocol.
```

```
#define CRLF  Chr(13)+Chr(10)

PROCEDURE Main
   LOCAL cBuffer, cRequest, cResponse, nBytes, pSocket

   // initialize sockets system and connect to server
   INetInit()
   pSocket := INetConnect( "www.xharbour.com", 80 )

   IF INetErrorCode( pSocket ) <> 0
      ? "Socket error:", INetErrorDesc( pSocket )
      INetCleanUp()
      QUIT
   ENDIF

   // send HTTP request to server
   cRequest := "GET / HTTP/1.1"                + CRLF + ;
               "Host: www.xharbour.com"        + CRLF + ;
               "User-Agent: HTTP-Test-Program" + CRLF + ;
                CRLF

   nBytes   := INetSend( pSocket, cRequest )
   cBuffer  := Space(4096)
   cResponse:= ""

   // get HTTP response from server
   DO WHILE ( nBytes > 0 )
      nBytes    := INetRecv( pSocket, @cBuffer )
      cResponse += Left( cBuffer, nBytes )
      cBuffer   := Space(4096)
   ENDDO

   // disconnect and cleanup memory
   INetClose( pSocket )
   INetCleanUp()

   // save response and display
   Memowrit( "xharbour.txt", cResponse )
   Memoedit( cResponse )
RETURN
```

INetConnectIP()

Establishes a sockets connection to a server using the IP address.

Syntax

```
INetConnectIP( <cIPAddress>, <nPort> ) --> pSocket

or

INetConnectIP( <cIPAddress>, <nPort>, <pSocket> ) --> NIL
```

Arguments

`<cIPAddress>`

This is a character string holding IP address of the server in dotted notation (e.g. "192.168.1.145").

`<nPort>`

This is the numeric port number on the local computer to use for the socket connection. The default internet port is 80.

`<pSocket>`

Optionally, a socket created with INetCreate() or previously closed with INetClose() can be passed to establish a connection.

Return

If called with two parameters, the return value is a pointer to a socket. If a socket is passed as third parameter, the return value is NIL.

Description

INetConnectIP() does the same as INetConnect() except for not resolving the IP address from the server's URL, or DNS name. INetConnectIP() accepts only the IP address of the server in dottet notation. Therefore, the function is thread safe and can be called simultaneously in multiple threads.

If the connection is successfully established, the socket can be used for sending and receiving data to/from the server. If the connection fails, the cause of failure can be detected with INetErrorCode().

Info

See also:	INetAccept(), INetCreate(), INetConnect(), INetDGram(), INetInit(), INetRecv(), INetSend()
Category:	Internet functions, Sockets functions, xHarbour extensions
Source:	vm\inet.c
LIB:	xhb.lib
DLL:	xhbdll.dll

INetCount()

Returns the number of bytes transferred in the last sockets operation

Syntax

```
INetCount( <pSocket> ) --> nBytes
```

Arguments

<pSocket>

This is a pointer to a socket as returned by INetConnect() or INetAccept().

Return

The function returns the number of bytes transferred in the last sockets operation as a numeric value.

Description

The function is used to determine how many bytes are read from the socket *<pSocket>* with INetRecv(), or written to it with INetSend().

Info

See also:	INetConnect(), INetDataReady(), INetRecv(), INetRecvAll(), INetSend(), INetSendAll()
Category:	Internet functions, Sockets functions, xHarbour extensions
Source:	vm\inet.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example uses INetCount() to detect the end of a file download
// from a HTTP server.

   PROCEDURE Main
      LOCAL pSocket, cBuffer, nBytes, cRequest, cResponse

      CLS

      // initialize sockets system and connect to server
      INetInit()
      pSocket := INetConnect( "www.xharbour.com", 80 )

      IF INetErrorCode( pSocket ) <> 0
         ? "Socket error:", INetErrorDesc( pSocket )
         INetCleanUp()
         QUIT
      ENDIF

      // send HTTP request to server
      cRequest := "GET / HTTP/1.1"                + INetCRLF() + ;
                  "Host: www.xharbour.com"        + INetCRLF() + ;
                  "User-Agent: HTTP-Test-Program" + INetCRLF() + ;
                                                    INetCRLF()

      INetSend( pSocket, cRequest )
      ? "Bytes sent:", INetCount( pSocket )

      cResponse:= ""
```

```
   // get HTTP response from server
   DO WHILE INetCount( pSocket) > 0
      cBuffer := Space( 4096 )
      INetRecv( pSocket, @cBuffer )

      nBytes := INetCount( pSocket )
      ? "Bytes received:", nBytes

      cResponse += Left( cBuffer, nBytes )
   ENDDO

   // disconnect and cleanup memory
   INetClose( pSocket )
   INetCleanUp()

   // save response
   Memowrit( "xharbour.txt", cResponse )
   ? "Data written to file: xHarbour.txt"
RETURN
```

INetCreate()

Creates a raw, unconnected socket.

Syntax

```
INetCreate() --> pSocket
```

Return

The functions returns a pointer to an unconnected socket.

Description

INetCreate() creates the raw data of a socket, that can be configured further before establishing a connection on the socket with INetAccept() or INetConnect(). For example, a timeout value or callback information can be added to a socket before it is passed to a connection function.

Info

See also:	INetAccept(), INetConnect(), INetSetPeriodCallback(), INetSetTimeout()
Category:	Internet functions, Sockets functions, xHarbour extensions
Source:	vm\inet.c
LIB:	xhb.lib
DLL:	xhbdll.dll

INetCRLF()

Returns new line characters used in internet communications.

Syntax

```
INetcrlf() --> cCRLF
```

Return

The return value is a carriage return+line feed pair.

Description

The function returns a platform independent pair of carriage return+line feed characters.

Info

See also:	INetSend(), INetSendAll()
Category:	Internet functions, Sockets functions, xHarbour extensions
Source:	vm\INet.c
LIB:	xhb.lib
DLL:	xhbdll.dll

INetDataReady()

Tests if incoming data is available to be read.

Syntax

```
INetDataReady( <pSocket> , [<nMilliSecs>] ) --> nResult
```

Arguments

`<pSocket>`

This is a pointer to a socket as returned by INetConnect() or INetAccept().

`<nMilliSecs>`

If specified, this is a numeric value indicating the maximum number of milli seconds to wait until incoming data is detected.

Return

The function returns 1 if incoming data on *<pSocket>* is detected, or 0 if no data is available. When a network error occurs, the return value is -1.

Description

Function INetDataReady() is used to detect if data is available to be read from *<pSocket>* without blocking program execution. If *<nMillisecs>* is not specified, the function returns immediately 1 when there is some data to be read. It returns 0, if there is no data, and -1 in case of error. If *<nMillisecs>* is given, the functon waits for this amount of time for incoming data. If some data arrives during that period of time, the wait state is immediately terminated and the function returns 1.

If incoming data is detected, a subsequent call to InetRecv() will read available data, up to the requested length, without blocking program execution.

If an error occurs, InetErrorCode() can be used to determine the cause of failure.

Info

See also:	INetAccept(), INetConnect(), INetCount(), INetRecv(), INetRecvAll()
Category:	Internet functions, Sockets functions, xHarbour extensions
Source:	vm\INet.c
LIB:	xhb.lib
DLL:	xhbdll.dll

INetDGram()

Creates an unbound datagram oriented socket.

Syntax

```
INetDGram( [<lBroadcast>] ) --> pSocket
```

Arguments

`<lBroadcast>`

This parameter defaults to .F. (false). When set to .T. (true), the function creates a broadcast capable socket, able to send and receive broadcast messages. Note that this requires certain user privileges on most operating systems.

Return

The functions returns a pointer to an unbound datagram socket.

Description

Function INetDGram() creates an unbound socket configured for the User Datagram Protocol (UDP). The socket is able to send and eventually receive data. Since the socket is not bound, a program cannot retrieve the address at which the socket appears to be, but a second socket receiving a message sent from *<pSocket>* is able to reply correctly with a datagram that can be read from an unbound socket.

If *<lBroadcast>* is set to .T. (true) a broadcast capable socket is returned. However, this may require certain user privileges.

If the socket cannot be created, a socket error is set that can be analyzed with function INetErrorCode().

Note: refer to function INetDGramSend() for an example of datagram exchange between two processes.

Info

See also:	INetDGramBind(), INetDGramRecv(), INetDGramSend()
Category:	Datagram functions, Sockets functions, xHarbour extensions
Source:	vm\INet.c
LIB:	xhb.lib
DLL:	xhbdll.dll

INetDGramBind()

Creates a bound datagram oriented socket.

Syntax

```
INetDGramBind( <nPort>       , ;
              [<cIPAddress>], ;
              [<lBroadcast>], ;
              [<cMulticast>]  ) --> pSocket
```

Arguments

<nPort>

This is the numeric port number on the local computer to bind the datagram socket to.

<cIPAddress>

If specified, this is a character string holding the IP address of a server in dotted notation (e.g. "192.168.1.145").

<lBroadcast>

This parameter defaults to .F. (false). When set to .T. (true), the function creates a broadcast capable socket, able to send and receive broadcast messages.

<cMulticast>

If specified, this is a character string holding the IP multicast address of a group in dotted notation.

Return

The functions returns a pointer to a bound datagram socket.

Description

Function INetDGramBind() creates a bound socket configured for the User Datagram Protocol (UDP). The socket is able to send and receive data.

If *<lBroadcast>* is set to .T. (true) a broadcast capable socket is returned. However, this may require certain user privileges.

A broadcast socket can be restricted to a group by specifiying a multicast IP address.

If the socket cannot be created, a socket error is set that can be analyzed with function INetErrorCode().

Note: refer to function INetDGramSend() for an example of datagram exchange between two processes.

Info

See also:	INetDGram(), INetDGramRecv(), INetDGramSend()
Category:	Datagram functions, Sockets functions, xHarbour extensions
Source:	vm\INet.c
LIB:	xhb.lib
DLL:	xhbdll.dll

INetDGramRecv()

Reads data from a datagram socket.

Syntax

```
INetDGramRecv( <pSocket>, ;
               @<cBuffer>, ;
               [<nBytes>]  ) --> nReceivedBytes
```

Arguments

`<pSocket>`

This is a pointer to a socket as returned by INetDGram() or INetDGramBind()

`<cBuffer>`

A memory variable holding a character string must be passed by reference to INetDGramRecv(). It must have at least *<nBytes>* characters.

`<nBytes>`

This is a numeric value specifying the number of bytes to transfer from the socket into the memory variable *<cBuffer>*. It defaults to Len(*<cBuffer>*).

Return

The function returns a numeric value indicating the number of bytes read from the socket. If the return value is smaller than Len(*<cBuffer>*), either no more bytes are available, or a network error ocurred. This can be identified using function INetErrorCode().

Description

INetDGramRecv() blocks the current thread until *<nBytes>* bytes are read from a datagram socket *<pSocket>*. The received bytes are copied into the memory variable *<cBuffer>*, which must be passed by reference.

To avoid blocking, function INetDataReady() can be used to detect if incoming data is available, or a timeout value can be set with INetSetTimeout().

Note: it is not guaranteed that all the data required to be read is sent from the kernel to the application. The kernel just returns the last complete datagram that has been received, up to *<nBytes>* bytes.

Refer to function INetDGramSend() for an example of datagram exchange between two processes.

Info

See also:	INetDGram(), INetDGramBind(), INetDGramSend(), INetSetTimeout()
Category:	Datagram functions, Sockets functions, xHarbour extensions
Source:	vm\INet.c
LIB:	xhb.lib
DLL:	xhbdll.dll

INetDGramSend()

Sends data to a datagram socket.

Syntax

```
INetDGramSend( <pSocket>   , ;
               <cIPAddress>, ;
               <nPort>     , ;
               <cData>     , ;
              [<nBytes>]     ) --> nSentBytes
```

Arguments

<pSocket>

This is a pointer to a socket as returned by INetDGram() or INetDGramBind()

<cIPAddress>

This is a character string holding the IP address of a remote station in dotted notation (e.g. "192.168.1.145").

<nPort>

This is the numeric port number on the local computer to use for sending data. Valid port numbers must be in the range from 0 to 65535.

<cData>

A character string holding the actual data to send via *<pSocket>*.

<nBytes>

This is a numeric value specifying the number of bytes to send. It defaults to Len(*<cBuffer>*).

Return

The function returns a numeric value indicating the number of bytes sent on the socket, or -1 on error. Use function INetErrorCode() to detect the cause of failure.

Description

Function INetDGramSend() sends the number of *<nBytes>* bytes of *<cData>* from the local computer to the specified IP address and port via the datagram socket *<pSocket>*.

If the function succeds, it returns the number of bytes that are actually sent.

Note: there is no guarantee that all the data requested to be sent is actually sent to the socket when its size exceeds the system datagram size. Therefore, the return value should be compared with the length of *<cData>* and, if a smaller value is returned, INetDGramSend() should be called iteratively to send remaining data until the message is complete.

Info

See also: INetDGram(), INetDGramBind(), INetDGramRecv()
Category: Datagram functions, Sockets functions, xHarbour extensions
Source: vm\INet.c
LIB: xhb.lib
DLL: xhbdll.dll

Examples

UDP Server

```
// This example implements a simple User Datagram Protocol server demonstrating
// basic steps for exchanging datagrams. A bound server socket is created with
// INetDGramBind(). Then the DO WHILE loop checks the keyboard twice a second.
// If no key is pressed, the server checks for incoming data with INetDataReady().
// If there is data, it is read in procedure ServeDGram() and an "OK!" response
// is returned to the connecting process.

   PROCEDURE Main
      LOCAL pSocket, cData

      CLS

      INetInit()

      // listen on port 1800
      pSocket := InetDGramBind( 1800 )

      ? "Server up and running", pSocket
      ? "Press any key to quit"

      DO WHILE Inkey(.5) == 0
         // check for incoming connection requests
         IF INetDataReady( pSocket ) > 0
            // process datagram connection
            ServeDGram( pSocket )
         ELSE
            // display some kind of "I am still alive" message
            ?? "."
         ENDIF
      ENDDO

      // close socket and cleanup memory
      INetClose( pSocket )
      INetCleanup()
   RETURN


   PROCEDURE ServeDGram( pSocket )
      LOCAL cBuffer, nBytes, cData := Space(80)

      // display IP address of client
      ? "Serving:", INetAddress( pSocket )

      // receive datagram (max 80 bytes in this example)
      nBytes := INetDGramRecv( pSocket, @cData, Len(cData) )

      ? "Bytes received:", nBytes
      ? "Data  received:", Left( cData, nBytes )
```

```
      // report "OK!" to remote process
      INetDGramSend( pSocket, INetAddress( pSocket ), INetPort( pSocket ), "OK!" )
   RETURN
```

UDP Client

```
// This example is the client side of a User Datagram Protocol communication.
// The client socket is created being capable of receiving broadcast messages.
// The user can enter data in a single GET. This data is transferred to the
// server process, and the response of the server process is displayed in the
// local process.
// Since a maximum of only 80 bytes is sent in the example, INetDGramSend()
// is not called iteratively.

   #include "Inkey.ch"

   PROCEDURE Main( cServerIPAddress )
      LOCAL pSocket, cData, nBytes

      IF Empty( cServerIPAddress )
         // IP address of the local station
         cServerIPAddress := "127.0.0.255"
      ENDIF

      INetInit()

      // create an unbound broadcast socket
      pSocket := INetDGRam( .T. )

      IF InetDGramSend( pSocket, cServerIPAddress, 1800, "ALIVE?" ) < 1
         ? "Unable to connect to "
         ?? cServerIPAddress, "Error:", InetErrorDesc( Socket )
         INetCleanUp()
         QUIT
      ELSE
         ? "Found a server at ", InetAddress( pSocket )
         WAIT
      ENDIF

      // set max. timeout to 10 seconds for receiving server response
      INetSetTimeout( pSocket, 10000 )

      DO WHILE Lastkey() <> K_ESC
         CLS

         // let the user enter some data
         cData := Space(80)
         @ 2,2 SAY "Enter a string:" GET cData PICTURE "@S30"
         READ

         IF Lastkey() == K_ESC
            EXIT
         ENDIF

         cData  := Trim( cData )
         nBytes := Len ( cData )

         InetDGramSend( pSocket, cServerIPAddress, 1800, cData, nBytes )

         CLS
         ? "sent:", LTrim(Str(nBytes)), "bytes"
```

```
      // Receive the server's response
      cData := Space( 80 )

      nBytes := INetDGramRecv( pSocket, @cData, 80 )

      IF INetErrorCode( pSocket ) == 0
         ? "received:", Left( cData, nBytes )
      ELSE
         ? "error:", INetErrorDesc( pSocket )
      ENDIF

      WAIT
   ENDDO

   // disconnect socket
   INetClose( pSocket )

   INetCleanUp()
RETURN
```

INetErrorCode()

Returns the last sockets error code.

Syntax

```
INetErrorCode( <pSocket> ) --> nErrorCode
```

Arguments

`<pSocket>`

This is a pointer to a socket.

Return

The function returnes the error code of the last sockets operation as a numeric value.

Description

When a sockets function fails, the error code of the sockets operation is stored in *<pSocket>* and can be retrieved later with function INetErrorCode(). This error code remains with *<pSocket>* until a new sockets function is called with this socket or INetClearError() is executed.

To obtain a human readable description of a sockets error, call INetErrorDesc() with *<pSocket>*.

Info

See also:	INetAccept(), INetClearError(), INetDGram(), INetConnect(), INetErrorDesc()
Category:	Internet functions, Sockets functions, xHarbour extensions
Source:	vm\INet.c
LIB:	xhb.lib
DLL:	xhbdll.dll

INetErrorDesc()

Returns a descriptive error message

Syntax

```
INetErrorDesc( <pSocket> ) --> cErrorMessage
```

Arguments

<pSocket>

This is a pointer to a socket.

Return

The function retunrs a character string holding an error message. If no error occurred, the return value is a null string ("").

Description

Function INetErrorDesc() is used to obtain the description of an error when a socket function fails. It translates the numeric error code into a human readable message.

Info

See also:	INetAccept(), INetConnect(), INetDGram(), INetErrorCode()
Category:	Internet functions, Sockets functions, xHarbour extensions
Source:	vm\INet.c
LIB:	xhb.lib
DLL:	xhbdll.dll

INetGetAlias()

Retrieves alias names from a server.

Syntax

```
INetGetAlias( <cURL> ) --> aAliasNames
```

Arguments

<cURL>

This is a character string holding the URL (DNS name) of the server to query.

Return

The function returns an array holding the server's alias names as character strings. If the server does not report aliases, an empty array is returned.

Description

INetGetAlias() is used to obtain alias names under which a server is currently known. Alias names are CNAME DNS records (Canonical Name records). A server can have an unlimited number of aliases. Whether or not they can be queried, however, depends on the verbosity of the server. If a server does not report aliases, the function returns an empty array.

Info

See also: INetAddress(), INetGetHosts()
Category: Internet functions, Sockets functions, xHarbour extensions
Source: vm\INet.c
LIB: xhb.lib
DLL: xhbdll.dll

INetGetHosts()

Queries IP addresses associated with a name.

Syntax

```
INetGetHosts( <cHostName> ) --> aIPAddresses
```

Arguments

<cHostName>

This is a character string holding either the URL (DNS name) of the host to query, or its IP address in dotted notation (e.g. "192.168.1.145").

Return

The function returns an array filled with character strings holding IP addresses in dotted notation. If a network error occurs, the function returns NIL.

Description

Function INetGetHosts() is used to resolve IP addresses from a host name. The IP addresses associated with a host name are collected in an array which is returned. The IP addresses are stored as character strings in dotted notation. These strings can be passed to INetConnectIP() for establishing a connection to a host.

Note: INetGetHosts() function is not thread safe by design. A programmer must be sure not to call it simultaneousley at the same time in two different threads, and not to use it at the same time as InetConnect(). If such a simultaneous call cannot be avoided. a call to INetGetHosts() and/or INetConnect() must be protected with a mutex.

Info

See also:	INetAddress(), INetConnect(), INetGetAlias()
Category:	Internet functions, Sockets functions, xHarbour extensions
Source:	vm\INet.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example dispays the IP addresses of a local and a remote host.

   PROCEDURE Main
      LOCAL aIP

      INetInit()

      aIP := INetGetAlias( "www.xHarbour.com" )

      ? ValToPrg( aIP )          // result: { "193.239.210.10" }

      aIP := INetGetAlias( "localhost" )

      ? ValToPrg( aIP )          // result: { "127.0.0.1" }

      INetCleanUp()
   RETURN
```

INetGetPeriodCallback()

Queries a callback associated with a socket.

Syntax

```
INetGetPeriodCallback( <pSocket> ) --> aExecutableArray
```

Arguments

<pSocket>

This is a pointer to a socket as returned by INetConnect() or INetAccept().

Return

The function returns the callback information associated with *<pSocket>*, or NIL if no callback is set.

Description

This function queries the callback information from a socket and, if present, returns it as an executable array. The callback must be previously set with INetSetPeriodCallback().

Info

See also:	HB_ExecFromArray(), INetClearPeriodCallback(), INetCreate(), INetGetTimeout(), INetSetPeriodCallback(), INetSetTimeout()
Category:	Internet functions, Sockets functions, xHarbour extensions
Source:	vm\INet.c
LIB:	xhb.lib
DLL:	xhbdll.dll

INetGetTimeout()

Queries a timout value for a socket.

Syntax

```
INetGetTimeout( <pSocket> ) --> nMilliSecs
```

Arguments

<pSocket>

This is a pointer to a socket as returned by INetConnect() or INetAccept().

Return

The function returns a timeout value in milliseconds as a numeric value. If not set, -1 is returned. This value stands for "infinity".

Description

Function INetGetTimeout() is used to query a timeout value set with INetSetTimeout() for a socket.

Info

See also:	INetClearTimeout(), INetCreate(), INetGetTimeout(), INetSetPeriodCallback(), INetSetTimeout()
Category:	Internet functions, Sockets functions, xHarbour extensions
Source:	vm\INet.c
LIB:	xhb.lib
DLL:	xhbdll.dll

INetInit()

Initializes the sockets subsystem.

Syntax

```
INetInit() --> NIL
```

Return

The return value is always NIL.

Description

Function INetInit() must be called to initialize the sockets subsystem and allocate all required memory resources. It is recommended to place one call to INetInit() at the begin of a program that uses sockets. After the sockets subsystem is initialized, function INetConnect() or INetAccept() can be called to establish a sockets connection.

Note: call function INetCleanup() to release memory resources for the sockets subsystem, when sockets are no longer needed.

Info

See also:	INetCreate(), INetConnect(), INetAccept(), INetCleanup()
Category:	Internet functions, Sockets functions, xHarbour extensions
Source:	vm\INet.c
LIB:	xhb.lib
DLL:	xhbdll.dll

INetPort()

Determines the port number a socket is connected to.

Syntax

```
INetPort( <pSocket> ) --> nPortNumber
```

Arguments

<pSocket>

This is a pointer to a socket.

Return

The function returns the port number of a connected socket as a numeric value.

Description

Function INetPort() is of informational character. It returns the port number a socket is connected to. The port number must be specified when connecting a socket to a remote station.

Info

See also: INetAccept(), INetAddress(), INetConnect(), INetDGram(), INetGetHosts()
Category: Internet functions, Sockets functions, xHarbour extensions
Source: vm\INet.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays the internet address of www.xHarbour.com

   PROCEDURE Main
      LOCAL pSocket

      INetInit()

      pSocket := INetConnect( "www.xharbour.com", 80 )

      ? INetPort( pSocket )   // result: 80

      INetClose( pSocket )
      INetCleanUp()
   RETURN
```

INetRecv()

Reads data from a socket.

Syntax

```
INetRecv( <pSocket>, @<cBuffer>, [<nBytes>] ) --> nBytesRead
```

Arguments

`<pSocket>`

This is a pointer to a socket as returned by INetConnect() or INetAccept().

`<cBuffer>`

A memory variable holding a character string must be passed by reference to INetRecv(). It must have at least *<nBytes>* characters.

`<nBytes>`

This is a numeric value specifying the number of bytes to transfer from the socket into the memory variable *<cBuffer>*. It defaults to Len(*<cBuffer>*).

Return

The function returns a numeric value indicating the number of bytes read from the socket. If the return value is smaller than Len(*<cBuffer>*), either no more bytes are available, or a network error ocurred. This can be identified using function INetErrorCode().

Description

Function INetRecv() blocks the current thread, reads *<nBytes>* bytes from the socket *<pSocket>* and copies them into a memory variable. To accomplish this, INetRecv() must receive a string buffer *<cBuffer>*, large enough to receive the number of bytes specified with *<nBytes>*. Since the bytes are copied into a memory variable, the buffer variable must be passed by reference to INetRecv().

Note: there is no guarantee that INetRecv() fills *<cBuffer>* with the specified number of bytes. The function is blocking until either no more data is available, the connection is lost or the socket is closed, whichever comes first.

Use INetRecvAll() to block the current thread until the whole *<cBuffer>* is filled or *<nBytes>* bytes are read.

Info

See also:	INetAccept(), INetConnect(), INetErrorCode(), INetRecvAll(), INetRecvLine(), INetSend(), INetSendAll()
Category:	Internet functions, Sockets functions, xHarbour extensions
Source:	vm\INet.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example outlines a simple internet communication with a HTTP server

   PROCEDURE Main
      LOCAL pSocket, cBuffer, nBytes, cRequest, cResponse

      CLS

      // initialize sockets system and connect to server
      INetInit()
```

```
   pSocket := INetConnect( "www.xharbour.com", 80 )

   IF INetErrorCode( pSocket ) <> 0
      ? "Socket error:", INetErrorDesc( pSocket )
      INetCleanUp()
      QUIT
   ENDIF

   // send HTTP request to server
   cRequest := "GET / HTTP/1.1"                + INetCRLF() + ;
               "Host: www.xharbour.com"        + INetCRLF() + ;
               "User-Agent: HTTP-Test-Program" + INetCRLF() + ;
                INetCRLF()

   nBytes   := INetSend( pSocket, cRequest )
   cResponse:= ""

   // get HTTP response from server
   DO WHILE nBytes > 0
      cBuffer   := Space( 4096 )
      nBytes    := INetRecv( pSocket, @cBuffer )
      cResponse += Left( cBuffer, nBytes )
   ENDDO

   // disconnect and cleanup memory
   INetClose( pSocket )
   INetCleanUp()

   // save response
   Memowrit( "xharbour.txt", cResponse )
   ? "Data written to file: xHarbour.txt"
RETURN
```

INetRecvAll()

Reads all data from a socket.

Syntax

```
INetRecvAll( <pSocket>, @<cBuffer>, [<nBytes>] ) --> nBytesRead
```

Arguments

`<pSocket>`

This is a pointer to a socket as returned by INetConnect() or INetAccept().

`<cBuffer>`

A memory variable holding a character string must be passed by reference to INetRecvAll(). It must have at least *<nBytes>* characters.

`<nBytes>`

This is a numeric value specifying the number of bytes to transfer from the socket into the memory variable *<cBuffer>*. It defaults to Len(*<cBuffer>*).

Return

The function returns a numeric value indicating the number of bytes read from the socket. If the return value is smaller than Len(*<cBuffer>*), either no more bytes are available, or a network error ocurred. This can be identified using function INetErrorCode().

Description

Function INetRecvAll() does exactly the same a INetRecv(), but additionally blocks the current thread until the amount of *<nBytes>* is read from the socket, or the socket is prematurely closed.

Info

See also:	INetAccept(), INetConnect(), INetErrorCode(), INetRecv(), INetRecvLine(), INetSend(), INetSendAll()
Category:	Internet functions, Sockets functions, xHarbour extensions
Source:	vm\INet.c
LIB:	xhb.lib
DLL:	xhbdll.dll

INetRecvEndblock()

Reads one block of data until an end-of-block marker is detected.

Syntax

```
INetRecvEndblock( <pSocket>     , ;
                 [<cEndOfBlock>], ;
                [@<nBytesRead>] , ;
                 [<nMaxLength>] , ;
                 [<nIncrement>]   ) --> cBlock
```

Arguments

<pSocket>

This is a pointer to a socket as returned by INetConnect() or INetAccept().

<cEndOfBlock>

Optionally, a character string can be passed specifying the end-of-block marker. It defaults to the return value of INetCRLF().

<nBytesRead>

If passed by reference, this parameter is assigned the number of bytes actually read from the socket, including the trailing end-of-block characters. This is Len(cBlock)+Len(*<cEndOfBlock>*).

<nMaxLength>

This is a numeric value specifying the maximum number of bytes to read from the socket for detecting an end-of-block character sequence.

<nIncrement>

This is an optional numeric value specifying the number of bytes to increment memory allocation while the function reads data from *<pSocket>* and no end-of-block marker has arrived. By default, memory allocation is increased in steps of 80 bytes.

Return

The function returns a character string read from the socket without trailing end-of-block marker. If an error occurs, the return value is NIL and a socket error is set.

Description

INetRecvEndblock() blocks the current thread until an end-of-block character sequence is read from *<pSocket>*. Since it is unknown how many data arrives before the end-of-block marker, the function incrementally increases memory allocation by *<nIncrement>* bytes until the end-of-block marker is detected, or until *<nMaxLength>* bytes are read. When no end-of-block marker is detected, or when the socket is prematurely closed, the function returns NIL.

Note: if *<cEndOfBlock>* is omitted, INetRecvEndblock() works exactly like INetRecvLine().

Info

See also:	INetAccept(), INetConnect(), INetErrorCode(), INetRecv(), INetRecvLine(), INetSend(), INetSendAll()
Category:	Internet functions, Sockets functions, xHarbour extensions
Source:	vm\INet.c
LIB:	xhb.lib
DLL:	xhbdll.dll

INetRecvLine()

Reads one line of data until CRLF is detected.

Syntax

```
INetRecvLine( <pSocket>    , ;
             [@<nBytesRead>], ;
             [<nMaxLength>] , ;
             [<nIncrement>]   ) --> cLine
```

Arguments

<pSocket>

This is a pointer to a socket as returned by INetConnect() or INetAccept().

<nBytesRead>

If passed by reference, this parameter is assigned the number of bytes actually read from the socket, including the trailing new-line characters. This is Len(cLine)+2.

<nMaxLength>

This is a numeric value specifying the maximum number of bytes to read from the socket for detecting a new-line character pair (CRLF).

<nIncrement>

This is an optional numeric value specifying the number of bytes to increment memory allocation while the function reads data from *<pSocket>* and no CRLF has arrived. By default, memory allocation is increased in steps of 80 bytes.

Return

The function returns a character string read from the socket without trailing new-line characters (CRLF). If an error occurs, the return value is NIL and a socket error is set.

Description

INetRecvLine() blocks the current thread until a new-line character sequence is read from *<pSocket>*. Since it is unknown how many data arrives before CRLF, the function incrementally increases memory allocation by *<nIncrement>* bytes until CRLF is detected, or until *<nMaxLength>* bytes are read. When no CRLF is detected, or when the socket is prematurely closed, the function returns NIL.

Info

See also:	INetAccept(), INetConnect(), INetErrorCode(), INetRecv(), INetRecvEndBlock(), INetSend(), INetSendAll()
Category:	Internet functions, Sockets functions, xHarbour extensions
Source:	vm\INet.c
LIB:	xhb.lib
DLL:	xhbdll.dll

INetSend()

Sends data to a socket.

Syntax

```
INetSend( <pSocket>, <cData>, [<nBytes>] ) --> nBytesSent
```

Arguments

<pSocket>

This is a pointer to a socket as returned by INetConnect() or INetAccept().

<cData>

This is a character string holding the data to send to the socket. It must have at least <*nBytes*> characters.

<nBytes>

This is a numeric value specifying the number of bytes to transfer to the socket. It defaults to Len(<*cData*>).

Return

The function returns a numeric value indicating the number of bytes sent to the socket. If the return value is smaller than <*nBytes*>, either no more bytes can be transmitted, or a network error ocurred. This can be identified using function INetErrorCode().

Description

Function INetSend() sends the number of <*nBytes*> bytes of <*cData*> from the local computer to the connected socket <*pSocket*>.

If the function succeds, it returns the number of bytes that are actually transmitted. If the socket is closed while sending data, the return value is 0, and -1 on error.

Note: there is no guarantee that all the data requested to be sent is actually sent to the socket. Therefore, the return value should be compared with the <*nBytes*> and, if a smaller value is returned, INetSend() should be called iteratively to transmit the remaining portions of <*cData*> until all data is sent.

Alternatively, data can be sent with the INetSendAll() function. It ensures all data is transmitted before the function returns.

Info

See also: INetAccept(), INetConnect(), INetErrorCode(), INetRecv(), INetRecvAll(), INetRecvLine(), INetSendAll()
Category: Internet functions, Sockets functions, xHarbour extensions
Source: vm\INet.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example outlines a simple internet communication with a HTTP server
// The HTTP request is sent with INetSend(), and the server response is
// collected with INetRecv().

   PROCEDURE Main
      LOCAL pSocket, cBuffer, nBytes, cRequest, cResponse
```

```
   CLS

   // initialize sockets system and connect to server
   INetInit()
   pSocket := INetConnect( "www.xharbour.com", 80 )

   IF INetErrorCode( pSocket ) <> 0
      ? "Socket error:", INetErrorDesc( pSocket )
      INetCleanUp()
      QUIT
   ENDIF

   // send HTTP request to server
   cRequest := "GET / HTTP/1.1"                + INetCRLF() + ;
               "Host: www.xharbour.com"        + INetCRLF() + ;
               "User-Agent: HTTP-Test-Program" + INetCRLF() + ;
                INetCRLF()

   nBytes   := INetSend( pSocket, cRequest )
   cResponse:= ""

   // get HTTP response from server
   DO WHILE nBytes > 0
      cBuffer   := Space( 4096 )
      nBytes    := INetRecv( pSocket, @cBuffer )
      cResponse += Left( cBuffer, nBytes )
   ENDDO

   // disconnect and cleanup memory
   INetClose( pSocket )
   INetCleanUp()

   // save response
   Memowrit( "xharbour.txt", cResponse )
   ? "Data written to file: xHarbour.txt"
RETURN
```

INetSendAll()

Sends all data to a socket.

Syntax

```
INetSendAll( <pSocket>, <cData>, [<nBytes>] ) --> nBytesSent
```

Arguments

<pSocket>

This is a pointer to a socket as returned by INetConnect() or INetAccept().

<cData>

This is a character string holding the data to send to the socket. It must have at least *<nBytes>* characters.

<nBytes>

This is a numeric value specifying the number of bytes to transfer to the socket. It defaults to Len(*<cData>*).

Return

The function returns a numeric value indicating the number of bytes sent to the socket. If the return value is smaller than *<nBytes>*, either no more bytes can be transmitted, or a network error ocurred. This can be identified using function INetErrorCode().

Description

Function INetSendAll() works exactly like INetSend(), except for not returning until all data is sent, the socket is prematurely closed, or a network error occurs, whichever comes first.

Info

See also:	INetAccept(), INetConnect(), INetErrorCode(), INetRecv(), INetRecvAll(), INetRecvLine(), INetSend()
Category:	Internet functions, Sockets functions, xHarbour extensions
Source:	vm\INet.c
LIB:	xhb.lib
DLL:	xhbdll.dll

INetServer()

Creates a server side socket.

Syntax

```
INetServer( <nPort>          , ;
           [<pRawSocket>]    , ;
           [<cIPAddress>]    , ;
           [<nMaxConnections>]) --> pServerSocket
```

Arguments

<nPort>

A numeric value must specify the port number the server socket is created for. Port numbers below 1024 serve special purposes. For example, the default port number for HTTP servers is 80. Refer to rfc1700 for a list of assigned or reserved ports.

<pRawSocket>

Optionally, a raw socket returned from INetCreate() can be passed. It is then configured as a server side socket.

<cIPAddress>

If specified, this parameter must contain the IP address of a client station in dotted notation. The server socket then accepts connection requests only from this address. If *<cIPAddress>* is omitted, connections from any IP address are accepted.

<nMaxConnections>

This is a numeric value specifying the maximum number of client requests for a connection that may be queued before a client receives a "server is busy/try later" response. The default value is 10, which is sufficient for most situations.

Return

The function returns a server side socket configured for listening to connection requests from remote client processes.

Description

Before INetServer() can be called, the sockets system must be initialized with INetInit(). INetServer() returns a socket configured for the server side of an internet communication. Unlike a client side socket, a server side socket cannot establish a connection to a remote process, but must wait until a connection request comes in. INetServer() configures a socket for this very purpose. The returned socket must then be passed to function INetAccept() which listens on the server socket until a remote process requests a connection. The return value of INetAccept() is a socket that can be used to exchange data between the current process (server) and a remote process (client).

The following examples outline the essentials how to use sockets functions for establishing a communication between a local server and remote client processes using the TCP/IP protocol.

Info

See also: INetAccept(), INetConnect(), INetRecv(), INetSend(), INetSetTimeout()
Category: Internet functions, Sockets functions, xHarbour extensions
Source: vm\INet.c
LIB: xhb.lib
DLL: xhbdll.dll

Examples

Server side

```
// This example is the server side of a TCP/IP communication between
// processes. The server socket is created with INetServer() and then
// configured with a timeout value. As a result, INetAccept() returns
// every half second when there is no connection request. This allows
// for querying the keyboard two times a second and check if the user
// pressed a key to terminate the server process.
//
// Note: the example must be built as multi-threading application
//  e.g. Xbuild -mt TestServer
//

   PROCEDURE Main
      LOCAL pServer, pClient

      CLS

      INetInit()

      // listen on port 1800
      pServer := INetServer( 1800 )

      // stop listening after .5 seconds
      INetSetTimeout( pServer, 500 )

      ? "Server up and running", pServer
      ? "Press any key to quit"

      DO WHILE Inkey(.1) == 0
         // wait for incoming connection requests
         pClient := INetAccept( pServer )

         IF INetErrorCode( pServer ) == 0
            // process client request in separate thread
            StartThread( @ServeClient(), pClient )
         ELSE
            // display some kind of "I am still alive" message
            ?? "."
         ENDIF
      ENDDO

      // make sure second thread has ended
      WaitForThreads()

      // close socket and cleanup memory
      INetClose( pServer )
      INetCleanup()
   RETURN
```

```
   PROCEDURE ServeClient( pSocket )
      LOCAL cBuffer, nBytes, cData := ""

      // display IP address of client
      ? "Serving:", INetAddress( pSocket )

      // wait until data containing CRLF has arrived
      // (this is blocking the thread)
      cData := INetRecvLine( pSocket, @nBytes )

      ? "Bytes received:", nBytes
      ? "Data  received:", cData

      cData := "Server reports: "    + ;
                LTrim(Str( nBytes )) + ;
               " bytes " + INetCRLF()

      // report to client process
      INetSend( pSocket, cData )
   RETURN
```

Client side

```
// This example is the client side of a TCP/IP communication between
// processes. The server IP address defaults to "127.0.0.1", which is
// the IP address of the local station. The user can enter data in a
// single GET. This data is transferred to the server process, and the
// response of the server process is displayed in the client process.
//
// Note: the example can be built as a single threaded application
//  e.g. Xbuild TestClient
//

   #include "Inkey.ch"

   PROCEDURE Main( cServerIPAddress )
      LOCAL pSocket, cData, nBytes

      IF Empty( cServerIPAddress )
         // IP address of the local station
         cServerIPAddress := "127.0.0.1"
      ENDIF

      INetInit()

      DO WHILE Lastkey() <> K_ESC
         CLS

         // let the user enter some data
         cData := Space(80)
         @ 2,2 SAY "Enter a string:" GET cData PICTURE "@S30"
         READ

         IF Lastkey() == K_ESC
            EXIT
         ENDIF

         cData   := Trim( cData )
         nBytes  := Len ( cData )

         // connect to server on port 1800
         pSocket := INetConnect( cServerIPAddress, 1800 )
```

```
      IF INetErrorCode( pSocket ) <> 0
         ? "Socket error:", INetErrorDesc( pSocket )
         INetCleanUp()
         QUIT
      ENDIF

      // Send data and append "new line" characters
      // This is what the "example server" expects
      INetSend( pSocket, cData + INetCRLF() )

      CLS
      ? "sent:", LTrim(Str(nBytes)), "bytes"

      // Receive the server's response
      cData := ""

      DO WHILE nBytes > 0
         cBuffer := Space( 100 )
         nBytes  := INetRecv( pSocket, @cBuffer )
         cData   += Left( cBuffer, nBytes )
      ENDDO

      // disconnect from server
      INetClose( pSocket )

      ? "received:", cData
      WAIT
   ENDDO

   INetCleanUp()
RETURN
```

INetSetPeriodCallback()

Associates callback information with a socket.

Syntax

```
INetSetPeriodCallback( <pSocket>, ;
                       <aExecutableArray> ) --> NIL
```

Arguments

<pSocket>

This is a pointer to a socket created from any socket creation function.

<aExecutableArray>

A one dimensional array holding executable data must be passed as second parameter. Refer to function HB_ExecFromArray() for more information on executable arrays.

Return

The return value is always NIL.

Description

The function associates an executable array with a socket. The executable data stored in the array is automatically processed when a blocking sockets operation is called and a timeout period defined with INetSetTimeout() has elapsed. This way, a callback function can be called while a blocking sockets operation is not complete. Blocking sockets operations are INetAccept(), INetRecv() or INetDGramRecv().

The callback function can be used to display progress information or "still alive" messages until the blocking function has returned.

Info

See also: HB_ExecFromArray(), INetGetPeriodCallback(), INetCreate(), INetSetTimeout()
Category: Internet functions, Sockets functions, xHarbour extensions
Source: vm\INet.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates how a callback can be installed before
// data is received from a HTTP server. The callback procedure Progress()
// displays a dot after each timeout period has elapsed while the program
// downloads data from the server.

   PROCEDURE Main
      LOCAL pSocket, nBytes, cRequest, cResponse

      CLS

      // initialize sockets system and connect to server
      INetInit()
      pSocket := INetConnect( "www.xharbour.com", 80 )

      IF INetErrorCode( pSocket ) <> 0
         ? "Socket error:", INetErrorDesc( pSocket )
         INetCleanUp()
         QUIT
```

```
   ENDIF

   // send HTTP request to server
   cRequest := "GET / HTTP/1.1"                + INetCRLF() + ;
               "Host: www.xharbour.com"        + INetCRLF() + ;
               "User-Agent: HTTP-Test-Program" + INetCRLF() + ;
                INetCRLF()

   nBytes   := INetSend( pSocket, cRequest )

   IF nBytes == Len( cRequest )
      // timeout is 0.1 second
      INetSetTimeout( pSocket, 10 )

      // pass callback information as an executable array
      INetSetPeriodCallback( pSocket, { @Progress() } )

      // get HTTP response from server
      cResponse := ""

      DO WHILE INetCount( pSocket) > 0
         // a rather small buffer to get some output from Progress()
         cBuffer   := Space( 512 )
         nBytes    := INetRecv( pSocket, @cBuffer )
         cResponse += Left( cBuffer, nBytes )
      ENDDO

      INetClearPeriodCallback( pSocket )
      INetClearTimeout( pSocket )
   ELSE
      cResponse := "error: " + INetErrorDesc( pSocket )
   ENDIF

   // disconnect and cleanup memory
   INetClose( pSocket )
   INetCleanUp()

   // save response
   Memowrit( "xharbour.txt", cResponse )
   ? "Data written to file: xHarbour.txt"
RETURN

PROCEDURE Progress
   // displaying a dot as progress is sufficient for the example
   ?? "."
RETURN
```

INetSetTimeout()

Sets a timout value in milliseconds for a socket.

Syntax

```
INetSetTimeout( <pSocket>, <nMilliSeconds> ) --> NIL
```

Arguments

<pSocket>

This is a pointer to a socket created from any socket creation function.

<nMilliSeconds>

A numeric value specifies the timeout period in milliseconds.

Return

The return value is always NIL.

Description

INetSetTimeout() sets the timeout value for a socket, after which a blocking sockets function returns. If the blocking sockets function is not completed withing the timeout period, it returns, and a sockets error of -1 is set.

Refer to function INetSetPeriodCallback() for an example of taking advantage of a timeout value.

Note: high-level sockets functions, like INetRecvAll(), internally call their low-level counterpart INetRecv() multiple times until all data is retrieved. The timeout value is applied to all individual INetRecv() calls, not for the entire INetRecvAll() call.

Info

See also: INetClearPeriodCallback(), INetClearTimeout(), INetCreate(), INetSetPeriodCallback()
Category: Internet functions, Sockets functions, xHarbour extensions
Source: vm\INet.c
LIB: xhb.lib
DLL: xhbdll.dll

Infinity()

Returns the largest number.

Syntax

```
Infinity( [<l32BitOS>] ) --> nLargestNumber
```

Arguments

<l32BitOS>

If .T. (true) is passed, the function returns the largest possible number for a 32-bit operating system. The default value is .F. (false).

Return

The function returns the largest possible number that can be represented in binary form. By default, the return value is the largest number for a 16-bit operating system (compatibility). The largest possible number for a 32-bit operating system is returned when .T. (true) is passed.

Info

See also:	Exponent(), Mantissa()
Category:	CT:NumBits, Numbers and Bits
Source:	ct\num1.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Inkey()

Retrieves a character from the keyboard buffer or a mouse event.

Syntax

```
Inkey( [<nWaitSeconds>] [,<nEventMask>] ) --> nInkeyCode
```

Arguments

`<nWaitSeconds>`

A numeric value specifying the number of seconds to wait for a key stroke or a mouse event until Inkey() returns. The value 0 instructs Inkey() to wait forever until a key is pressed or a mouse event occurs. When *<nWaitSeconds>* is omitted, the function returns immediately, even if no key stroke or mouse event is pending.

`<nEventMask>`

A numeric value specifying the type of events Inkey() should recognize. #define constants from INKEY.CH must be used for *<nEventMask>*. They are listed below:

Constants for *<nEventMask>*

Constant	Value	Events returned by Inkey()
INKEY_MOVE	1	Mouse pointer moved
INKEY_LDOWN	2	Left mouse button pressed
INKEY_LUP	4	Left mouse button released
INKEY_RDOWN	8	Right mouse button pressed
INKEY_RUP	16	Right mouse button released
INKEY_MMIDDLE	32	Middle mouse button pressed
INKEY_MWHEEL	64	Mouse wheel turned
INKEY_KEYBOARD	128	Key pressed
INKEY_ALL	255	All events are returned

IF *<nEventMask>* is omitted, the current SET EVENTMASK setting is used. If this is not issued, Inkey() returns only keyboard events.

Return

The function returns a numeric value identifying the keyboard or mouse event that occured last. If no key stroke or mouse event is pending and *<nWaitSeconds>* is not set to zero, Inkey() returns zero.

Description

The Inkey() function is used to retrieve key strokes from the keyboard buffer or monitor mouse events, if *<nEventMask>* is set accordingly. Inkey() removes the key or mouse event from the internal buffer and stores it so that it can be queried later using LastKey(). The similar function NextKey() reads a key or mouse event without removing it from the internal buffer.

Inkey() can be used to interrupt program execution for a period of *<nWaitSeconds>* time while no keyboard or mouse events are pending. The return value of Inkey() is usually processed in a DO CASE or SWITCH control structure that results in an appropriate action for a key or mouse event. The INKEY.CH file contains numerous symbolic #define constants used to identify single key strokes or mouse events. It is recommended to use these constants in program code rather than the numeric value of a keyboard or mouse event.

Note that SetKey() code blocks are not evaluated by Inkey().

Info

See also: Chr(), HB_KeyPut(), Lastkey(), Nextkey(), MCol(), MRow(), SET KEY, Set()
Category: Keyboard functions, Mouse functions
Header: Inkey.ch
Source: rtl\inkey.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example changes the event mask for Inkey() to ALL events
// and displays the mouse cursor position.

   #include "Inkey.ch"

   PROCEDURE Main
      LOCAL nEvent
      CLS
      ? "Waiting for events (press ESC to quit)"
      SET EVENTMASK TO INKEY_ALL

      DO WHILE Lastkey() <> K_ESC
         nEvent := Inkey(0)
         @ 0, 0 CLEAR TO 1, MaxCol()

         IF nEvent >= K_MINMOUSE
            // display current mouse cursor position
            @ 0,1 SAY "Mouse Row:"
            ?? MRow()
            @ 1,1 SAY "Mouse Col:"
            ?? MCol()
         ELSE
            @ 0,1 SAY "Key Code:"
            ?? nEvent
         ENDIF
      ENDDO

   RETURN
```

Int()

Converts a numeric value to an integer.

Syntax

```
Int( <nNumber> ) --> nInteger
```

Arguments

`<nNumber>`

A numeric value to convert to an integer.

Return

The function returns an integer numeric value.

Description

The function Int() truncates decimal fractions of a numeric value and returns the numeric integer value.

Info

See also:	Abs(), Round()
Category:	Numeric functions
Source:	rtl\round.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows a user defined function which tests if a number
// is an integer.

   PROCEDURE Main
      ? IsInteger( 2 )                // result: .T.
      ? IsInteger( SQrt(2) )          // result: .F.
   RETURN

   FUNCTION IsInteger( nNumber )
   RETURN ( nNumber - Int(nNumber) == 0 )
```

InvertAttr()

Exchanges the foreground and background color.

Syntax

```
InvertAttr( <xColor> ) --> nInvertedAttribute
```

Arguments

<xColor>

This is either a single color value (see SetColor()), or its numeric color attribute (see ColorToN()).

Return

The function returns the color attribute of the inverted color as a numeric value

Info

See also: NtoColor(), ScreenAttr()
Category: CT:Video, Screen functions
Source: ct\color.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays a (inverted) color attribute and the
// corresponding SetColor() values.

   PROCEDURE Main
      LOCAL nColorAttr  := GetClearA()
      LOCAL nInvertAttr := InvertAttr( nColorAttr )

      CLS
      ? nColorAttr                    // result:   7
      ? nInvertAttr                   // result: 112.00

      ? NToColor( nColorAttr , .T. )  // result: W/N
      ? NToColor( nInvertAttr, .T. )  // result: N/W
   RETURN
```

InvertWin()

Exchanges the foreground and background color on the screen.

Syntax

```
InvertWin( [<nTop>]   , ;
           [<nLeft>]  , ;
           [<nBottom>], ;
           [<nRight>]   ) --> cNull
```

Arguments

`<nTop> and <nLeft>`

Numeric values indicating the screen coordinates for the upper left corner of the screen area. The default value for both parameters is zero.

`<nBottom>`

Numeric value indicating the bottom row screen coordinate. It defaults to MaxRow().

`<nRight>`

Numeric value indicating the right column screen coordinate. It defaults to MaxCol().

Return

The return value is always a null string ("").

Info

See also: InvertAttr(), SetColor()
Category: CT:Video, Screen functions
Source: ct\invrtwin.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example produces a blink effect by inverting
// color attributes in a rectangular area on the screen.

   #include "Box.ch"

   PROCEDURE Main
      SET COLOR TO W+/B
      CLS

      DispBox( 5, 5, 20, 75, B_SINGLE+ " ", "W+/R" )

      @ Row(), Col() SAY "Blink effect for 1 second press a key"
      Inkey(0)

      WaitPeriod( 100 )

      DO WHILE WaitPeriod()
         InvertWin1( 20, 5, 5, 75 )
         Inkey(.1)
      ENDDO
   RETURN
```

IsAffirm()

Converts "Yes" in a national language to a logical value.

Syntax

```
IsAffirm( <cChar> ) --> lIsYes
```

Arguments

<cChar>

This is a single character. It is compared case-insensitive with the first letter of the word "Yes" of the currently selected national language.

Return

The function returns .T. (true) if *<cChar>* is the first letter of the word "Yes" in the currently selected national language (see HB_LangSelect()). Otherwise, .F. (false) is returned.

Info

See also:	HB_LangSelect(), IsNegative()
Category:	Language specific, xHarbour extensions
Source:	rtl\natmsg.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example uses three national languages and displays the result
// of IsAffirm().

   REQUEST HB_LANG_DE          // request German language
   REQUEST HB_LANG_FR          // request French language

   PROCEDURE Main

      // English language is default: "Yes"

      ? IsAffirm( "J" )        // result: .F.
      ? IsAffirm( "O" )        // result: .F.
      ? IsAffirm( "Y" )        // result: .T.

      HB_LangSelect( "DE" )    // "Ja" == "Yes"

      ? IsAffirm( "J" )        // result: .T.
      ? IsAffirm( "O" )        // result: .F.
      ? IsAffirm( "Y" )        // result: .F.


      HB_LangSelect( "FR" )    // "Oui" == "Yes"

      ? IsAffirm( "J" )        // result: .F.
      ? IsAffirm( "O" )        // result: .T.
      ? IsAffirm( "Y" )        // result: .F.
   RETURN
```

IsAlNum()

Checks if the first character of a string is alpha-numeric.

Syntax

```
IsAlNum( <cString> ) --> lIsAlphaNumeric
```

Arguments

<cString>

A character string to check for its first character.

Return

The function returns .T. (true) when the leftmost character of a string is alpha-numeric, otherwise .F. (false).

Description

The function is used to check if a string begins either with a digit, or with an upper or lower case letter from A to Z. It returns .T. (true) when a string begins with an alpha-numeric character, and .F. (false) when it begins with any other character.

Info

See also:	IsAlpha(), IsAscii(), IsCntrl(), IsDigit(), IsGraph(), IsLower(), IsPrint(), IsPunct(), IsSpace(), IsUpper(), IsXDigit(), Lower(), Upper()
Category:	Character functions, xHarbour extensions
Source:	rtl\is.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates various results of IsAlNum()

   PROCEDURE Main

      ? IsAlNum( "ABC" )              // result: .T.
      ? IsAlNum( "xyz" )              // result: .T.
      ? IsAlNum( "123" )              // result: .T.

      ? IsAlNum( " xHarbour" )        // result: .F.
      ? IsAlNum( ".Net" )             // result: .F.
      ? IsAlNum( "-10" )              // result: .F.
      ? IsAlNum( ".5" )               // result: .F.

   RETURN
```

IsAlpha()

Checks if the first character of a string is a letter.

Syntax

```
IsAlpha( <cString> ) --> lIsAlphabetic
```

Arguments

<cString>

A character string to check for its first character.

Return

The function returns .T. (true) when the leftmost character of a string is a letter, otherwise .F. (false).

Description

The function is used to check if a string begins with an upper or lower case letter from A to Z. It returns .T. (true) when a string begins with a letter, and .F. (false) when it begins with any other character.

Info

See also: IsAlNum(), IsAscii(), IsCntrl(), IsDigit(), IsGraph(), IsLower(), IsPrint(), IsPunct(), IsSpace(), IsUpper(), IsXDigit(), Lower(), Upper()
Category: Character functions
Source: rtl\is.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates various results of IsAlpha()

   PROCEDURE Main

      ? IsAlpha( "ABC" )              // result: .T.
      ? IsAlpha( "xyz" )              // result: .T.

      ? IsAlpha( " xHarbour" )        // result: .F.
      ? IsAlpha( ".Net" )             // result: .F.

   RETURN
```

IsAscii()

Checks if the first character of a string is a 7-bit ASCII character.

Syntax

```
IsAscii( <cString> ) --> lIs7BitChar
```

Arguments

<cString>

A character string to check for its first character.

Return

The function returns .T. (true) when the leftmost character of a string belongs to the 7-bit ASCII character set, otherwise .F. (false).

Description

The function is used to check if a string begins with a character that belongs to the 7-bit ASCII character set (ASCII values 0 - 127). It returns .T. (true) when the ASCII value of the first character is smaller than 128, and .F. (false) when it begins with any other character.

Info

See also:	IsAlNum(), IsAlpha(), IsCntrl(), IsDigit(), IsGraph(), IsLower(), IsPrint(), IsPunct(), IsSpace(), IsUpper(), IsXDigit(), Lower(), Upper()
Category:	Character functions, xHarbour extensions
Source:	rtl\is.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates various results of IsAscii()

   PROCEDURE Main

      ? IsAscii( "ABC" )                // result: .T.
      ? IsAscii( "xyz" )                // result: .T.
      ? IsAscii( "123" )                // result: .T.
      ? IsAscii( " xHarbour" )          // result: .T.

      ? IsAscii( "©" )                  // result: .F.
      ? IsAscii( "Ñ" )                  // result: .F.

   RETURN
```

IsBit()

Checks whether a bit at a specified position is set.

Syntax

```
IsBit( <nInteger>|<cHex>, [<nBitPos>]) --> lBitIsSet
```

Arguments

`<nInteger>`

A numeric 32-bit integer value can be specified as first parameter.

`<cHex>`

Alternatively, the integer can be passed as a hex-encoded character string (see NumToHex()).

`<nBitPos>`

The position of the bit to test is defined as a numeric value. <*nBitPos*> defaults to 1, and must be in the range between 1 and 32.

Return

The function returns .T. (true) when the bit at position <*nBitPos*> is set, otherwise .F. (false) is returned.

Info

See also:	NumAND(), NumNOT(), NumOR(), NumXOR(), SetBit()
Category:	CT:NumBits, Numbers and Bits
Source:	ct\bit2.c
LIB:	xhb.lib
DLL:	xhbdll.dll

IsCntrl()

Checks if the first character of a string is a control character.

Syntax

```
IsCntrl( <cString> ) --> lIsCtrlChar
```

Arguments

<cString>

A character string to check for its first character.

Return

The function returns .T. (true) when the leftmost character of a string is a control character, otherwise .F. (false).

Description

The function is used to check if a string begins with a control character. It returns .T. (true) when the ASCII value of the first character is in the range 1-31 or is 127, and .F. (false) when it begins with any other character.

Info

See also:	IsAlNum(), IsAlpha(), IsAscii(), IsDigit(), IsGraph(), IsLower(), IsPrint(), IsPunct(), IsSpace(), IsUpper(), IsXDigit(), Lower(), Upper()
Category:	Character functions, xHarbour extensions
Source:	rtl\is.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates various results of IsCntrl()

   #define  CRLF    Chr(13)+Chr(10)
   #define  TAB     Chr(9)
   #define  BELL    Chr(7)

   PROCEDURE Main

      ? IsCntrl( "ABC" )              // result: .F.
      ? IsCntrl( "xyz" )              // result: .F.
      ? IsCntrl( "123" )              // result: .F.
      ? IsCntrl( " xHarbour" )        // result: .F.

      ? IsCntrl( CRLF )               // result: .T.
      ? IsCntrl( TAB )                // result: .T.
      ? IsCntrl( BELL )               // result: .T.

   RETURN
```

IsColor()

Determines if the current computer has color capability.

Syntax

```
IsColor() --> lIsColor
```

Return

The function returns .T. (true) when the current computer has color capability.

Description

IsColor() exists for compatibility reasons only. It returns always .T. (true) since today's computers have color capability.

Info

See also:	SetColor()
Source:	rtl\gx.c
LIB:	xhb.lib
DLL:	xhbdll.dll

IsDefColor()

Checks if the default color is set.

Syntax

```
IsDefColor() --> lIsDefaultColor
```

Return

The function returns .T. (true) when the default color string is set with SetColor(), otherwise .F. (false) is returned. The default color string is "W/N,N/W,N/N,N/N,N/W".

Info

See also:	ColorSelect(), SetColor()
Category:	Screen functions, Get system
Source:	rtl\tget.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

IsDigit()

Checks if the first character of a string is a digit.

Syntax

```
IsDigit( <cString> ) --> lIsDigit
```

Arguments

<cString>

A character string to check for its first character.

Return

The function returns .T. (true) when the leftmost character of a string is a digit, otherwise .F. (false).

Description

The function is used to check if a string begins with a digit. It returns .T. (true) when a string begins with a digit, and .F. (false) when it begins with any other character.

Info

See also: IsAlNum(), IsAlpha(), IsAscii(), IsCntrl(), IsGraph(), IsLower(), IsPrint(), IsPunct(), IsSpace(), IsUpper(), IsXDigit(), Lower(), Upper()
Category: Character functions
Source: rtl\is.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates various results of IsDigit()

   PROCEDURE Main

      ? IsDigit( "ABC" )              // result: .F.
      ? IsDigit( " 10" )              // result: .F.

      ? IsDigit( "10" )               // result: .T.
      ? IsDigit( ".1" )               // result: .F.
      ? IsDigit( "-1" )               // result: .F.

   RETURN
```

IsDir()

Checks if a character string contains the name of an existing directory.

Syntax

```
IsDir( <cDirSpec> ) --> lIsDirectory
```

Arguments

<cDirName>

This is a character string holding the relative or absolute name of a directory.

Return

The function is an abbreviation of IsDIrectory(). It returns .T. (true) when the directory *<cDirSpec>* exists, otherwise .F. (false) is returned.

Info

See also: IsDirectory()
Category: CT:DiskUtil, Directory functions, Disks and Drives
Source: ct\util.prg
LIB: xhb.lib
DLL: xhbdll.dll

IsDirectory()

Checks if a character string contains the name of an existing directory.

Syntax

```
IsDirectory( <cDirName> ) --> lIsDirectory
```

Arguments

<cDirName>

This is a character string holding the relative or absolute name of a directory.

Return

The function returns .T. (true) when the directory *<cDirSpec>* exists, otherwise .F. (false) is returned.

Description

The IsDirectory() function is used to check if a directory exists that matches the specification *<cDirSpec>*. This is a character string containing the absolute or relative path of a directory.

Info

See also:	Directory(), DirChange(), DirRemove(), File(), MakeDir()
Category:	Directory functions, File functions, xHarbour extensions
Source:	rtl\filehb.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates return values of IsDirectory()

   PROCEDURE Main
      ? IsDirectory( "C:" )                     // result: .T.
      ? IsDirectory( "C" )                      // result: .F.

      ? IsDirectory( "C:\xharbour" )            // result: .T.
      ? IsDirectory( "C:\xharbour\Test.prg" )   // result: .F.

      ? IsDirectory( "." )                      // result: .T.
      ? IsDirectory( ".." )                     // result: .T.

      ? CurDir()                                // result: xHarbour\samples
      ? IsDirectory( "..\bin" )                 // result: .T.
   RETURN
```

IsDisk()

Verify if a drive is ready

Syntax

```
IsDisk( <cDrive> ) --> lDriveIsReady
```

Arguments

<cDrive>

This is single character specifying the drive letter of the disk drive to check for readiness.

Return

The function returns .T. (true) if the disk drive *<cDrive>* is ready, otherwise .F. (false)

Description

This function attempts to access a drive. If the access to the drive is successful, it returns .T. (true). The function is useful for backup routines that require to test if a disk drive is ready, before starting with the backup procedure.

Note: IsDisk() uses operating system functionalities. If a disk drive exists but has no disk inserted, the operating system prompts the user for inserting a disk.

To suppress this behavior, call SetErrorMode() and pass 1 to it.

Info

See also:	Directory(), DirChange(), DirRemove(), DiskChange(), DiskName(), File(), MakeDir(), SetErrorMode()
Category:	Disks and Drives, File functions, xHarbour extensions
Source:	rtl\dirdrive.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example collects drive letters in an array for all drives
// ready to use.

   PROCEDURE Main
      LOCAL aDisk := GetReadyDrives()
      LOCAL cDisk

      FOR EACH cDisk IN aDisk
         ? cDisk
      END

      ? IsDisk( "A" )
   RETURN


   FUNCTION GetReadyDrives()
      LOCAL aDrives := {}
      LOCAL nDrive  := 1
      LOCAL nMode   := SetErrorMode(1)

      FOR nDrive := 1 TO 26
         IF IsDisk( Chr( 64 + nDrive ) )
            AAdd( aDrives, Chr( 64 + nDrive ) )
```

```
      ENDIF
   NEXT

   SetErrorMode( nMode )
RETURN aDrives
```

IsGraph()

Checks if the first character of a string is a 7-bit graphical ASCII character.

Syntax

```
IsGraph( <cString> ) --> lIs7BitGraphicalChar
```

Arguments

<cString>

A character string to check for its first character.

Return

The function returns .T. (true) when the leftmost character of a string belongs to the 7-bit ASCII character set with a graphical representation, otherwise .F. (false).

Description

The function is used to check if a string begins with a character that belongs to the 7-bit ASCII character set that has a grahpical representation (ASCII values 33 - 126). It returns .T. (true) when the ASCII value of the first character is within the range of 33 to 126, and .F. (false) when it begins with any other character.

Info

See also:	IsAlNum(), IsAlpha(), IsAscii(), IsCntrl(), IsDigit(), IsLower(), IsPrint(), IsPunct(), IsSpace(), IsUpper(), IsXDigit(), Lower(), Upper()
Category:	Character functions, xHarbour extensions
Source:	rtl\is.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example lists all 7-bit ASCII values with a graphical
// representation to a file

   PROCEDURE Main
      LOCAL i

      SET ALTERNATE TO IsGraph.txt
      SET ALTERNATE ON

      FOR i := 0 TO 255
         IF IsGraph( Chr(i) )
            ?? Chr(i), i
            ?
         ENDIF
      NEXT

   RETURN
```

IsLeap()

Checks if a Date value belongs to a leap year.

Syntax

```
IsLeap( [<dDate>] ) --> lIsLeapYear
```

Arguments

`<dDate>`

Any Date value, except for an empty date, can be passed. The default is the return value of Date().

Return

The function returns .F. (true), when *<dDate>* falls into a leap year, otherwise .F. (false) is returned.

Info

See also:	DaysInMonth(), Quarter(), Year()
Category:	CT:DateTime, Date and time
Source:	ct\dattime2.c
LIB:	xhb.lib
DLL:	xhbdll.dll

IsLocked()

Checks if a record is locked.

Syntax

```
IsLocked( [<nRecno>] ) --> lIsLocked
```

Arguments

<nRecno>

This is an optional numeric value identifying the record number of the record to test. It defaults to Recno(), which is the record number of the current record.

Return

The function returns .T. (true) when the record with the record number *<nRecno>* is currently locked by this process, otherwise .F. (false) is returned.

Description

IsLocked() tests if a database record is currently locked for write access by this process. It cannot detect, if a record lock is set by another process active on a different workstation.

Info

See also:	DbRLock(), DbRLockList(), DbUnlock(), NetRecLock()
Category:	Network functions, Screen functions, xHarbour extensions
Source:	rtl\ttable.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

IsLower()

Checks if the first character of a string is a lowercase letter.

Syntax

```
IsLower( <cString> ) --> lIsLowerCase
```

Arguments

<cString>

A character string to check for its first character.

Return

The function returns .T. (true) when the leftmost character of a string is a lowercase letter, otherwise .F. (false).

Description

The function is used to check if a string begins with a lowercase letter from "a" to "z". It returns .T. (true) when a string begins with a lowercase letter, and .F. (false) when it begins with any other character.

Info

See also:	IsAlNum(), IsAlpha(), IsAscii(), IsCntrl(), IsDigit(), IsGraph(), IsPrint(), IsPunct(), IsSpace(), IsUpper(), IsXDigit(), Lower(), Upper()
Category:	Character functions
Source:	rtl\is.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates various results of IsLower()

   PROCEDURE Main

      ? IsLower( "ABC" )               // result: .F.
      ? IsLower( "xyz" )               // result: .T.

      ? IsLower( " xHarbour" )         // result: .F.
      ? IsLower( ".net" )              // result: .F.

   RETURN
```

IsNegative()

Converts "No" in a national language to a logical value.

Syntax

```
IsNegative( <cChar> ) --> lIsNo
```

Arguments

<cChar>

This is a single character. It is compared case-insensitive with the first letter of the word "No" of the currently selected national language.

Return

The function returns .T. (true) if *<cChar>* is the first letter of the word "No" in the currently selected national language (see HB_LangSelect()). Otherwise, .F. (false) is returned.

Note: the letter "N" means "No" in many languages. "Yes" begins with different characters in many languages (see IsAffirm()).

Info

See also:	HB_LangSelect(), IsAffirm()
Category:	Language specific, xHarbour extensions
Source:	rtl\natmsg.c
LIB:	xhb.lib
DLL:	xhbdll.dll

IsPrint()

Checks if the first character of a string is a printable 7-bit ASCII character.

Syntax

```
IsGraph( <cString> ) --> lIs7BitPrintableChar
```

Arguments

<cString>

A character string to check for its first character.

Return

The function returns .T. (true) when the leftmost character of a string belongs to the 7-bit ASCII character set and is printable, otherwise .F. (false).

Description

The function is used to check if a string begins with a character that belongs to the 7-bit ASCII character set that is printable (ASCII values 32 - 126). It returns .T. (true) when the ASCII value of the first character is within the range 32 to 126, and .F. (false) when it begins with any other character.

Info

See also:	IsAlNum(), IsAlpha(), IsAscii(), IsCntrl(), IsDigit(), IsGraph(), IsLower(), IsPunct(), IsSpace(), IsUpper(), IsXDigit(), Lower(), Upper()
Category:	Character functions, xHarbour extensions
Source:	rtl\is.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example lists all printable 7-bit ASCII values to a file

   PROCEDURE Main
      LOCAL i

      SET ALTERNATE TO IsPrint.txt
      SET ALTERNATE ON

      FOR i := 0 TO 255
         IF IsPrint( Chr(i) )
            ?? Chr(i), i
            ?
         ENDIF
      NEXT

   RETURN
```

IsPrinter()

Determines if print output can be processed.

Syntax

```
IsPrinter( [<cPrinterName>] ) --> lIsPrinter
```

Arguments

<cPrinterName>

An optional character string holding the name of a particular printer driver.

Return

The function returns .T. (true) when the operating system can process print output, otherwise .F. (false) is returned.

Description

The IsPrinter() function is used to test if an application can pass print output to the operating system. This is usually the case when a print spooler is active. IsPrinter() cannot detect, if a physical printer is ready to print, since this is monitored by the operating system.

If a character string holding a particular printer name is passed, IsPrinter() returns .T. (true) if this printer is installed.

Note: under CA-Clipper (DOS), IsPrinter() checked the readiness of a physical printer. This is not possible with xHarbour's IsPrinter() function due to differences between DOS and modern operating systems.

Info

See also:	GetDefaultPrinter(), GetPrinters(), PCol(), PRow(), SET DEVICE, SET PRINTER, SetPrc()
Category:	Printer functions
Source:	rtl\isprint.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example outlines possibilities of getting printer information
// different from CA-Clipper (DOS).

   PROCEDURE Main
      LOCAL aPrinter := GetPrinters()

      ? ValToPrg( aPrinter )

      ? GetDefaultPrinter()

      ? IsPrinter()

      ? IsPrinter( aPrinter[1] )

      ? IsPrinter( "HP LaserJet" )
   RETURN
```

IsPunct()

Checks if the first character of a string is a punctuation character.

Syntax

```
IsPunct( <cString> ) --> lIsPunctuationChar
```

Arguments

<cString>

A character string to check for its first character.

Return

The function returns .T. (true) when the leftmost character of a string is a punctuation character, otherwise .F. (false).

Description

The function is used to check if a string begins with a punctuation character. It returns .T. (true) when the first character is a punctuation character, and .F. (false) when it begins with any other character.

Info

See also: IsAlNum(), IsAlpha(), IsAscii(), IsCntrl(), IsDigit(), IsGraph(), IsLower(), IsPrint(), IsSpace(), IsUpper(), IsXDigit(), Lower(), Upper()
Category: Character functions, xHarbour extensions
Source: rtl\is.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example collects all punctuation characters in a string
// and displays it.

   PROCEDURE Main
      LOCAL i, cStr := ""

      FOR i := 0 TO 255
         IF IsPunct( Chr(i) )
            cStr += Chr(i)
         ENDIF
      NEXT

      ? cStr           // result: !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
   RETURN
```

IsSameThread()

Compares two thread handles.

Syntax

```
IsSameThread( <pThreadHandle1>, ;
              <pThreadHandle2>  ) --> lIsSameThread
```

Arguments

<pThreadHandle>

These are the handles of two threads.

Return

The function returns .T. (true) when both thread handles belong to the same thread, otherwise .F. (false) is returned.

Description

The function compares the result of two expressions, or contents of two variables, and returns .T. (true) when both yield a thread handle belonging to the same thread.

Info

See also:	GetCurrentThread(), GetThreadID(), IsValidThread(), StartThread(), ThreadSleep()
Category:	Multi-threading functions, xHarbour extensions
Source:	vm\thread.c
LIB:	xhbmt.lib
DLL:	xhbmtdll.dll

Example

```
// The example shows the result of IsSameThread() for running
// and terminated threads. They are the same.

   PROCEDURE Main

      LOCAL pThread1, pThread2
      CLS

      pThread1:= StartThread( "RunInThread" )
      pThread2:= StartThread( "RunInThread" )

      ? IsSameThread( pThread1, pThread2 )  // result: .F.
      ? IsSameThread( pThread1, pThread1 )  // result: .T.

      ThreadSleep( 500 )                    // wait for threads' end

      ? IsSameThread( pThread1, pThread2 )  // result: .F.
      ? IsSameThread( pThread1, pThread1 )  // result: .T.

   RETURN

   PROCEDURE RunInThread()
      ThreadSleep( 100 )
   RETURN
```

IsSpace()

Checks if the first character of a string is a white-space character.

Syntax

```
IsSpace( <cString> ) --> lIsWhiteSpaceChar
```

Arguments

<cString>

A character string to check for its first character.

Return

The function returns .T. (true) when the leftmost character of a string is a white-space character, otherwise .F. (false).

Description

The function is used to check if a string begins with a white-space character. It returns .T. (true) when the first character is a white-space character, and .F. (false) when it begins with any other character.

Info

See also: IsAlNum(), IsAlpha(), IsAscii(), IsCntrl(), IsDigit(), IsGraph(), IsLower(), IsPrint(), IsSpace(), IsUpper(), IsXDigit(), Lower(), Upper()
Category: Character functions, xHarbour extensions
Source: rtl\is.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example collects all white-space characters in a string
// and displays their ASCII values.

   PROCEDURE Main
      LOCAL i, aSpace := {}

      FOR i := 0 TO 255
         IF IsSpace( Chr(i) )
            AAdd( aSpace, i )
         ENDIF
      NEXT

      AEval( aSpace, {|n| QOut(n) } )

      // result:
      //   9         <-- Horizontal tab
      //  10         <-- Line feed
      //  11         <-- Vertical tab
      //  12         <-- Form feed
      //  13         <-- Carriage return
      //  32         <-- Space
   RETURN
```

IsUpper()

Checks if the first character of a string is an uppercase letter.

Syntax

```
IsUpper( <cString> ) --> lIsUpperCase
```

Arguments

<cString>

A character string to check for its first character.

Return

The function returns .T. (true) when the leftmost character of a string is an uppercase letter, otherwise .F. (false).

Description

The function is used to check if a string begins with an uppercase letter from "A" to "Z". It returns .T. (true) when a string begins with an uppercase letter, and .F. (false) when it begins with any other character.

Info

See also:	IsAlNum(), IsAlpha(), IsAscii(), IsCntrl(), IsDigit(), IsGraph(), IsPrint(), IsPunct(), IsSpace(), IsUpper(), IsXDigit(), Lower(), Upper()
Category:	Character functions
Source:	rtl\is.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates various results of IsUpper()

   PROCEDURE Main

      ? IsLower( "ABC" )               // result: .T.
      ? IsLower( "xyz" )               // result: .F.

      ? IsLower( " XHARBOUR" )         // result: .F.
      ? IsLower( ".NET" )              // result: .F.

   RETURN
```

IsValidThread()

Checks if an expression is the thread handle of a running thread.

Syntax

```
IsValidThread( <xValue> ) --> lIsValid
```

Arguments

<xValue>

This is a value of any data type.

Return

The function returns .T. (true) when *<xValue>* is the thread handle of a running thread, otherwise .F. (false) is returned.

Description

Function IsValidThread() is used to detect if the passed parameter is a handle to a valid thread. A thread is valid when it is successfully started and has not ended, yet. Thus, IsValidThread() can be used to detect if a thread is "running". It cannot detect, however, if a thread is actually executing program code since "sleeping" or suspended threads waiting for Mutexes are valid, but are put "on hold".

Info

See also: GetCurrentThread(), GetThreadID(), IsSameThread(), StartThread(), ThreadSleep()
Category: Multi-threading functions, xHarbour extensions
Source: vm\thread.c
LIB: xhbmt.lib
DLL: xhbmtdll.dll

Example

```
// The example demonstrates that IsValidThread() returns only
// True when a handle to a started thread is passed.

   PROCEDURE Main
      LOCAL pThread

      ? IsValidThread( pThread )         // result: .F.

      pThread:= StartThread( "RunInThread" )

      ? IsValidThread( pThread )         // result: .T.

      JoinThread( pThread )

      ? IsValidThread( pThread )         // result: .F.
  RETURN


  PROCEDURE RunInThread()
     ? "Thread started:", GetThreadID()

     ThreadSleep( 500 )
  RETURN
```

IsXDigit()

Checks if the first character of a string is a hexadecimal digit.

Syntax

```
IsXDigit( <cString> ) --> lIsHexDigit
```

Arguments

<cString>

A character string to check for its first character.

Return

The function returns .T. (true) when the leftmost character of a string is a hexadecimal digit, otherwise .F. (false).

Description

The function is used to check if a string begins with a hexadecimal digit. It returns .T. (true) when the first character is a hexadecimal digit, and .F. (false) when it begins with any other character.

Info

See also: IsAlNum(), IsAlpha(), IsAscii(), IsCntrl(), IsDigit(), IsGraph(), IsLower(), IsPrint(), IsSpace(), IsUpper(), Lower(), Upper()
Category: Character functions, xHarbour extensions
Source: rtl\is.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example collects all hexadecimal digits in a string
// and displays it.

   PROCEDURE Main
      LOCAL i, cHex := ""

      FOR i := 0 TO 255
         IF IsXDigit( Chr(i) )
            cHex += Chr(i)
         ENDIF
      NEXT

      ? cHex      // result:  0123456789ABCDEFabcdef
   RETURN
```

JoinThread()

Suspends the current thread until a second thread has terminated.

Syntax

```
JoinThread( <pThreadHandle> ) --> NIL
```

Arguments

<pThreadHandle>

This is the handle of the thread to wait for. A thread handle is returned from function StartThread().

Return

The return value is always NIL.

Description

Function JoinThread() suspends procgram execution of the current thread and waits until a second thread has ended. JoinThread() can thus be used to synchronize the current thread with the end of a second one.

If the thread *<pThreadHandle>* is already terminated, the function returns immediately and does not wait.

Info

See also: GetCurrentThread(), GetThreadID(), HB_MutexCreate(), IsSameThread(), StartThread(), StopThread(), ThreadSleep()
Category: Multi-threading functions, xHarbour extensions
Source: vm\thread.c
LIB: xhbmt.lib
DLL: xhbmtdll.dll

Example

```
// The example calculates statistical data from a set of numbers.
// The calculation is split into two threads, each of which compute
// temporary results required for the final caclulation. The main thread
// display the final data and must wait until the second thread has
// finished its calculation.

   PROCEDURE Main
      LOCAL hResult := {=>}
      LOCAL aPrime  := { 1, 3, 5, 7, 9, 11, 13, 17, 19, 23, 29 }
      LOCAL nCount  := Len( aPrime )
      LOCAL pThread
      LOCAL nNumber
      LOCAL nVariance

      hResult["SUM"]       := 0
      hResult["SUMSQUARE"] := 0

      pThread := StartThread( "SumSquare", hResult, aPrime )

      FOR EACH nNumber IN aPrime
         hResult["SUM"] += nNumber
      NEXT
```

```
   // Make sure the second thread has completed its calculation
   JoinThread( pThread )

   nVariance := nCount * hResult["SUMSQUARE"] - hResult["SUM"] ^ 2
   nVariance /= nCount * (nCount-1)

   ? "Count   :", nCount
   ? "Sum     :", hResult["SUM"]
   ? "Sum^2   :", hResult["SUMSQUARE"]
   ? "Average :", hResult["SUM"] / nCount
   ? "Std.Dev.:", Sqrt( nVariance )
RETURN


PROCEDURE SumSquare( hResult, aNumbers )
   LOCAL nNumber

   FOR EACH nNumber IN aNumbers
      hResult["SUMSQUARE"] += ( nNumber ^ 2 )
   NEXT
RETURN
```

JustLeft()

Left justifies characters in a character string.

Syntax

```
JustLeft( <cString>, [<xChar>] ) --> cResult
```

Arguments

<cString>

This is the string to process.

<xChar>

This is a single character or its numeric ASCII code that is searched at the beginning of *<cString>*. It defaults to a blank space (Chr(32)).

Return

The function searches *<xChar>* at the beginning of *<cString>* and moves all *<xChar>* from the beginning to the end. The modified string is returned.

Info

See also: Center(), JustRight()
Category: CT:String manipulation, Character functions
Source: ct\justify.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays results of JustLeft()

   PROCEDURE Main
      LOCAL cStr1 := "   xHarbour"
      LOCAL cStr2 := "....ABCD"

      ? ">"+JustLeft( cStr1 )+"<"       // result: >xHarbour   <

      ? ">"+JustLeft( cStr2, "." )+"<"  // result: >ABCD....<
   RETURN
```

JustRight()

Right justifies characters in a character string.

Syntax

```
JustRight( <cString>, [<xChar>] ) --> cResult
```

Arguments

`<cString>`

This is the string to process.

`<xChar>`

This is a single character or its numeric ASCII code that is searched at the end of *<cString>*. It defaults to a blank space (Chr(32)).

Return

The function searches *<xChar>* at the end of *<cString>* and moves all *<xChar>* from the end to the beginning. The modified string is returned.

Info

See also:	Center(), JustLeft()
Category:	CT:String manipulation, Character functions
Source:	ct\justify.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays results of JustRight()

   PROCEDURE Main
      LOCAL cStr1 := "xHarbour   "
      LOCAL cStr2 := "ABCD...."

      ? ">"+JustRight( cStr1 )+"<"       // result: >   xHarbour<

      ? ">"+JustRight( cStr2, "." )+"<"  // result: >....ABCD<
   RETURN
```

KbdStat()

Determines the state of special keys like Ctrl or Shift keys.

Syntax

```
KbdStat() --> nKeyState
```

Return

The function returns a numeric value indicating special keys pressed when the function is called. Each bit of the returned value represents a special key:

Bits set for special keys

Bit	Key pressed
1	Shift key
3	Ctrl key
4	Alt key
5	Scroll Lock ON
6	Num Lock ON
7	Caps Lock ON
8	Insert ON

Info

Category: CT:Miscellaneous, Keyboard functions, Miscellaneous functions
Source: ct\misc3.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays the result of KbdStat() and
// the individual bits set

   #include "Inkey.ch"

   PROCEDURE Main()
      LOCAL nKey := 0
      LOCAL nKbdStat

      CLS
      SET CURSOR OFF

      DO WHILE nKey <>  K_ESC
         nKey     := Inkey(0.1)

         IF nKey == K_INS
            Set( _SET_INSERT, .NOT. Set( _SET_INSERT ) )
         ENDIF

         nKbdStat := Kbdstat()

         @ 5, 20 SAY "Key state: " + CStr( nKbdStat )
         @ 6, 20 SAY " Bits set: " + NToC( nKbdStat, 2, 8, "0" )
         @ 7, 20 SAY "           87654321"
      ENDDO
   RETURN
```

KeySec()

Starts a timer to write a key code into the keyboard buffer.

Syntax

```
Keysec( <nKey>     , ;
        <nTime>    , ;
       [<nCounter>], ;
       [<lMode>]     ) --> lSuccess
```

Arguments

<nKey>

This is the numeric key code to write into the keyboard buffer. #define constants listed in the Inkey.ch file are used for *<nKey>*.

<nTime>

This is a numeric value specifying the time after which the character is written into the keyboard buffer. Positive values indicate the time in seconds, while negative values are interpreted as multiples of 1/18.2 seconds.

<nCounter>

This parameter specifies the number of times *<nKey>* should be placed into the keyboard buffer before the timer ends. The default value is 1.

<lMode>

This parameter defaults to .F. (false). When .T. (true) is passed, the timer is reset when a key is pressed.

Return

The function returns .T. (true) when the timer is successfully activated, otherwise .F. (false).

Note: when the function is called without a parameter, the timer is deactivated.

Info

See also: HB_IdleAdd(), KEYBOARD, KeyTime()
Category: CT:Miscellaneous, Keyboard functions, Miscellaneous functions
Source: ct\keysec.prg
LIB: xhb.lib
DLL: xhbdll.dll

KeyTime()

Writes a key code into the keyboard buffer at a specified time.

Syntax

```
Keytime( <nKey>, <cTime> ) --> lSuccess
```

Arguments

`<nKey>`

This is the numeric key code to write into the keyboard buffer. #define constants listed in the Inkey.ch file are used for *<nKey>*.

`<cTime>`

This is a Time() formatted character string indicating the time at which the key should be written to the keyboard buffer.

When the hour is set to 99, the key is written every full hour. A time string of "15:99:00" will place a key every minute between 3:00pm and 3:59pm.

Return

The function returns .T. (true) when the timer is successfully activated, otherwise .F. (false).

Note: when the function is called without a parameter, the timer is deactivated.

Info

See also:	HB_IdleAdd(), KEYBOARD, KeySec()
Category:	CT:Miscellaneous, Keyboard functions, Miscellaneous functions
Source:	ct\keytime.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

KillAllThreads()

Kills all running threads except for the main thread.

Syntax

```
KillAllThreads() --> NIL
```

Return

The return value is always NIL.

Description

Function KillAllThreads() is provided for "emergency recovery", such as dead locks between threads, and may only be called in the main thread of an xHarbour application. The function sends a thread termination request to the operating system for all threads but the main thread, and returns immediately. It does not guarantee that threads have actually terminated when the function returns. Use WaitForThreads() to make sure all threads have terminated.

Important: if a call to KillAllThreads() becomes necessary, the program logic of the multi-threading application should be carefully revised. It is an indication for an error in the usage of multiple threads.

Info

See also:	HB_MutexCreate(), KillThread(), StartThread(), StopThread(), ThreadSleep(), WaitForThreads()
Category:	Multi-threading functions, xHarbour extensions
Source:	vm\thread.c
LIB:	xhbmt.lib
DLL:	xhbmtdll.dll

KillThread()

Kills a running thread.

Syntax

```
KillThread( <pThreadHandle> ) --> NIL
```

Arguments

<pThreadHandle>

This is the handle of the thread to terminate. A thread handle is returned from function StartThread().

Return

The return value is always NIL.

Description

Function KillThread() is provided for "emergency recovery", such as dead locks between threads. The function sends a thread termination request to the operating system for the specified thread and returns immediately. It does not guarantee that the thread has actually terminated when the function returns. Use JoinThread() to make sure the thread has terminated.

Important: if a call to KillThread() becomes necessary, the program logic of the multi-threading application should be carefully revised. It is an indication for an error in the usage of multiple threads. To stop a thread from outside, use StopThread(). StopThread() works synchronously while KillThread() works asynchronously.

Info

See also:	HB_MutexCreate(), JoinThread(), KillAllThreads(), StartThread(), StopThread(), ThreadSleep()
Category:	Multi-threading functions, xHarbour extensions
Source:	vm\thread.c
LIB:	xhbmt.lib
DLL:	xhbmtdll.dll

KSetCaps()

Queries or changes the status of the Caps lock key

Syntax

```
KSetcaps( [<lNewMode>] ) --> lOldMode
```

Arguments

<lNewMode>

This is an optional logical value defining the new status for the Caps lock key. .T. (true) activates the Caps lock key, and .F. (false) deactivates it.

Return

The function returns the previous activation mode of the Caps lock key as a logical value.

Info

See also: KbdStat(), KSetIns(), KSetNum(), KSetScroll()
Category: CT:Settings, Keyboard functions
Source: ct\keyset.c
LIB: xhb.lib
DLL: xhbdll.dll

KSetIns()

Queries or changes the status of the Insert/Overwrite key

Syntax

```
KSetIns( [<lNewMode>] ) --> lOldMode
```

Arguments

<lNewMode>

This is an optional logical value defining the new status for the Insert/Overwrite key. .T. (true) activates the Insert mode, and .F. (false) deactivates it.

Return

The function returns the previous activation mode of the Insert/Overwrite key as a logical value.

Info

See also:	KbdStat(), KSetCaps(), KSetNum(), KSetScroll()
Category:	CT:Settings, Keyboard functions
Source:	ct\keyset.c
LIB:	xhb.lib
DLL:	xhbdll.dll

KSetNum()

Queries or changes the status of the Num lock key

Syntax

```
KSetNum( [<lNewMode>] ) --> lOldMode
```

Arguments

<lNewMode>

This is an optional logical value defining the new status for the Num lock key. .T. (true) activates the Num lock key, and .F. (false) deactivates it.

Return

The function returns the previous activation mode of the Num lock key as a logical value.

Info

See also:	KbdStat(), KSetCaps(), KSetIns(), KSetScroll()
Category:	CT:Settings, Keyboard functions
Source:	ct\keyset.c
LIB:	xhb.lib
DLL:	xhbdll.dll

KSetScroll()

Queries or changes the status of the Scroll lock key

Syntax

```
KSetScroll( [<lNewMode>] ) --> lOldMode
```

Arguments

<lNewMode>

This is an optional logical value defining the new status for the Scroll lock key. .T. (true) activates the Scroll lock key, and .F. (false) deactivates it.

Return

The function returns the previous activation mode of the Scroll lock key as a logical value.

Info

See also:	KbdStat(), KSetCaps(), KSetIns(), KSetNum()
Category:	CT:Settings, Keyboard functions
Source:	ct\keyset.c
LIB:	xhb.lib
DLL:	xhbdll.dll

L2bin()

Converts a numeric value to a signed long binary integer (4 bytes).

Syntax

```
L2Bin( <nNumber> ) --> cInteger
```

Arguments

`<nNumber>`

A numeric value in the range of -(2^31) to +(2^31) - 1.

Return

The function returns a four-byte character string representing a 32-bit signed long binary integer .

Description

L2Bin() is a binary conversion function that converts a numeric value (Valtype()=="N") to a four-byte binary number (Valtype()=="C").

The range for the numeric return value is determined by a signed long integer. If <*nNumber*> is outside this range, a runtime error is raised.

L2bin() is the inverse function of Bin2L().

Info

See also:	Asc(), Bin2I(), Bin2L(), Bin2U(), Bin2W(), Chr(), I2Bin(), U2Bin(), W2Bin()
Category:	Binary functions, Conversion functions
Source:	rtl\binnum.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates return values of L2Bin().

   PROCEDURE Main
      LOCAL c

      c := L2Bin(0)
      ? Asc(c[1]), Asc(c[2]), Asc(c[3]), Asc(c[4])
      //      0          0          0          0

      c := L2Bin(1)
      ? Asc(c[1]), Asc(c[2]), Asc(c[3]), Asc(c[4])
      //      1          0          0          0

      c := L2Bin(-1)
      ? Asc(c[1]), Asc(c[2]), Asc(c[3]), Asc(c[4])
      //    255        255        255        255

      c := L2Bin(256)
      ? Asc(c[1]), Asc(c[2]), Asc(c[3]), Asc(c[4])
      //      0          1          0          0

      c := L2Bin(-256)
      ? Asc(c[1]), Asc(c[2]), Asc(c[3]), Asc(c[4])
      //      0        255        255        255
```

```
   c := L2Bin(2147483647)
   ? Asc(c[1]), Asc(c[2]), Asc(c[3]), Asc(c[4])
   //    255        255        255        127

   c := L2Bin(-2147483648)
   ? Asc(c[1]), Asc(c[2]), Asc(c[3]), Asc(c[4])
   //      0          0          0        128
RETURN
```

LastDayoM()

Returns the number of days in a month.

Syntax

```
LastDayoM( <dDate>|<nMonth> ) --> nDaysInMonth
```

Arguments

<dDate>

Any Date value, except for an empty date, can be passed. The default is the return value of Date().

<nMonth>

Alternatively, the numeric month between 1 and 12 can be passed.

Return

The function returns the number of days of the specified month as a numeric value.

Info

See also:	DaysInMonth(), EoM()
Category:	CT:DateTime, Date and time
Source:	ct\dattime2.c
LIB:	xhb.lib
DLL:	xhbdll.dll

LastKey()

Returns the last Inkey() code retrieved.

Syntax

```
LastKey( [<nEventMask>] ) --> nLastInkeyCode
```

Arguments

<nEventMask>

A numeric value specifying the type of events LastKey() should retrieve. #define constants from INKEY.CH must be used for *<nEventMask>*. They are listed below:

Constants for *<nEventMask>*

Constant	Value	Events returned by LastKey()
INKEY_MOVE	1	Mouse pointer moved
INKEY_LDOWN	2	Left mouse button pressed
INKEY_LUP	4	Left mouse button released
INKEY_RDOWN	8	Right mouse button pressed
INKEY_RUP	16	Right mouse button released
INKEY_MMIDDLE	32	Middle mouse button pressed
INKEY_MWHEEL	64	Mouse wheel turned
INKEY_KEYBOARD	128	Key pressed
INKEY_ALL	255	All events are returned

IF *<nEventMask>* is omitted, the current SET EVENTMASK setting is used. If this is not issued, LastKey() returns only keyboard events.

Return

The function returns a numeric value identifying the last keyboard or mouse event retrieved by the Inkey() function.

Description

The LastKey() function returns the last keyboard or mouse event that was removed from the internal input buffers by the Inkey() function. LastKey() retains its value until the next pending key stroke or mouse event is removed from the internal input buffers by Inkey(). This way, the last Inkey code does not need to be preserved in a variable that must be passed to sub-routines for processing user input. Instead, user input can be queried with LastKey() until the next Inkey() call is made.

If only a subset of the Inkey codes is required, LastKey() can be passed a parameter *<nEventMask>* that filters Inkey codes retrieved from Inkey(). Only Inkey codes matching *<nEventMask>* are returned from LastKey().

Note: the file INKEY.CH contains numerous symbolic #define constants that identify different key strokes or mouse input. It is recommended to use #define constants for prosessing Inkey codes, rather than using their numeric values.

Info

See also: Chr(), Inkey(), KEYBOARD, NextKey(), SetLastKey()
Category: Keyboard functions, Mouse functions
Header: Inkey.ch
Source: rtl\inkey.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// In this example, all possible user input is recognized by
// the Inkey() function. LastKey(), however, reports only
// key strokes and a left button click. Other mouse input
// is ignored by LastKey().

   #include "Inkey.ch"

   PROCEDURE Main
      LOCAL nEvent

      CLS
      ?? "Waiting for events (press ESC to quit)"
      SET EVENTMASK TO INKEY_ALL

      DO WHILE Lastkey() <> K_ESC
         nEvent := Inkey(0)

         nEvent := LastKey( INKEY_KEYBOARD + INKEY_LUP )
         IF nEvent == 0
            LOOP
         ENDIF

         @ 0, 0 CLEAR TO 1, MaxCol()

         IF nEvent >= K_MINMOUSE
            // display current mouse cursor position
            @ 0,1 SAY "Mouse Row:"
            ?? MRow()
            @ 1,1 SAY "Mouse Col:"
            ?? MCol()
         ELSE
            @ 0,1 SAY "Key Code:"
            ?? nEvent
         ENDIF
      ENDDO

   RETURN
```

LastRec()

Returns the number of records available in a work area.

Syntax

```
LastRec() --> nRecords
```

Return

The function returns the number of records available in a work area as a numeric value. If the work area is not used or if a database is empty, the return value is zero.

Description

The LastRec() function returns the number of records available in a work area. It is the actual number of physical records stored in a database file, not the number of records that a logically visible in a work area.

Logical visibility of records can be reduced with SET FILTER, SET DELETED or SET SCOPE so that the number of records that are accessible during database navigation can be less than the number of records reported by LastRec().

Info

See also:	COUNT, Eof(), OrdKeyCount(), RecCount(), RecNo()
Category:	Database functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example illustrates LastRec() usage with and without
// alias operator

   PROCEDURE Main
      USE Customer  ALIAS Cust NEW
      USE Invoice   ALIAS Inv  NEW

      ? Alias()                 // result: INV
      ? LastRec()               // result: 12233
      ? Cust->(LastRec())       // result: 5612

      CLOSE ALL
   RETURN
```

Left()

Extracts characters from the left side of a string

Syntax

```
Left( <cString>, <nCount> ) --> cSubString
```

Arguments

<cString>

A character string to extract a substring from.

<nCount>

A numeric value specifying the number of characters to extract from the left side of *<cString>*.

Return

The function returns a character string containing Min(*<nCount>*, Len(*<cString>*)) characters.

Description

The character function Left() extracts a substring from the left side of *<cString>*. The returned substring contains the first *<nCount>* characters. If *<nCount>* is larger than Len(*<cString>*), the return value is a copy of *<cString>*.

Left() is an abbreviated form of SubStr(*<cString>*, 1, *<nCount>*) and has a counterpart Right(), which extracts a substring from the right side of a string.

Info

See also:	At(), HB_ATokens(), LTrim(), RAt(), Right(), RTrim(), Stuff(), SubStr()
Category:	Character functions
Source:	rtl\left.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example illustrates return values of function Left()

   PROCEDURE Main
      ? CDoW(Date())                  // result: Friday
      ? Left( CDoW(Date()), 3)        // result: Fri

      ? Time()                        // result: 14:45:12
      ? Left( Time(), 5)              // result: 14:45

      ? Left("xHarbour", 7)           // result: xHarbou
   RETURN
```

Len()

Returns the number of items contained in an array, hash or string

Syntax

```
Len( <aArray> | <cString> | <hHash> ) --> nCount
```

Arguments

<aArray>

An array whose number of elements is determined.

<cString>

A character string whose number of characters is determined.

<hHash>

A hash whose number of key/value pairs is determined.

Return

The function returns the number of items stored in the passed parameter as a numeric value. When the passed parameter is empty (contains no items), the return value is zero.

Description

The function Len() accepts parameters of three different data types: Array, Character string and Hash. It returns the number of items stored in the passed parameter.

Return value of Len()

Data type	Description
Array	Number of array elements in first dimension
Character string	Number of characters
Hash	Number of key/value pairs

The return value is zero, when an array has no elements, a string has no characters or a hash has no key/value pair.

Info

See also: Array(), Empty(), Hash(), LenNum(), LTrim(), RTrim()
Category: Array functions, Character functions, Hash functions
Source: rtl\len.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example shows return values of Len() with different data types

   PROCEDURE Main
      LOCAL aArray  := Directory( "*.prg" )
      LOCAL cString := "Hello World"
      LOCAL hHash   := { "A" => 1, "B" => 2, "C" => 3, "D" => 4 }

      ? "Array :" , Len( aArray  )           // result: 217
      ? "String:" , Len( cString )           // result:  11
      ? "Hash  :" , Len( hHash   )           // result:   4
   RETURN
```

LenNum()

Returns the number of characters a numeric value needs for display.

Syntax

```
LenNum( <nNumber> ) --> nLength
```

Arguments

`<nNumber>`

A numeric value.

Return

The function returns a numeric value. It is the number of characters required to display the numeric value, including the decimal point.

Description

LenNum() is a utility function that calculates the number of characters a numeric value needs for display. It is an abbreviation for the expression Len(LTrim(Str(*<nNumber>*))).

Info

See also: Len(), Str(), LTrim()
Category: Numeric functions, xHarbour extensions
Source: rtl\lennum.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example shows various results of LenNum()

   PROCEDURE Main
      ? LenNum( 1 )               // result: 1
      ? LenNum( 10 )              // result: 2
      ? LenNum( 100 )             // result: 3
      ? LenNum( 1000 )            // result: 4

      ? LenNum( 10.1 )            // result: 4
      ? LenNum( 100.12 )          // result: 6
      ? LenNum( 1000.123 )        // result: 8

      ? LenNum( 0.1 )             // result: 3
      ? LenNum( 0.01 )            // result: 4
      ? LenNum( 0.001 )           // result: 5
      ? LenNum( 0.0001 )          // result: 6
   RETURN
```

LibFree()

Releases a dynamically loaded xHarbour DLL from memory.

Syntax

```
LibFree( <pDllHandle> ) --> lSuccess
```

Arguments

<pDllHandle>

This is the pointer of the DLL to release as returned from LibLoad().

Return

The function returns a logical value indicating a successful operation.

Description

Function LibFree() releases a DLL previously loaded with LibLoad() from memory. When the function is successful, it returns .T. (true) and the pointer *<pDllHandle>* of the xHarbour DLL can no longer be used.

Info

See also:	HB_LibDo(), LibLoad(), FreeLibrary()
Category:	DLL functions, xHarbour extensions
Source:	vm\dynlibhb.c
LIB:	xhb.lib
DLL:	xhbdll.dll

LibLoad()

Loads an xHarbour DLL file into memory.

Syntax

```
LibLoad( <cDllFile> ) --> pDLL
```

Arguments

<cDLLFile>

This is a character string holding the name of the xHarbour DLL file to load into memory. It must contain complete path information, unless the file is located in the current directory, or in the list of directories held in the SET PATH environment variable of the operating system.

Return

The function returns a pointer to the DLL file. If the DLL file cannot be loaded, or does not exist, the return value is a null pointer.

Description

The LibLoad() function loads a DLL file created by xHarbour at runtime of an application into memory. All symbolic names of the functions residing in the DLL are added to the symbol table of the application. These functions can then be invoked via HB_LibDo() or by using the macro operator. Refer to function HB_LibDo() for an extensive example on the possibilities of invoking functions within a dynamically loaded DLL.

Note: function LoadLibrary() is available to load DLLs at runtime which are not created by xHarbour.

Info

See also:	HB_LibDo(), LibFree(), LoadLibrary()
Category:	DLL functions, xHarbour extensions
Source:	vm\dynlibhb.c
LIB:	xhb.lib
DLL:	xhbdll.dll

LoadLibrary()

Loads an external DLL file into memory.

Syntax

```
LoadLibrary( <cDLLFile> ) --> nDllHandle
```

Arguments

<cDLLFile>

This is a character string holding the name of the DLL file to load into memory. It must contain complete path information, unless the file is located in the current directory, or in the list of directories held in the SET PATH environment variable of the operating system.

Return

The function returns a numeric DLL handle > 0. If the DLL file cannot be loaded, or does not exist, the return value is zero.

Description

The LoadLibrary() function loads a DLL file at runtime of an xHarbour application into memory that is not created with xHarbour. Functions residing in this DLL can then be invoked via DllCall() by passing the returned DLL handle.

If the DLL is already in use by other applications, LoadLibrary() does not load the DLL a second time, but increments the load counter of the DLL. This signals the operating system that the xHarbour application requires the DLL in addition to other applications.

When the DLL is no longer required, the DLL handle should be freed with FreeLibrary().

Note: function LibLoad() is available to load a DLL cerated with xHarbour.

Info

See also:	DllCall(), FreeLibrary(), GetProcAddress(), LibLoad()
Category:	DLL functions, xHarbour extensions
Source:	rtl\dllcall.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows a recomended use of LoadLibrary() when DLL handles
// are required during the entire lifetime of an xHarbour application.
// The DLL is loaded in an INIT PROCEDURE and the DLL handle is available
// via a GLOBAL variable. This way, the DLL handle can be used in different
// API wrappers that invoke WinAPI functions within the same DLL.
// The example implements two wrappers for converting normal Ansi strings
// to wide character string (Unicode) and vice versa.

   #define DC_CALL_STD            0x0020

   GLOBAL gKernel32DLL

   INIT PROCEDURE InitDlls
      gKernel32DLL := LoadLibrary( "Kernel32.dll" )
   RETURN

   EXIT PROCEDURE FreeDlls
      FreeLibrary( gKernel32DLL )
```

```
RETURN


PROCEDURE Main
   LOCAL cString, cWideString

   cString     := "Hello World"
   cWideString := AnsiToWide( cString )

   ? Len( cString ), cString
   ? Len( cWideString ), cWideString

   ? WideToAnsi( cWideString )
RETURN


FUNCTION AnsiToWide( cString )
   LOCAL nWideLen  := 2 * ( Len( cString ) )
   LOCAL cWideChar := Replicate( Chr(0), nWideLen )
   LOCAL nRet

   nRet := ;
     DllCall( gKernel32DLL          , ;
              DC_CALL_STD           , ;
              "MultiByteToWideChar" , ;
              0                     , ;
              0                     , ;
              cString               , ;
              -1                    , ;
             @cWideChar             , ;
              nWideLen                )
RETURN cWideChar


FUNCTION WideToAnsi( cWideChar )
   LOCAL nLen    := Int( Len( cWideChar ) / 2 )
   LOCAL cString := Replicate( Chr(0), nLen )
   LOCAL nRet

   nRet := ;
     DllCall( gKernel32DLL          , ;
              DC_CALL_STD           , ;
              "WideCharToMultiByte" , ;
              0                     , ;
              0                     , ;
              cWideChar             , ;
              -1                    , ;
             @cstring               , ;
              nLen                  , ;
              0                     , ;
              0                       )
RETURN cString
```

Log()

Calculates the natural logarithm of a numeric value.

Syntax

```
Log( <nNumber> ) --> nNaturalLog
```

Arguments

<nNumber>

A numeric value greater than zero.

Return

The function returns the natural logarithm of *<nNumber>* as a numeric value.

Description

The mathematical function Log() calculates the natural logarithm for a mumeric value. The natural logarithm is based on the Euler number 2.7183... which is usually abbreviated as "e".

The inverse function of Log() is Exp().

Info

See also: Exp(), SET DECIMALS, SET FIXED
Category: Mathematical functions, Numeric functions
Source: rtl\math.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example shows results of function Log()

   PROCEDURE Main
      SET DECIMALS TO 5

      ? Log(10)                  // result: 2.30259
      ? Log(100)                 // result: 4.60517

      ? Log(1)                   // result: 0.00000
      ? Exp(Log(1))              // result: 1.00000
   RETURN
```

Log10()

Calculates the base 10 logarithm.

Syntax

```
Log10( <nValue> ) --> nLog10
```

Arguments

<nValue>

Any positive numeric value greater than zero can be passed.

Return

The function returns the logarithm base 10 as a numeric value.

Info

See also: Exp(), Log()
Category: CT:Math, Mathematical functions
Source: ct\ctmath2.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The exapmple displays results of function Log10().

   PROCEDURE Main

      ? Log10( 1 )           // result: 0
      ? Log10( 10 )          // result: 1
      ? Log10( 1000 )        // result: 3
      ? Log10( Infinity() )  // result: 19.97

   RETURN
```

Lower()

Converts a character string to lowercase.

Syntax

```
Lower( <cString> ) --> cLowerCaseString
```

Arguments

<cString>

A character string to convert to lowercase letters.

Return

The return value is a character string containing only lowercase letters.

Description

The character function Lower() copies the passed string, replaces all uppercase letters with lowercase letters and returns the result. It is related to function Upper() wich converts a string to uppercase letters. Both functions are used for case-insensitive string routines.

Info

See also:	IsLower(), IsUpper(), Upper()
Category:	Character functions
Source:	rtl\strcase.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates various results of LOWER()

   PROCEDURE Main

      ? Lower( "xHARbOUR" )            // result: xharbour
      ? Lower( "123 ABC - def" )       // result: 123 abc - def

   RETURN
```

LtoC()

Converts a logical value to a character.

Syntax

```
LtoN( [<lLogic>] ) --> nChar
```

Arguments

```
<lValue>
```

This is a logical value. It defaults to .F. (false).

Return

The function returns "T" for .T. (true) and "F" for .F. (false).

Info

See also: LtoN()
Category: CT:Miscellaneous, Miscellaneous functions
Source: ct\ctmisc.prg
LIB: xhb.lib
DLL: xhbdll.dll

LtoN()

Converts a logical value to a numeric value.

Syntax

```
LtoN( [<lLogic>] ) --> nLogic
```

Arguments

<lValue>

This is a logical value. It defaults to .F. (false).

Return

The function returns One for .T. (true) and Zero for .F. (false).

Info

See also: CtoN()
Category: CT:NumBits, Numbers and Bits
Source: ct\lton.c
LIB: xhb.lib
DLL: xhbdll.dll

LTrim()

Removes leading white-space characters from a character string.

Syntax

```
LTrim( <cString> ) --> cTrimmedString
```

Arguments

<cString>

A character string which is copied without leading white-space characters.

Return

The function returns a copy of *<cString>* without white-spaces at the beginning of the string.

Description

LTrim() is used for formatting character strings whose first characters consist of white-space characters (Chr(9),Chr(10),Chr(13),(Chr(32)). The function creates a copy of *<cString>* but ignores white spaces at the beginning of the input string.

Info

See also:	AllTrim(), IsSpace(), PadC() | PadL() | PadR(), RTrim(), Str(), SubStr()
Category:	Character functions
Source:	rtl\trim.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows various results of function LTrim().

   #define CRLF     Chr(13)+Chr(10)
   #define TAB      Chr(9)
   #define SPACE    Chr(32)

   PROCEDURE Main
      LOCAL cStr := TAB+SPACE+CRLF+" xHarbour"

      ? Len( cStr )                 // result: 13
      ? Asc( cStr )                 // result:  9

      ? Len( LTrim( cStr ) )        // result:  8
      ? Asc( LTrim( cStr ) )        // result:  120

      ? Str(5)                      // result:          5
      ? Len( Str(5) )               // result: 10
      ? LTrim( Str(5) )             // result: 5

   RETURN
```

LUpdate()

Returns the last modification date of a database open in a work area.

Syntax

```
LUpdate() --> dLastChange
```

Return

The function returns the date of the last change applied to a database fiel open in a work area. If the work area is unused, an empty date is returned.

Description

LUpdate() returns the date of the last change applied to a database file in a work area. This information is stored in the header of a database file and is only updated when the file is closed.

Info

See also:	DbInfo(), FCount(), FieldName(), Header(), LastRec(), RecSize()
Category:	Database functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The function illustrates that the last modification date
// is only updated when a database file is closed.

   PROCEDURE Main
      USE Customer ALIAS Cust NEW EXCLUSIVE

      ? DtoS(Date())           // result: 20060324
      ? DtoS(LUpdate())        // result: 20060103

      REPLACE Cust->LastNamet WITH "Miller"

      ? DtoS(LUpdate())        // result: 20060103

      CLOSE Cust

      USE Customer NEW
      ? DtoS(LUpdate())        // result: 20060324

      CLOSE ALL
   RETURN
```

MakeDir()

Creates a new directory.

Syntax

```
MakeDir( <cDirectory> ) --> nOSError
```

Arguments

`<cDirectory>`

A character expression specifying the directory to create. The directory can be specified relative to the current directory, or absolute, including a drive letter followed by a colon.

Return

The function returns a numeric value representing the operating system error code (DOS error). A value of 0 indicates a successful operation.

Description

The function attempts to create the directory specified with <*cDirectory*>. If this operation fails, the function returns the OS error code indicating the reason for failure. See the FError() function for a description of OS errors.

Note that <*cDirectory*> cannot contain subdirectories more than one level deep.

Info

See also:	DirChange(), DirRemove(), DiskChange(), DiskName(), FError(), IsDisk()
Category:	Directory functions, File functions
Source:	rtl\dirdrive.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates how to create nested sub-directories
// in the current directory

   PROCEDURE Main
      LOCAL i, j, aSubDir, cSubDir, nError
      LOCAL aNewDir := { ;
         "payments\salaries"   , ;
         "payments\purchases" , ;
         "customer\marketing" , ;
         "customer\orders"     , ;
         "customer\support"     }

      FOR i:=1 TO Len( aNewDir )
         cSubDir := CurDrive()+ ":\" + CurDir() + "\"
         aSubDir := HB_ATokens( aNewDir[i], "\" )

         FOR j:=1 TO Len( aSubDir )
            cSubDir += aSubDir[j] + "\"

            nError := MakeDir( cSubDir )
            IF nError == 0
               ? "Directory", cSubDir, "successfully created"
            ELSEIF nError == 5
               ? "Directory", cSubDir, "exists already"
```

```
         ELSE
            ? "Error for", cSubDir, LTrim( Str( nError ) )
         ENDIF
      NEXT j
   NEXT i

RETURN
```

Mantissa()

Calculates the mantissa of a floating point number.

Syntax

```
Mantissa( <nFloat> ) --> nMantissa
```

Arguments

<nFloatNumber>

Any numeric value can be passed.

Return

The function returns the mantissa base 2 of a floating point number. It is used together with the Exponent() for the binary representation of floating point numbers.

Info

See also:	Exponent()
Category:	CT:NumBits, Numbers and Bits
Source:	ct\exponent.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays results of Mantissa() and shows
// how a number is calculated from its mantissa and exponent.

   PROCEDURE Main
      LOCAL nE, nM

      ? Mantissa( -10 )      // result:  -1.25
      ? Mantissa(  -1 )      // result:  -1.00
      ? Mantissa(-0.1 )      // result:  -1.60
      ? Mantissa(   0 )      // result:   0.00
      ? Mantissa( 0.1 )      // result:   1.60
      ? Mantissa(   1 )      // result:   1.00
      ? Mantissa(  10 )      // result:   1.25

      nE := Exponent( 10 )
      nM := Mantissa( 10 )
      ? nM * 2^nE            // result:  10.0000
   RETURN
```

Max()

Returns the larger value of two Numerics or Dates.

Syntax

```
Max( <nNum1> , <nNum2>  ) --> nLargerNumber
Max( <dDate1>, <dDate2> ) --> dLargerDate
```

Arguments

`<nNum1> and <nNum2>`

Two numeric values to be compared.

`<nDate1> and <nDate2>`

Two Date values to be compared.

Return

The function returns the larger value of the two arguments.

Description

The function compares two numeric or two Date values and returns the larger value of both. Max() is used to normalize values against a limit. The inverse function is Min().

Info

See also:	Min()
Category:	Numeric functions, Date and time
Source:	rtl\minmax.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example uses Max() to identify the maximum length of a string within
// a set of character strings for formatted console output

   PROCEDURE Main
      LOCAL aLabel := { "Day", "Hour", "Minutes", "Seconds" }
      LOCAL dDate  := Date()
      LOCAL cTime  := Time()
      LOCAL nPad   := 0

      AEval( aLabel, {|c| nPad := Max( nPad, Len(c) ) } )

      ? PadL( aLabel[1], nPad ), dDate
      ? PadL( aLabel[2], nPad ), SubStr( cTime, 1, 2 )
      ? PadL( aLabel[3], nPad ), SubStr( cTime, 4, 2 )
      ? PadL( aLabel[4], nPad ), SubStr( cTime, 7, 2 )

   RETURN
```

MaxCol()

Determines the rightmost column position of the screen buffer.

Syntax

```
MaxCol() --> nColumn
```

Return

The function returns the ordinal position of the rightmost column in a console window as a numeric value.

Description

MaxCol() determines the maximum number of columns in a console window in which characters can be displayed. The leftmost column has number zero, the rightmost column has number MaxCol().

Info

See also:	Col(), MaxRow(), Row(), SetMode()
Category:	Screen functions
Source:	rtl\maxrow.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// In this example, a box is drawn and column numbers are displayed
// using a different color for every 10th column position.

   PROCEDURE Main
      LOCAL i

      @ 0, 0 TO 2, MaxCol() DOUBLE

      FOR i:=1 TO MaxCol()-1
         IF i % 10 > 0
            @ Row(), Col() SAY Str(i%10,1) COLOR "N/BG"
         ELSE
            @ Row(), Col() SAY Str(i/10,1) COLOR "GR+/BG"
         ENDIF
      NEXT
      @ 4, 0 CLEAR
   RETURN
```

MaxLine()

Returns the longest line in an ASCII formatted character string.

Syntax

```
MaxLine( <cText> ) --> nMaxLineLength
```

Arguments

<cText>

This is the ASCII formatted character string to process.

Return

The function recognizes Chr(13) and Chr(10) as end-of-line markers and counts all other characters. It returns the largest number of characters found in a single text line as a numeric value.

Note: the tab character (Chr(9)) is counted as a single character. If it exists in a text string, the text should be converted with TabExpand() before MaxLine() is called.

Info

See also:	MlCount(), NumLine(), TabExpand()
Category:	CT:String manipulation, Character functions
Source:	ct\maxline.c
LIB:	xhb.lib
DLL:	xhbdll.dll

MaxRow()

Determines the bottom row position of the screen buffer.

Syntax

```
MaxRow() --> nRow
```

Return

The function returns the ordinal position of the bottom row in a console window as a numeric value.

Description

MaxRow() determines the maximum number of rows in a console window in which characters can be displayed. The top row has number zero, the bottom row has number MaxRow().

Info

See also:	Col(), MaxCol(), Row(), SetMode()
Category:	Screen functions
Source:	rtl\maxrow.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The function changes the screen buffer, or console window size,
// and displays the size of the screen buffer.

   PROCEDURE Main
      LOCAL nRow := MaxRow(), nCol := MaxCol()
      CLS

      SetMode( 25, 80 )
      ? MaxRow(), MaxCol(), ScreenBufferSize()
      WAIT

      SetMode( 50, 80 )
      ? MaxRow(), MaxCol(), ScreenBufferSize()
      WAIT

      SetMode( 50, 135 )
      ? MaxRow(), MaxCol(), ScreenBufferSize()
      WAIT

      SetMode( nRow+1, nCol+1 )
   RETURN

   FUNCTION ScreenBufferSize
   RETURN 2 * ( 1 + MaxRow() ) * ( 1 + MaxCol() )
```

MCol()

Determines the screen column position of the mouse cursor.

Syntax

```
MCol() --> nMouseCol
```

Return

The function returns the column position of the mouse cursor as a numeric value.

Description

MCol() is used in full screen or console window applications to determine the current screen column position of the mouse cursor. The leftmost column has number zero, while the rightmost column has number MaxCol().

Note to obtain proper mouse input from the user, function Inkey() should be called including mouse events.

Info

See also:	Inkey(), MDblClk(), MHide(), MLeftDown(), MRightDown(), MRow(), MSetCursor(), MShow(), SET EVENTMASK
Category:	Mouse functions
Source:	rtl\mouseapi.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example changes the event mask for Inkey() to ALL events
// and displays the mouse cursor position when the left button is pressed.

   #include "Inkey.ch"

   PROCEDURE Main
      LOCAL nEvent, nRow, nCol, cPos
      CLS
      MShow()

      ? "Click with left mouse button (press ESC to quit)"
      SET EVENTMASK TO INKEY_ALL

      DO WHILE Lastkey() <> K_ESC
         nEvent := Inkey(0)

         IF nEvent == K_LBUTTONDOWN
            nRow := MRow()
            nCol := MCol()
            cPos := LTrim( Str(nRow) ) + "," + LTrim( Str(nCol) )
            // display current mouse cursor position
            @ nRow, nCol SAY cPos
         ENDIF
      ENDDO

   RETURN
```

MDblClk()

Determines the double-click interval for the mouse.

Syntax

```
MDblClk( [<nNewInterval>] ) --> nMilliSeconds
```

Arguments

`<nNewInterval>`

A numeric value specifying the interval in milliseconds between two mouse button clicks for being recognized as a double-click.

Return

The return value is the double-click interval present before the function is called.

Description

The function MDblClk() exists for compatibility reasons only and is not recommended to use. Mouse properties are nowadays set via the operating system configuration.

Info

See also:	MCol(), MRow()
Category:	Mouse functions
Source:	rtl\mouseapi.c
LIB:	xhb.lib
DLL:	xhbdll.dll

MDY()

Formats a date as "Month dd, yy".

Syntax

```
MDY( [<dDate>] ) --> cDate
```

Arguments

<dDate>

An expression returning a Date value. It defaults to Date().

Return

The function returns the formatted date as a character string. The string contains the month name. A two digit year is inserted when SET CENTURY is OFF.

Info

See also:	DMY(), SET CENTURY, SET DATE
Category:	CT:DateTime, Date and time
Source:	ct\dattime2.c
LIB:	xhb.lib
DLL:	xhbdll.dll

MemoEdit()

Displays and/or edits character strings and memo fields in text mode.

Syntax

```
MemoEdit( [<cString>]   , ;
          [<nTop>]      , ;
          [<nLeft>]     , ;
          [<nBottom>]   , ;
          [<nRight>]    , ;
          [<lEditMode>] , ;
          [<cUserFunc>] , ;
          [<nLineLen>]  , ;
          [<nTabSize>]  , ;
          [<nBufferRow>], ;
          [<nBufferCol>], ;
          [<nRowOffset>], ;
          [<nColOffset>]  ) --> cTextBuffer
```

Arguments

<cString>

A character string to be copied to the text buffer. The text buffer is displayed by MemoEdit() and can be edited by the user. The default value is an empty string ("").

<nTop>

A numeric value indicating the screen coordinate for the top row of the MemoEdit() display rectangle. The default value is 0.

<nLeft>

A numeric values indicating the screen coordinate for the left column of the MemoEdit() display rectangle. The default value is 0.

<nBottom>

A numeric value indicating the screen coordinate for the bottom row of the MemoEdit() display rectangle. The default value is MaxRow().

<nRight>

A numeric values indicating the screen coordinate for the right column of the MemoEdit() display rectangle. The default value is MaxCol().

<lEditMode>

The default value is .T. (true), indicating that a user can edit the text buffer maintained by MemoEdit(). If .F. (false) is passed, MemoEdit() displays the characater string *<cString>* within the boundaries of its display rectangle. The user can scroll the displayed text but cannot edit the text buffer.

<cUserFunc>

A character string holding the name of a user defined function. The function must be visible in the entire application, i.e. it must not be declared as STATIC FUNCTION. *<cUserFunc>* allows for modifying the standard edit behavior of MemoEdit() (see description below).

As an alternative, the value .F. (false) can be passed for *<cUserFunc>*. This causes MemoEdit() to display *<cString>* in the specified rectangle and return immediately. The passed string can neither be scrolled nor edited.

`<nLineLen>`

A numeric value specifying the number of characters that can appear in a single line of the text buffer. The default value is *<nRight>* - *<nLeft>*.

If a single text line in the text buffer contains more than *<nLineLen>* characters, the text line is word wrapped.

If *<nLineLen>* is specified as being larger than *<nRight>* - *<nLeft>* of the display rectangle, and the user navigates the text cursor to a column position beyond the coordinates of the MemoEdit() rectangle, the displayed text is scrolled horizontally.

`<nTabSize>`

A numeric value specifying the number of spaces a Tab character (Chr(9)) contained in *<cString>* should be replaced with in the displayed text buffer. The default value is 3 spaces.

`<nBufferRow>`

A numeric value specifying the first text buffer row to be displayed in the top screen row of the display rectangle. The default value is 1.

`<nBufferCol>`

A numeric value specifying the first text buffer column to be displayed in the left screen column of the display rectangle. The default value is 0.

`<nRowOffset>`

A numeric value specifying the row offset for initial positioning of the edit cursor. The default value is 0, i.e. the edit cursor is displayed in the first row of the display rectangle.

`<nColOffset>`

A numeric value specifying the column offset for initial positioning of the edit cursor. The default value is *<nBufferCol>*, i.e. the edit cursor is displayed in the specified column of the display rectangle, which is the first column, by default.

Return

The return value of MemoEdit() depends on the key a user pressed to terminate the function.

Alt+W When the user pressed the key combination Alt+W (K_ALT_W), the edited text buffer is returned as a character string.

Esc When the user pressed the Escape key (K_ESC), the return value is a copy of *<cString>*.

Description

MemoEdit() is a function for editing character strings, or memo fields, in console windows or full screen applications. The function provides a character oriented user interface and processes user input for editing text. Similar to AChoice() and DbEdit(), MemoEdit() employs a standard behaviour that can be configured by a user defined function *<cUserFunc>*.

Altough MemoEdit() is equipped with an extensive parameter profile, it is simple to use since all parameters are optional. The simplest form of using MemoEdit() is given by this line of code:

```
cText := MemoEdit()
```

Assigning the return value of MemoEdit() to a memory variable invokes the standard text editing behaviour of the function, i.e. the entire console window, or screen, is used for displaying the internal text buffer. The edited text is assigned to a memory variable when the user terminates text editing by pressing Alt+W. If text editing is ended with the Escape key, MemoEdit() returns an unchanged copy of *<cString>*.

It is important to understand that the user edits only the internal text buffer. This internal buffer is filled with a copy of parameter *<cString>*, if specified. All other parameters for row and column coordinates

change the default (initial) display of the rectangle where text is displayed and can be edited by the user.

Whether or not MemoEdit() exposes standard or user-configured text editing behavior depends on the parameter <*cUserFunc*>, which is a character string containing the name of a user defined function.

MemoEdit() without user function

In its standard behavior, MemoEdit() processes the keys listed in the following table:

Standard key processing of MemoEdit()

Key	Description
Navigation in the text buffer	
Up (Ctrl+E)	Go to previous line
Down (Ctrl+X)	Go to next line
Left (Ctrl+S)	Go to previous character
Right(Ctrl+D)	Go to next character
Ctrl+Left (Ctrl+A)	Go to previous word
Ctrl+Right (Ctrl+F)	Go to next word
Home	Go to begin-of-line
End	Go to end-of-line
Ctrl+Home	Go to first displayed line
Ctrl+End	Go to last displayed line
PgUp	Go to previous page
PgDn	Go to next page
Ctrl+PgUp	Go to first line
Ctrl+PgDn	Go to last line
Editing the text buffer	
Printable characters	Insert character
Return	Begin a new paragraph
Delete	Delete character at cursor
Backspace	Delete character to left of cursor
Tab	Insert tab character or spaces
Ctrl+Y	Delete the current line
Ctrl+T	Delete word right
Ctrl+B	Reformat paragraph
Ctrl+V (Ins)	Toggle insert/overstrike mode
Alt+W	Terminate editing with changes
Esc	Terminate editing without changes

Keys are grouped into the tasks navigation within the text buffer and editing it. MemoEdit() supports insert and overstrike modes, which is toggled by the Ins key. An automatic word wrap is applied to the text buffer when the number of characters entered in the current text line exceeds the value for parameter <*nLineLen*>. This word wrap is marked in the text buffer with a so called "Soft carriage return" (Chr(141)) which indicates the end-of-line, not the end-of-paragraph. The latter is marked in the text when the user presses the Return key. This inserts a "Hard carriage return" (Chr(13)) into the text buffer.

Note that "Soft carriage returns" remain in the text returned my MemoEdit(). When this text is output to a printer, or processed otherwise, it may be necessary to replace "Soft carriage returns" with blank spaces or "Hard carriage returns". The functions HardCR(), MemoTran() or StrTran() can be used to accomplish this.

MemoEdit() with user function

If a user function <*cUserFunc*> is specified, the standard editing behavior of MemoEdit() becomes configurable. For this, MemoEdit() distinguishes non-configurable keys from key-exceptions. The function applies its default action to the text buffer, when non-configurable keys are pressed. The user function is called for keys that yield an exception, and the return value of the user function instructs MemoEdit() how to process such a key. If there are no more keys pending in the keyboard buffer, the user function is called once again. As a consequence, the user function is called during different MemoEdit() modes.

The user function receives three numeric values from MemoEdit(): the current MemoEdit() mode, the current row, and the current column of the text buffer. #define constants are available in MEMOEDIT.CH that identify the different modes of MemoEdit().

MemoEdit() modes

Constant	Mode	Description
ME_IDLE	0	MemoEdit() is idle, all keys are processed
ME_UNKEY	1	Unknown key, text buffer is unaltered
ME_UNKEYX	2	Unknown key, text buffer is altered
ME_INIT	3	Initialization mode

ME_INIT The user function is called for the first time when MemoEdit() is invoked. At this point, MemoEdit() is initializing its text buffer and has not displayed it yet. The return value of the user function can be used at this stage to initialize word wrapping or the scrolling mode of MemoEdit() (see ME_TOGGLEWRAP and ME_TOGGLESCROLL in the table further down). The user function is called repeatedly in the initialization stage of MemoEdit() until the value ME_DEFAULT (0) is returned. After this, the text buffer is displayed and MemoEdit() continues key processing according to <*lEditMode*>. That is, if <*lEditMode*> is .F. (false), the user can scroll the text but cannot edit it.

ME_UNKEY This mode tells the user function that a configurable (unknown) key is pressed and the text buffer is not altered, yet. The return value of the user function instructs MemoEdit() how to treat a configurable key. Use function LastKey() to obtain the Inkey code of such key.

Configurable keys of MemoEdit()

Key	Default key processing
Ctrl+Y	Delete the current line
Ctrl+T	Delete word right
Ctrl+B	Reformat paragraph
Ctrl+V (Ins)	Toggle insert/overstrike mode
Alt+W	Terminate editing with changes
Esc	Terminate editing without changes

Returning ME_DEFAULT (0) from the user function for a configurable key instructs MemoEdit() to perform its default action. A different return value causes a different action, thus redefines the key (return values from the user function are explained further down).

ME_UNKEYX The same as ME_UNKEY, but the text buffer is altered.

ME_IDLE There are no more keys avalable in the keyboard buffer for processing. MemoEdit() calls the user function once in this situation and enters a wait state afterwards, until a new key is pressed. This mode is usually used to update row and column numbers on the screen if this information is presented to the user.

The second parameter passed to the user function is the current row in the text buffer. Rows are numbered beginning with 1.

The third parameter passed to the user function is the current column in the text buffer. Columns are numbered beginning with 0.

During the modes ME_INIT, ME_UNKEY and ME_UNKEYX, the return value of the user function instructs MemoEdit() what action to take for the pressed key. There are several #define constants available in MEMOEDIT.CH for this purpose.

Return values for the MemoEdit() user function

Constant	Value	Description
ME_DEFAULT	0	Perform default action
ME_UNKEY	1-31	Process requested action corresponding to key code
ME_IGNORE	32	Ignore unknown key
ME_DATA	33	Treat unknown key as data
ME_TOGGLEWRAP	34	Toggle word wrap mode
ME_TOGGLESCROLL	35	Toggle scroll mode
ME_WORDRIGHT	100	Perform word-right operation
ME_BOTTOMRIGHT	101	Perform bottom-right operation

Info

See also: HardCR(), MemoLine(), MemoRead(), MemoTran(), MemoWrit(), StrTran()
Category: Character functions, Memo functions, UI functions
Header: inkey.ch, memoedit.ch
Source: rtl\memoedit.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example implements a simple file editor using MemoEdit()
// with user function. The user function displays status information
// and configures the Alt+S (Save) and Alt+C (Cancel) as termination
// keys

   #include "Inkey.ch"
   #include "Memoedit.ch"

   STATIC slChanged := .F.

   PROCEDURE Main( cFileName )
      LOCAL cScreen
      LOCAL cText := ""

      SAVE SCREEN TO cScreen

      SET SCOREBOARD OFF
      SetCancel( .F. )
      CLS

      IF .NOT. Empty( cFileName ) .AND. File( cFileName )
         cText := MemoRead( cFileName )
      ENDIF
      @ 0, 0 TO MaxRow(), MaxCol() DOUBLE

      cText := MemoEdit( cText, ;
                         1, 1 , ;
                         MaxRow()-1, MaxCol()-1, ;
```

```
                     .T., "USERFUNC" )

   IF .NOT. Empty( cFileName ) .AND. ;
            File( cFileName  ) .AND. ;
            slChanged          .AND. ;
            Alert( "Save changes?", { "Yes", "No" } ) == 1

      // remove "soft carriage return/line feeds"
      cText := StrTran( cText, Chr(141)+Chr(10), " " )

      // save file
      MemoWrit( cFileName, cText )
   ENDIF

   RESTORE SCREEN FROM cScreen
RETURN


FUNCTION UserFunc( nMode, nRow, nCol )
   LOCAL nKey  := LastKey()
   LOCAL nRet  := ME_DEFAULT
   LOCAL cInfo := ""

   DO CASE
   CASE nMode == ME_INIT
      Set( _SET_INSERT, .F. )  // start in overstrike mode

   CASE nMode == ME_IDLE
      IF nKey > 31 .AND. nKey < 256
         slChanged := .T.
      ENDIF

      cInfo := "[row: " + LTrim(Str(nRow))
      cInfo += " col: " + LTrim(Str(nCol))+"]"
      cInfo += Chr(205)+Chr(205)

      IF Set( _SET_INSERT )
         cInfo += "[Ins]"
      ELSE
         cInfo += "[Ovr]"
      ENDIF

      IF slChanged
         cInfo += Chr(205)+Chr(205)+"[Chg]"
      ENDIF

      @ MaxRow(), 2 SAY cInfo + Replicate(Chr(205),6)

   CASE nMode == ME_UNKEY .OR. nMode == ME_UNKEYX
      // buffer is changed
      slChanged := ( nMode == ME_UNKEYX )

      DO CASE
      CASE nKey IN { K_ALT_W, K_CTRL_W, K_ESC }
         nRet := ME_IGNORE     // ignore default termination keys

      CASE nKey == K_ALT_S
         nRet := K_ALT_W       // Save with Alt+S

      CASE nKey == K_ALT_C
         nRet      := K_ESC    // Cancel with Alt+C
         slChanged := .F.
```

```
            ENDCASE

         ENDCASE

      RETURN nRet
```

MemoLine()

Extracts a line of text from a formatted character string or memo field.

Syntax

```
MemoLine( <cString>    , ;
         [<nLineLen>]  , ;
         [<nLineNum>]  , ;
         [<nTabSize>]  , ;
         [<lWrap>]     , ;
         [<lLongLines>], ;
         [@<nOffSet>]    ) --> cTextLine
```

Arguments

`<cString>`

A character string or memo field to extract a text line from. It can be a formatted text string that includes Tab and Hard/Soft carriage return characters.

`<nLineLen>`

A numeric value specifying the number of characters per extracted line. It is usually a value between 4 and 254. If *<nLineLen>* is larger than 254 characters, parameter *<lLongLines>* must be set to .T. (true). The default value for *<nLineLen>* is 79.

`<nLineNum>`

A numeric value specifying the line number to extract. It defaults to 1.

`<nTabSize>`

A numeric value specifying the number of blank spaces the Tab character should be expanded to. It defaults to 4 blank spaces.

`<lWrap>`

A logical value indicating if word wrapping should be applied to *<cString>* when lines are extracted. The default value is .T. (true), resulting in extracted text lines that contain whole words only. When a word does not fit entirely to the end of the extracted text line, it is wrapped to the next text line. Passing .F. (false) for this parameter turns word wrapping off so that only lines ending with a hard carriage return are extracted.

`<lLongLines>`

This parameter defaults to .F. (false). It must be set to .T. (true) when text lines of more than 254 characters should be extracted.

`@<nOffSet>`

A numeric value specifying the first character in *<cString>* to begin with extracting text lines. It defaults to 1. If passed by reference, *<nOffSet>* receives the starting position of the next line that can be extracted from *<cString>* in a repeated call to MemoLine(). This improves the speed of MemoLine() considerably when multiple lines are extracted from text of more than 64 kB.

Return

The function returns a character string of *<nLineLen>* characters. If *<lWrap>* is .T. (true) and the extracted line does not end with a complete word, the line is filled with blank spaces up to the length of *<nLineLen>*. If *<lWrap>* is .F. (false), the returned string contains *<nLineLen>* characters and a next call to MemoLine() extracts characters following the next hard carriage return.

If *<nLineNum>* is larger than the number of lines contained in *<cString>*, an empty string ("") is returned.

Description

MemoLine() is used to extract single text lines from a formatted text string that may include Tab characters (Chr(9)), Soft carriage returns (Chr(141)) or Hard carriage returns (Chr(13)). Multiple lines are usually extracted within a FOR..NEXT loop whose upper boundary is determined by the total number of text lines in a string, as reported by function MLCount().

If multiple lines are extracted, it is strongly recommended to pass *<nOffSet>* by reference to MemoLine(), initializing the parameter with 1. This improves the speed of text extraction considerably.

note: when text lines extracted with MemoLine() are displayed, they have only the same length if a fixed-size font is used for display.

Info

See also:	FLineCount(), HB_FReadLine(), HB_ReadLine(), MemoEdit(), MemoRead(), MemoWrit(), MLCount(), MLPos()
Category:	Character functions, Memo functions
Source:	rtl\txtline.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates MemoLine() usage for extracting all lines
// of a text file. The difference of using line numbers versus using
// line offsets is outlined.

   PROCEDURE Main( cFileName )
      LOCAL nLineLen  := 60
      LOCAL nTabSize  :=  8
      LOCAL lWrap     := .T.
      LOCAL i, imax, cText, cLine, nOffSet, nTextLen
      LOCAL t1, t2, t3

      IF Empty( cFileName ) .OR. .NOT. File( cFileName )
         ? "No file specified"
         QUIT
      ENDIF

      cText := MemoRead( cFileName )

      t1   := Seconds()
      imax := MLCount( cText, nLineLen, nTabSize, lWrap )
      FOR i:=1 TO imax
         cLine := MemoLine( cText, nLineLen, i, nTabSize, lWrap )
      NEXT
      t2 := Seconds()

      // This example shows the use of the <nOffSet> parameter.
      // Note that the line number does not change, but the starting
      // position in the text to begin extracting a line
      nTextLen := Len( cText )
      nOffSet  := 1
      i        := 0
      DO WHILE nOffSet <= nTextLen
         cLine := MemoLine( cText, nLineLen, 1, nTabSize, lWrap,, @nOffSet )
         i++
      ENDDO
      t3 := Seconds()

      ? "Using line number:", t2-t1, "secs (" + Ltrim(Str(imax)) + " lines)"
```

```
   ? "Using line offset:", t3-t2, "secs (" + Ltrim(Str(i)) + " lines)"
RETURN
```

MemoRead()

Reads an entire file from disk into memory.

Syntax

```
MemoRead( <cFileName> ) --> cString
```

Arguments

<cFileName>

This is a character string holding the name of the file to read. It must include path and file extension. If the path is omitted from *<cFileName>*, the file is searched in the current directory.

Return

The function returns the entire contents of file *<cFileName>* as a character string. If the file is not found, an empty string ("") is returned.

Description

MemoRead() provides the fastest way of reading the entire contents of a file from disk and assign it to a memory or field variable. The function searches a file only in the current directory, if *<cFileName>* is not a full qualified file name. The settings of SET PATH and SET DEFAULT are ignored.

The file is opened as read-only and in shared mode. If this fails due to another process having obtained exclusive access to the file, or if the file is not found, the return value is an empty string ("").

Info

See also:	MemoEdit(), MemoLine(), MemoWrit(), REPLACE
Category:	Character functions, Memo functions
Source:	rtl\memofile.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// See the examples of function MemoEdit() or MemoLine() for
// a usage scenario of MemoRead()
```

Memory()

Returns memory statistics.

Syntax

```
Memory( <nWhichMemory> ) --> nMemoryUsage
```

Arguments

<nWhichMemory>

A numeric value must be passed to identify the memory to query. The following table lists #define constants listed in HBMemory.ch that can be used for *<nWichMemory>*.

Constants for memory statistics

Constant	Value	Description
HB_MEM_CHAR	0	Free Variable Space (KB)
HB_MEM_BLOCK	1	Largest String (KB)
HB_MEM_RUN	2	RUN Memory (KB)
HB_MEM_VM	3	Virtual Memory (KB)
HB_MEM_FM	101	Fixed Memory/Heap (KB)
HB_MEM_FMSEGS	102	Segments in Fixed Memory/Heap
HB_MEM_SWAP	103	Free Swap Memory (KB)
HB_MEM_CONV	104	Free Conventional (KB)
HB_MEM_EMSUSED	105	Used Expanded Memory (KB)
HB_MEM_USED	1001	Memory used (bytes)
HB_MEM_USEDMAX	1002	Maximum memory used (bytes)
HB_MEM_STACKITEMS	1003	Total items on the stack
HB_MEM_STACK	1004	Total memory size used by the stack (bytes)
HB_MEM_STACK_TOP	1005	Total items currently on the stack
HB_MEM_LIST_BLOCKS	1006	List all allocated blocks

Return

The function returns a numeric value indicating usage of the specified memory.

Description

The function is used to obtain values for current memory usage. The type of memory to use must be specified with #define constants. Not all memory types are supported on all platforms. If a memory type cannot be queried, the return value is zero.

Memory() can give only a "snapshot" of current memory usage while the function is being executed. The memory usage may have changed when the function returns due to the activity of xHarbour's garbage collector.

Info

See also:	GetEnv(), HB_GCAll()
Category:	Debug functions, Environment functions, xHarbour extensions
Header:	hbmemory.ch
Source:	vm\fm.c
LIB:	xhb.lib
DLL:	xhbdll.dll

MemoTran()

Replaces "carriage return/line feed" pairs in a character string.

Syntax

```
MemoTran( <cString>          , ;
         [<cReplaceHardCRLF>], ;
         [<cReplaceSoftCRLF>]  ) --> cNewString
```

Arguments

`<cString>`

A character string or memo field to replace carriage return/line feed pairs in.

`<cReplaceHardCRLF>`

A single character used to replace hard carriage return/line feed pairs with. It defaults to a semicolon (;).

`<cReplaceSoftCRLF>`

A single character used to replace soft carriage return/line feed pairs with. It defaults to a blank space.

Return

The function returns a copy of *<cString>* where all carriage return / line feed pairs are replaced.

Description

The function replaces hard and soft carriage return/line feed pairs with single characters in a memo field or a character string. A hard carriage return is Chr(13), while a soft carriage return is Chr(141). The latter is inserted into a text string by function MemoEdit() when automatic word wrap is set to .T. (true).

Info

See also:	HardCR(), MemoEdit(), StrTran()
Category:	Character functions, Memo functions
Source:	rtl\mtran.c
LIB:	xhb.lib
DLL:	xhbdll.dll

MemoWrit()

Writes a character string or a memo field to a file.

Syntax

```
MemoWrit( <cFileName>, <cString> ) --> lSuccess
```

Arguments

<cFileName>

This is a character string holding the name of the file to create. It must include path and file extension. If the path is omitted from *<cFileName>*, the file is created in the current directory.

<cString>

A character string to be written to the file *<cFileName>*.

Return

The function returns .T. (true) when the string *<cString>* could be written to the file *<cFileName>*, otherwise .F. (false) is returned.

Description

MemoWrit() creates a file, writes the character string *<cString>* to it, and closes the file. If the file exists already, it is overwritten without warning. When the file cannot be created, the return value is .F. (false).

Note: the function writes an end-of-file marker (Chr(26)) to the created file. That means, the file created by MemoWrit() contains one byte more than the input string *<cString>*. This must be taken into consideration if the file is later read using MemoRead().

Info

See also:	FCreate(), FWrite(), MemoEdit(), MemoRead()
Category:	Character functions, Memo functions
Source:	rtl\memofile.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example contains the shortest possible source code of
// a complete text file editor (8 lines of code).

   PROCEDURE Main( cFileName )
      IF Empty( cFileName )
         ? "No file name specified"
         QUIT
      ENDIF

      MemoWrit( cFileName, MemoEdit( MemoRead( cFilename ) ) )
   RETURN
```

MemVarBlock()

Creates a set/get code block for a dynamic memory variable.

Syntax

```
MemVarBlock( <cMemVarName> ) --> bMemVarBlock
```

Arguments

`<cMemVarName>`

A character string holding the name of a dynamic memory variable (PRIVATE or PUBLIC).

Return

The function returns a set/get code block that accesses the memory variable *<cMemVarName>*. If the variable does not exist, the return value is NIL.

Description

The function creates a code block accessing a dynamic memory variable, i.e. a variable of PRIVATE or PUBLIC scope. The code block accepts one parameter. If a parameter is passed, its value is assigned to the variable *<cMemVarName>*. Evaluating the code block without a parameter returns the value of the variable.

Info

See also:	FieldBlock(), FieldWBlock()
Category:	Code block functions
Source:	rtl\memvabl.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example creates two code blocks accessing a PRIVATE
// and a PUBLIC variable.

   PROCEDURE Main
      MEMVAR cPrivate, cPublic
      LOCAL bMemVar1 , bMemVar2

      PUBLIC  cPublic  := "PUBLIC"
      PRIVATE cPrivate := "PRIVATE"

      bMemVar1 := MemVarBlock( "cPublic"  )
      bMemVar2 := MemVarBlock( "cPrivate" )

      ? Eval( bMemVar1 )              // result: PUBLIC
      ? Eval( bMemVar2 )              // result: PRIVATE

      ? Eval( bMemVar1, "Hello" )     // result: Hello
      ? Eval( bMemVar2, "World" )     // result: World

      ? cPublic                       // result: Hello
      ? cPrivate                      // result: World
   RETURN
```

MenuModal()

Activates the menu system represented by a TopBarMenu object.

Syntax

```
MenuModal( <oTopBarMenu> , ;
           [<nStartItem>], ;
           [<nMsgRow>]   , ;
           [<nMsgLeft>]  , ;
           [<nMsgRight>] , ;
           [<cMsgColor>] , ;
           [<GetList>      ) --> nMenuItemID
```

Arguments

<oTopBarMenu>

This is a TopBarMenu() object to activate.

<nStartItem>

This is a numeric value indicating the ordinal position of the first menu item to be selected when the menu is activated.

<nMsgRow>

This is a numeric value specifying the screen row for displaying messages assigned to the instance variable :message of menu items. The range for *<MsgnRow>* is between 0 and MaxRow().

<nMsgLeft>

This is a numeric value specifying the left screen coordinate for displaying menu messages. Usually, *<nMsgLeft>* is set to the value 0.

<nMsgRight>

This is a numeric value specifying the right screen coordinate for displaying menu messages. Usually, *<nMsgRight>* is set to the value of MaxCol().

<cMsgColor>

The parameter *<cMsgColor>* is an optional character string defining the color for the message to display. It defaults to SetColor().

<GetList>

Optionally, a *GetList* array can be passed when the menu is activated during READ.

Return

The function returns the ordinal position of the selected menu item. If the user cancels menu selection with the Esc key, the return value is zero.

Description

Function MenuModal() is provided for compatibility reasons. It calls method :modal() of the *<oTopBarMenu>* object and returns this method's result.

Info

See also:	READ, TopBarMenu()
Category:	Get system, UI functions
Source:	rtl\ttopbar.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

MHide()

Hides the mouse cursor.

Syntax

```
MHide() --> NIL
```

Return

The function returns always NIL.

Description

The function MHide() exists for compatibility reasons only and is not recommended to use. Hiding the mouse cursor is only relevant when an application runs in full screen mode. Use function MSetCursor() for hiding/displaying the mouse cursor in such an application.

Info

See also:	MShow(), MSetCursor()
Category:	Mouse functions
Source:	rtl\mouseapi.c
LIB:	xhb.lib
DLL:	xhbdll.dll

MilliSec()

Defines a time delay in milliseconds.

Syntax

```
MilliSec( <nMilliSeconds> ) --> cNull
```

Arguments

<nMilliSeconds>

This is a numeric value indicating the number of millseconds to suspend program execution. No CPU resources are consumed until the function returns.

Return

The function returns always a null string ("").

Info

See also:	Seconds(), WaitPeriod()
Category:	CT:DateTime, Date and time
Source:	ct\cttime.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Min()

Returns the smallerr value of two Numerics or Dates.

Syntax

```
Min( <nNum1> , <nNum2>  ) --> nSmallerNumber
Min( <dDate1>, <dDate2> ) --> dSMallerDate
```

Arguments

`<nNum1> and <nNum2>`

Two numeric values to be compared.

`<nDate1> and <nDate2>`

Two Date values to be compared.

Return

The function returns the smaller value of the two arguments.

Description

The function compares two numeric or two Date values and returns the smaller value of both. Min() is used to normalize values against a limit. The inverse function is Max().

Info

See also:	Max()
Category:	Numeric functions, Date and time
Source:	rtl\minmax.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example compares elements of two arrays of different
// length. The limit for the loop counter is determined with
// function Min(). Array elements containing same values are
// collected in a result array.

   PROCEDURE Main
      LOCAL aArray1 := {"A", "B", "C", "D" }
      LOCAL aArray2 := {"D", "B", "C" }
      LOCAL aResult := {}
      LOCAL i, imax

      imax := Min( Len( aArray1 ), Len( aArray2 ) )
      FOR i:=1 TO imax
          IF aArray1[i] == aArray2[i]
             AAdd( aResult, aArray1[i] )
          ENDIF
      NEXT

      ? ValToPrg( aResult )   // result: { "B", "C" }
   RETURN
```

Minute()

Extracts the minute from a DateTime value

Syntax

```
Minute( <dDateTime> ) --> nMinute
```

Arguments

<dDateTime>

This is a DateTime() value.

Return

The function extracts the minute from a DateTime value and returns it as a numeric value.

Info

See also:	DateTime(), Day(), Hour(), Month(), Secs(), Year()
Category:	Conversion functions, Date and time, xHarbour extensions
Source:	rtl\dateshb.c
LIB:	xhb.lib
DLL:	xhbdll.dll

MLCount()

Counts the number of lines in a character string or memo field

Syntax

```
MLCount( <cString>    , ;
        [<nLineLen>] , ;
        [<nTabSize>] , ;
        [<lWrap>]    , ;
        [<lLongLines>] ) --> nLineCount
```

Arguments

`<cString>`

A character string or memo field to be counted. It can be a formatted text string that includes Tab and Hard/Soft carriage return characters.

`<nLineLen>`

A numeric value specifying the number of characters per line. It is usually a value between 4 and 254. If *<nLineLen>* is larger than 254 characters, parameter *<lLongLines>* must be set to .T. (true). The default value for *<nLineLen>* is 79.

`<nTabSize>`

A numeric value specifying the number of blank spaces the Tab character should be expanded to. It defaults to 4 blank spaces.

`<lWrap>`

A logical value indicating if word wrapping should be applied to *<cString>* when lines are counted. The default value is .T. (true), resulting in text lines being counted that contain whole words only. When a word does not fit entirely to the end of a text line, it is wrapped to the next text line. Passing .F. (false) for this parameter turns word wrapping off so that only lines ending with a hard carriage return are counted.

`<lLongLines>`

This parameter defaults to .F. (false). It must be set to .T. (true) when text lines of more than 254 characters should be counted.

Return

The function returns the number of lines contained in *<cString>* when it is formatted according to the parameters *<nLineLen>*, *<nTabSize>* and *<lWrap>*.

Description

MLCount() is used in conjunction with MemoLine() that extracts single lines of text from a formatted text string or memo field. MLCount() determines the total number of lines that can be extracted from *<cString>*.

Info

See also: MemoLine(), MemoTran(), MLPos()
Category: Memo functions, Character functions
Source: rtl\txtline.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// Refer to function MemoLine() for a usage example of MLCount()
```

MLCToPos()

Determines the position of a single character in a formatted text string or memo field.

Syntax

```
MLCToPos( <cString>  , ;
         [<nLineLen>], ;
         [<nTextRow>], ;
         [<nTextCol>], ;
         [<nTabSize>], ;
         [<lWrap>]   , ;
         [<lLongLines>]) --> nPosition
```

Arguments

<cString>

A character string or memo field to scan. It can be a formatted text string that includes Tab and Hard/Soft carriage return characters.

<nLineLen>

A numeric value specifying the number of characters per text line. It is usually a value between 4 and 254. If *<nLineLen>* is larger than 254 characters, parameter *<lLongLines>* must be set to .T. (true). The default value for *<nLineLen>* is 79.

<nTextRow>

A numeric value specifying the text row, or line number, to find a character in. Text lines are counted beginning with 1, which is also the default value for *<nTextRow>*.

<nTextCol>

A numeric value specifying the column within the text row to find a character in. Columns are counted beginning with 0, which is also the default value for *<nTextCol>*.

<nTabSize>

A numeric value specifying the number of blank spaces the Tab character should be expanded to. It defaults to 4 blank spaces.

<lWrap>

A logical value indicating if word wrapping should be applied to *<cString>* when lines are scanned. The default value is .T. (true), resulting in text lines that contain whole words only. When a word does not fit entirely to the end of the scanned text line, it is wrapped to the next text line. Passing .F. (false) for this parameter turns word wrapping off so that only lines ending with a hard carriage return are scanned.

<lLongLines>

This parameter defaults to .F. (false). It must be set to .T. (true) when text lines of more than 254 characters should be scanned.

Return

The function returns the position of the character in *<cString>* that is located in the text row *<nTextRow>* and column *<nTextCol>*. Position counting starts with 1.

Description

MLCToPos() calculates the position of a single character in a formatted text string, based on row and column position. Row and column positions are compatible with the parameters MemoEdit() passes to its user function. The return value of MLCToPos(), in contrast, is compatible with the SubStr() function, so that substrings can be extracted from a formatted character string.

Info

See also: MemoEdit(), MLCount(), MLPos(), MPosToLC(), SubStr()
Category: Character functions, Memo functions
Source: rtl\mlcount.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// This example determines the byte position of line 3, column 6
// when the text string is formatted with 10 characters per line:

   PROCEDURE Main
      LOCAL cString  := "The quick brown fox jumps over the lazy dog"
      LOCAL cLine    := ""
      LOCAL nLineLen := 10
      LOCAL i, imax  := MLCount( cString, nLineLen )

      ? '"' + cString + '"'
      ?
      FOR i:=1 TO imax
         cLine := MemoLine( cString, nLineLen, i )
         ? '"' + cLine + '"'
      NEXT
      // Output so far:
      // "The quick brown fox jumps over the lazy dog"
      //
      // "The quick "
      // "brown fox "
      // "jumps over"
      // "the lazy  "
      // "dog       "

      ? i := MLCtoPos( cString, nLineLen, 3, 6, 0) // result: 27
      ? SubStr( cString, i, nLineLen )             // result: over the l
   RETURN
```

MLeftDown()

Determines the status of the left mouse button.

Syntax

```
MLeftDown() --> lLeftButtonIsPressed
```

Return

MLeftDown() returns .T. (true) if the left mouse button is pressed when the function is called, otherwise .F. (false) is returned.

Description

MLeftDown() is used in full screen or console window applications to determine the current status of the left mouse button.

Info

See also:	Inkey(), MDblClk(), MHide(), MRightDown(), MRow(), MShow(), SET EVENTMASK
Category:	Mouse functions
Source:	rtl\mouseapi.c
LIB:	xhb.lib
DLL:	xhbdll.dll

MlPos()

Determines the starting position of a line in a formatted character string or memo field.

Syntax

```
MlPos( <cString>   , ;
      [<nLineLen>  , ;
      [<nLineNum>] , ;
      [<nTabSize>] , ;
      [<lWrap>]    , ;
      [<lLongLines>] ) --> nPosition
```

Arguments

<cString>

A character string or memo field to scan. It can be a formatted text string that includes Tab and Hard/Soft carriage return characters.

<nLineLen>

A numeric value specifying the number of characters per text line. It is usually a value between 4 and 254. If *<nLineLen>* is larger than 254 characters, parameter *<lLongLines>* must be set to .T. (true). The default value for *<nLineLen>* is 79.

<nLineNum>

A numeric value specifying the text row, or line number, to find the starting position for. Text lines are counted beginning with 1, which is also the default value for *<nLineNum>*.

<nTabSize>

A numeric value specifying the number of blank spaces the Tab character should be expanded to. It defaults to 4 blank spaces.

<lWrap>

A logical value indicating if word wrapping should be applied to *<cString>* when lines are scanned. The default value is .T. (true), resulting in text lines that contain whole words only. When a word does not fit entirely to the end of the scanned text line, it is wrapped to the next text line. Passing .F. (false) for this parameter turns word wrapping off so that only lines ending with a hard carriage return are scanned.

<lLongLines>

This parameter defaults to .F. (false). It must be set to .T. (true) when text lines of more than 254 characters should be scanned.

Return

The function returns the starting position of the text line having number *<nLineNum>* in *<cString>*. Position counting starts with 1.

Description

MLPos() is used similar to MLCToPos() for finding an exact byte position in a formatted text string. MlPos() ingnores column positions within a text line and searches only for the beginning of a line of text.

Info

See also: MemoLine(), MemoTran(), MLCount(), MLCToPos(), MPosToLC()
Category: Character functions, Memo functions
Source: rtl\mlpos.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// This example determines the starting position of the 4th line
// when the text string is formatted with 10 characters per line:

   PROCEDURE Main
      LOCAL cString  := "The quick brown fox jumps over the lazy dog"
      LOCAL cLine    := ""
      LOCAL nLineLen := 10
      LOCAL i, imax  := MLCount( cString, nLineLen )

      ? '"' + cString + '"'
      ?
      FOR i:=1 TO imax
         cLine := MemoLine( cString, nLineLen, i )
         ? '"' + cLine + '"'
      NEXT
      // Output so far:
      // "The quick brown fox jumps over the lazy dog"
      //
      // "The quick "
      // "brown fox "
      // "jumps over"
      // "the lazy  "
      // "dog       "
      ?
      ? i := MLPos( cString, nLineLen, 4 )   // result: 32
      ? SubStr( cString, i, nLineLen )       // result: the lazy d
   RETURN
```

Mod()

Calculates the modulus of two numbers.

Syntax

```
Mod( <nDividend>, <nDivisor> ) --> nRemainder
```

Arguments

`<nDividend>`

This is the numeric dividend of the division operation.

`<nDivisor>`

This is the numeric divisor of the division operation.

Return

The function returns the numeric remainder of a division of *<nDividend>* by *<nDivisor>*.

Description

Function Mod() exists for compatibility reasons only. It is superseeded by the modulus operator.

Info

See also:	% operator
Source:	rtl\mod.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Month()

Extracts the numeric month number from a Date value.

Syntax

```
Month( <dDate> ) --> nMonth
```

Arguments

<dDate>

This is an expression returning a value of data type Date.

Return

The function returns the numeric month number of *<dDate>*. When *<dDate>* is an empty date, the function returns zero.

Description

This function is used to extract the numeric month from a Date value.

Info

See also:	CMonth(), Day(), DoW(), Hour(), Minute(), Secs(), Year()
Category:	Conversion functions, Date and time
Source:	rtl\dateshb.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows results of the function Month()

   PROCEDURE Main

      ? Date()                          // result: 02/25/06
      ? Month( Date() )                 // result: 2
      ? Month( Date() + 7 )             // result: 3

   RETURN
```

MPosToLC()

Calculates row and column position of a character in a formatted string or memo field.

Syntax

```
MPosToLC( <cString>  , ;
         [<nLineLen>], ;
         [<nCharPos>], ;
         [<nTabSize>], ;
         [<lWrap>]   , ;
         [<lLongLines>]) --> { nTextRow, nTextCol }
```

Arguments

<cString>

A character string or memo field to scan. It can be a formatted text string that includes Tab and Hard/Soft carriage return characters.

<nLineLen>

A numeric value specifying the number of characters per text line. It is usually a value between 4 and 254. If *<nLineLen>* is larger than 254 characters, parameter *<lLongLines>* must be set to .T. (true). The default value for *<nLineLen>* is 79.

<nCharPos>

A numeric value indicating the position of the character in *<cString>*. The default value is 1.

<nTabSize>

A numeric value specifying the number of blank spaces the Tab character should be expanded to. It defaults to 4 blank spaces.

<lWrap>

A logical value indicating if word wrapping should be applied to *<cString>* when lines are scanned. The default value is .T. (true), resulting in text lines that contain whole words only. When a word does not fit entirely to the end of the extracted text line, it is wrapped to the next text line. Passing .F. (false) for this parameter turns word wrapping off so that only lines ending with a hard carriage return are scanned.

<lLongLines>

This parameter defaults to .F. (false). It must be set to .T. (true) when text lines of more than 254 characters should be extracted.

Return

The function returns an array of two elements. The first element contains the row in the text where the character at position *<nCharPos>* is located, and the second element is the column number.

Description

MPosToLC() calculates the row and column position of a single character in a formatted text string, based on its byte position. Row and column positions are compatible with the parameters MemoEdit() passes to its user function. That is, rownumbersing begins with 1 and columns are numbered beginning with 0.

Info

See also: MemoEdit(), MLCToPos(), MLPos()
Category: Character functions, Memo functions
Source: rtl\mpostolc.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// This example determines the text row and column of the character
// at position 36, when the text is formatted with 10 characters per line:

   PROCEDURE Main
      LOCAL cString  := "The quick brown fox jumps over the lazy dog"
      LOCAL aLines   := {}
      LOCAL nLineLen := 10
      LOCAL i, imax  := MLCount( cString, nLineLen )
      LOCAL aRowCol

      ? '"' + cString + '"'
      ?
      FOR i:=1 TO imax
         AAdd( aLines, MemoLine( cString, nLineLen, i ) )
         ? '"' + aLines[i] + '"'
      NEXT
      // Output so far:
      // "The quick brown fox jumps over the lazy dog"
      //
      // "The quick "
      // "brown fox "
      // "jumps over"
      // "the lazy  "
      // "dog       "

      ? SubStr( cString, 36 )                      // result: lazy dog

      aRowCol := MPosToLC( cString, nLineLen, 36 )
      ? aRowCol[1], aRowCol[2]                     // result:   4  4

      ? SubStr( aLines[aRowCol[1]], 1+aRowCol[2] )  // result: lazy
   RETURN
```

MPresent()

Determines if a mouse is available.

Syntax

```
MPresent() --> lMouseIsPresent
```

Return

The function returns .T. (true) when a mouse is present, otherwise the return value is .F. (false).

Description

The function tests the presence of a mouse. It is a compatibility function for text mode and full screen applications.

Info

See also:	MRestState(), MSaveState(), MSetCursor()
Category:	Mouse functions
Source:	rtl\mouseapi.c
LIB:	xhb.lib
DLL:	xhbdll.dll

MRestState()

Restores a previously saved state of the mouse.

Syntax

```
MRestState( <cSavedState> ) --> NIL
```

Arguments

<cSavedState>

This must be a character string previously returned from function MSaveState().

Return

The function always returns NIL.

Description

The function restores settings of the mouse that are previously saved with a call to MSaveState(). It is a compatibility function for text mode and full screen applications.

Info

See also: MPresent(), MSaveState(), MSetCursor()
Category: Mouse functions
Source: rtl\mouseapi.c
LIB: xhb.lib
DLL: xhbdll.dll

MRightDown()

Determines the status of the left mouse button.

Syntax

```
MRightDown() --> lRightButtonIsPressed
```

Return

MRightDown() returns .T. (true) if the right mouse button is pressed when the function is called, otherwise .F. (false) is returned.

Description

MRightDown() is used in full screen or console window applications to determine the current status of the right mouse button.

Info

See also:	Inkey(), MDblClk(), MHide(), MLeftDown(), MRow(), MShow(), SET EVENTMASK
Category:	Mouse functions
Source:	rtl\mouseapi.c
LIB:	xhb.lib
DLL:	xhbdll.dll

MRow()

Determines the screen row position of the mouse cursor.

Syntax

```
MRow() --> nMouseRow
```

Return

The function returns the row position of the mouse cursor as a numeric value.

Description

MRow() is used in full screen or console window applications to determine the current screen row position of the mouse cursor. The top screen row has number zero, while the bottom screen row has number MaxRow().

Note to obtain proper mouse input from the user, function Inkey() should be called including mouse events.

Info

See also:	Inkey(), MCol(), MDblClk(), MHide(), MLeftDown(), MRightDown(), MSetCursor(), MShow(), SET EVENTMASK
Category:	Mouse functions
Source:	rtl\mouseapi.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example changes the event mask for Inkey() to ALL events
// and displays the mouse cursor position when the left button is pressed.

   #include "Inkey.ch"

   PROCEDURE Main
      LOCAL nEvent, nRow, nCol, cPos
      CLS
      MShow()

      ? "Click with left mouse button (press ESC to quit)"
      SET EVENTMASK TO INKEY_ALL

      DO WHILE Lastkey() <> K_ESC
         nEvent := Inkey(0)

         IF nEvent == K_LBUTTONDOWN
            nRow := MRow()
            nCol := MCol()
            cPos := LTrim( Str(nRow) ) + "," + LTrim( Str(nCol) )
            // display current mouse cursor position
            @ nRow, nCol SAY cPos
         ENDIF
      ENDDO

   RETURN
```

MSaveState()

Saves the current state of the mouse.

Syntax

```
MSaveState() --> cSavedState
```

Return

The function returns the current mouse state encoded in a binary character string.

Description

MSaveState() is used in text mode and console window applications to save the current settings of the mouse. This includes the screen position of the mouse cursor, its visibility and boundaries. The return value can be passed to function MRestState() to restore the saved state.

Info

See also:	MPresent(), MRestState(), MSetCursor()
Category:	Mouse functions
Source:	rtl\mouseapi.c
LIB:	xhb.lib
DLL:	xhbdll.dll

MSetBounds()

Sets restricting boundaries for the mouse cursor.

Syntax

```
MSetBounds( [<nTop>], [<nLeft>], [<nBottom>], [<nRight>] ) --> NIL
```

Arguments

`<nTop>`

A numeric value indicating the screen coordinate for the top boundary of the restricting rectangle. The default value is 0.

`<nLeft>`

A numeric value indicating the screen coordinate for the left boundary of the restricting rectangle. The default value is 0.

`<nBottom>`

A numeric value indicating the screen coordinate for the bottm boundary of the restricting rectangle. The default value is MaxRow().

`<nRight>`

A numeric value indicating the screen coordinate for the right boundary of the restricting rectangle. The default value is MaxCol().

Return

The function always returns NIL.

Description

The function restricts mouse cursor movement to the rectangle defined with the passed parameters. If no parameter is passed to MSetBounds(), the function releases all boundaries set. MSetBound() is a compatibility function for text mode and full screen applications.

Info

See also:	MPresent(), MRestState(), MSaveState(), MSetCursor()
Category:	Mouse functions
Source:	rtl\mouseapi.c
LIB:	xhb.lib
DLL:	xhbdll.dll

MSetCursor()

Determines the visibility of the mouse cursor.

Syntax

```
MSetCursor( [<lOnOff>] ) --> lPreviousSetting
```

Arguments

<lOnOff>

A logical value can be passed. .T. (true) makes the mouse cursor visible and .F. (false) hides it.

Return

The function returns a logical value indicating the visibility of the mouse cursor before MSetCursor() is called.

Description

MSetCursor() is used to change the visibility of the mouse cursor. This is only relevant for full screen text mode applications that use SaveScreen() for saving and the screen buffer. The mouse cursor must be hidden before SaveScreen() is called.

Info

See also:	MPresent(), MRestState(), MSaveState(), MSetBounds(), SetMouse()
Category:	Mouse functions
Source:	rtl\mouseapi.c
LIB:	xhb.lib
DLL:	xhbdll.dll

MSetPos()

Moves the mouse cursor to a new position on screen.

Syntax

```
MSetPos( <nRow>, <nCol> ) --> NIL
```

Arguments

<nRow>

A numeric value between 0 and MaxRow() specifying the new row position of the mouse cursor.

<nCol>

A numeric value between 0 and MaxCol() specifying the new columnn position of the mouse cursor.

Return

The return value is always NIL.

Description

MSetPos() is used in full screen or console window applications to move the mouse cursor to a new position on the screen. It is a compatibility function.

Info

See also:	MCol(), MRow(), MSetBounds(), MSetCursor(), SetMouse()
Category:	Mouse functions
Source:	rtl\mouseapi.c
LIB:	xhb.lib
DLL:	xhbdll.dll

MShow()

Displays the mouse cursor.

Syntax

```
MShow() --> NIL
```

Return

The function returns always NIL.

Description

The function MShow() exists for compatibility reasons only and is not recommended to use. Displaying the mouse cursor is only relevant when an application runs in full screen mode. Use function MSetCursor() for hiding/displaying the mouse cursor in such an application.

Info

See also:	MHide(), MSetCursor()
Category:	Mouse functions
Source:	rtl\mouseapi.c
LIB:	xhb.lib
DLL:	xhbdll.dll

NationMsg()

Returns application specific messages.

Syntax

```
NationMsg( <nMessageID> ) --> cMessage
```

Arguments

<nMessageID>

This is a numeric code identifying the message to retrieve.

Return

The function returns a character string identified by its numeric identifier according to the currently selected national language.

Description

NationMsg() is part of xHarbour's language specific system. The returned character string depemds on the currently selected national language (see HB_LangSelect()).

The function is used internally by text-mode commands and functions providing a user interface, such as @...GET, Browse() or MemoEdit(). Messages displayed by such commands and functions indicate the Insert/Overstrike mode during editing, or the Deleted flag of a database record. This information is retrieved internally with function NationMsg().

NationMsg() is available for programmers who want to replace built-in user interface functions and re-use the built-in national language messages.

Info

See also: HB_LangName(), HB_LangErrMsg(), HB_LangMessage(), HB_LangSelect()
Category: Language specific, xHarbour extensions
Source: rtl\natmsg.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays all available messages that can be retrieved
// from NationMsg() in English, French, German and Spanish language

   REQUEST HB_LANG_DE          // request German language
   REQUEST HB_LANG_ES          // request Spanish language
   REQUEST HB_LANG_FR          // request French language

   PROCEDURE Main
      LOCAL i

      CLS                      // English language
      ? HB_LangName()

      FOR i:=1 TO 13
         ? NationMsg(i)
      NEXT
      WAIT

      HB_LangSelect("FR")      // French language
      CLS
```

```
   ? HB_LangName()

   FOR i:=1 TO 13
      ? NationMsg(i)
   NEXT
   WAIT

   HB_LangSelect("DE")       // German language
   CLS
   ? HB_LangName()

   FOR i:=1 TO 13
      ? NationMsg(i)
   NEXT
   WAIT

   HB_LangSelect("ES")       // Spanish language
   CLS
   ? HB_LangName()

   FOR i:=1 TO 13
      ? NationMsg(i)
   NEXT
RETURN
```

NetAppend()

Appends a new record to a database open in shared mode in a work area.

Syntax

```
NetAppend( [<nSeconds>], [<lReleaseLocks>] ) --> lSuccess
```

Arguments

`<nSeconds>`

Optionally, a numeric value can be passed specifying the amount of seconds to retry the operation before the function returns without success. It defaults to the return value of SetNetDelay().

`<lReleaseLocks>`

This parameter defaults to .T. (true). It causes all pending record locks be released before a new record is appended to the database. Pass .F. (false) to keep all record locks intact.

Return

The function returns .T. (true) when a new record is successfully added to the database, otherwise .F. (false) is returned.

Description

NetAppend() is a network function used to add a blank record to a database file open in shared mode for multi-user access. If the record cannot be added due to the file being locked by another process, the function retries the operation for a maximum of *<nSeconds>* seconds and prompts the user if the operation should be retried or aborted.

The function uses DbAppend() and implements additional default logic for database usage in a network when a new record cannot be added at the moment. Refer to DbAppend() for more information on adding a new database record.

Info

See also:	APPEND BLANK, DbAppend(), NetCommitall(), NetDbUse(), NetOpenFiles()
Category:	Database functions, Network functions, xHarbour extensions
Source:	rtl\ttable.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

NetCancel()

Releases an existing network connection.

Syntax

```
NetCancel( <cLocalDevice> ) --> lIsReleased
```

Arguments

<cLocalDevice>

This is a character string holding the name of the network device to release (e.g. "LPT1" or "K:").

Return

The function returns .T. (true) when the network device is successfully released, otherwise .F. (false).

Info

See also:	NetRedir()
Category:	CT:Network, Network functions
Source:	ct\ctnet.c
LIB:	xhb.lib
DLL:	xhbdll.dll

NetCommitAll()

Writes database and index buffers of all used work areas to disk.

Syntax

```
NetCommitAll() --> MAX_TABLE_AREAS
```

Return

The function returns the value of the #define constant MAX_TABLE_AREAS as a numeric value (see TTable.ch).

Description

The NetCommitAll() function writes buffers of all work areas to disk and releases all pending record locks. It iterates the work areas from 1 to MAX_TABLE_AREAS.

Info

See also:	CLOSE, DbCloseAll(), DbCommit(), NetLock(), NetRecLock()
Category:	Database functions, Network functions, xHarbour extensions
Header:	ttable.ch
Source:	rtl\ttable.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

NetDbUse()

Opens a database file for shared access in a work area.

Syntax

```
NetDbUse( <cDatabase>, ;
         [<cAlias>]  , ;
         [<nSeconds>], ;
         [<cRddName>], ;
         [<lNewArea>], ;
         [<lShared>] , ;
         [<lReadOnly>] ) --> lSuccess
```

Arguments

<cDatabase>

This is a character string holding the name of the database file to open. It can include path and file extension. The default file extension is DBF.

<cAlias>

This is the symbolic alias name of the work area as a character string. It defaults to the file name of *<cDatabase>* without extension.

<nSeconds>

Optionally, a numeric value can be passed specifying the amount of seconds to retry the operation before the function returns without success. It defaults to the return value of SetNetDelay().

<cRddName>

<cRddName> is an optional character string with the name of the RDD to use for opening the database file. It defaults to the return value of RddSetDefault().

<lNewArea>

This parameter defaults to .T. (true) which causes the function to open the database in the next unused work area. If .F. (false) is passed, the function opens the database in the current work area. If the current work area is used, all files are closed before the new database is opened.

<lShared>

This parameter defaults to .T. (true) which opens the database in SHARED mode. The the SET EXCLUSIVE setting is ignored.

<lReadOnly>

Specifying .T. (true) for *<lReadOnly>* opens the database in READONLY mode. The default value is .F. (false) which opens the file for read and write access.

Return

The function returns .T. (true) when the database is successfully opened, otherwise .F. (false) is returned.

Description

NetDbUse() is a network function used to open a database file in shared mode for multi-user access. If the database cannot be opened due to the file being locked by another process, the function retries the operation for a maximum of *<nSeconds>* seconds and prompts the user if the operation should be retried or aborted.

The function uses DbUseArea() and implements additional default logic for database usage in a network when the database cannot be opened at the moment. Refer to DbUseArea() for more information on opening a database file.

Info

See also: DbCloseArea(), DbUseArea(), NetOpenFiles(), RddSetDefault(), Select(), SET DEFAULT, SET PATH, Set()
Category: Database functions, Network functions, xHarbour extensions
Source: rtl\ttable.prg
LIB: xhb.lib
DLL: xhbdll.dll

NetDelete()

Marks records for deletion.

Syntax

```
NetDelete() --> lSuccess
```

Return

The function returns .T. (true) when the current database record is successfully marked for deletion, otherwise .F. (false) is returned.

Description

NetDelete() is a network function used in shared database access. The function marks the current database record as deleted. If the record cannot be marked for deletion due to the record being locked by another process, the function retries the operation for a maximum of SetNetDelay() seconds and prompts the user if the operation should be retried or aborted.

The function uses DbDelete() and implements additional default logic for record deletion in a network when the record cannot be marked for deletion at the moment. Refer to DbDelete() for more information on deleting records.

Info

See also:	DbDelete(), DELETE, Deleted(), NetRecall(), PACK, RECALL
Category:	Database functions, Network functions, xHarbour extensions
Source:	rtl\ttable.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

NetDisk()

Tests if a drive is a network drive.

Syntax

```
NetDisk( <cDrive> ) --> lIsNetworkDrive
```

Arguments

<cDrive>

This is a single character (A-Z), followed by a colon, specifying the disk drive to test.

Return

The function returns .T. (true) when the specified drive is a network drive, otherwise .F. (false).

Info

See also:	NetRedir()
Category:	CT:Network, Network functions
Source:	ct\ctnet.c
LIB:	xhb.lib
DLL:	xhbdll.dll

NetErr()

Determines if a database command has failed in network operation.

Syntax

```
NetErr( [<lNewStatus>] ) --> lIsNetworkError
```

Arguments

<lNewStatus>

An optional logical value that sets the global status for errors which may occur with database operations in a network.

Return

The function returns .T. (true) if a database could not be opened or a new record could not be created during concurrent access in a network. Otherwise, the return value is .F. (false).

Description

NetErr() is a function used to detect database errors that may occur with the commands USE, USE...EXCLUSIVE or APPEND BLANK when a database is concurrently accessed by multiple applications in a network. The function maintains a status variable indicating such errors. The status variable is set by xHarbour's default error handling system in following situations:

Possible NetErr() conditions

Failed command	Description
USE	USE EXCLUSIVE is issued by another process
USE...EXCLUSIVE	USE or USE EXCLUSIVE is issued by another process
APPEND BLANK	LastRec()+1 is locked with FLock() or Rlock() by another process

NetErr() returns .T. (true) if a database command failed due to these conditions.

Info

See also: APPEND BLANK, DbAppend(), DbRLock(), DbUseArea(), FLock(), NetName(), RLock(), USE
Category: Database functions, Network functions
Source: rdd\dbcmd.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example illustrates the situations where NetErr() is used
// to detect possible database access errors in a network:

   PROCEDURE Main

      USE Customer ALIAS Cust NEW EXCLUSIVE
      IF .NOT. NetErr()
         INDEX ON CustID                     TO Cust01
         INDEX ON Upper(LastName+FirstName) TO Cust02
         USE
      ELSE
         ? "File cannot be opened exclusively"
      ENDIF
```

```
   USE Customer ALIAS Cust SHARED NEW
   IF .NOT. NetErr()
       SET INDEX TO Cust01, Cust02
   ELSE
      ? "File is exclusively used by another process"
   ENDIF

   IF Select( "Cust" ) > 0
      SELECT Cust
      APPEND BLANK
      IF .NOT. NetErr()
         REPLACE CustID WITH "X087"  , ;
                 Name   WITH "Miller"
      ELSE
         ? "Record is currently locked by another process"
      ENDIF
   ENDIF

RETURN
```

NetError()

Determines if a Net*() function has failed.

Syntax

```
NetError()  --> lIsNetworkError
```

Return

The function returns .T. (true) if a Net*() function has failed, otherwise the return value is .F. (false).

Description

NetError() returns the status flag used by other Net*() functions to indicate success or failure of a database operation in a network.

Info

See also:	NetAppend(), NetDbUse(), NetDelete(), NetOpenFiles(), NetRecall()
Category:	Database functions, Network functions, xHarbour extensions
Source:	rtl\ttable.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

NetFileLock()

Applies a file lock to an open, shared database.

Syntax

```
NetFileLock( [<nSeconds>] ) --> lSuccess
```

Arguments

<nSeconds>

Optionally, a numeric value can be passed specifying the amount of seconds to retry the operation before the function returns without success. It defaults to the return value of SetNetDelay().

Return

The function returns .T. (true) when the current database file is successfully locked, otherwise .F. (false) is returned.

Description

NetFileLock() is a network function used to obtain a lock on the database file open in shared mode for multi-user access. If the database file cannot be locked due to the file being used by another process, the function retries the operation for a maximum of *<nSeconds>* seconds and prompts the user if the operation should be retried or aborted.

The function uses FLock() and implements additional default logic for database locking in a network when the database cannot be locked at the moment. Refer to FLock() for more information on locking a database file.

Note: call function DbUnlock() to release the file lock after data is written to the database file.

Info

See also:	DbRLock(), DbUnlock(), DbUnlockAll(), DbUseArea(), FLock(), RLock(), SET EXCLUSIVE, UNLOCK, USE
Category:	Database functions, Network functions, xHarbour extensions
Source:	rtl\ttable.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

NetFunc()

Evaluates a code block until a timeout period expires.

Syntax

```
Netfunc( <bCodeblock> [, <nSeconds>] ) --> lSuccess
```

Arguments

<bCodeblock>

This is a code block which must return a logical value. The function returns when the code block yields .T. (true), and evaluates the code block again when it returns .F. (false)

<nSeconds>

Optionally, a numeric value can be passed specifying the amount of seconds to retry the operation before the function returns without success. It defaults to the return value of SetNetDelay().

Note: specifying zero for *<nSeconds>* means "forever", i.e. there is no timeout period and the code block must yield .T. (true) for the function to return.

Return

The return value is .T. (true) when the code block is successfully evaluated, otherwise .F. (false) is returnd.

Description

NetFunc() is a generic network function which evaluates a code block and monitors a timeout period if the code block evaluates to .F. (false). The code block is repeatedly evaluated until either the timeout period expires or the code block returns .T. (true).

Info

See also:	NetAppend(), NetFileLock(), NetRecLock()
Category:	Database functions, Network functions, xHarbour extensions
Source:	rtl\ttable.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

NetLock()

Applies locks to a database file or record with timeout.

Syntax

```
NetLock( <nType>          , ;
        [<lReleaseLocks>], ;
        [<nSeconds>]       ) --> lSuccess
```

Arguments

<nType>

The following #define constants must be used for *<nType>* to specify the requested locking type. They are listet in the TTable.ch file:

Types for locks in a network

Constant	Value	Description
NET_RECLOCK	1	Obtains a record lock.
NET_FILELOCK	2	Obtains a file lock.
NET_APPEND	3	Adds a new record.

<lReleaseLocks>

This parameter defaults to .F. (false) which leaves all current locks intact. Passing .T. (true) causes all pending record locks be released before the requested operation is performed.

<nSeconds>

Optionally, a numeric value can be passed specifying the amount of seconds to retry the operation before the function returns without success. It defaults to the return value of SetNetDelay().

Return

The function returns .T. (true) when the requested operation is successfully executed, otherwise .F. (false) is returned.

Description

NetLock() is a utility function used by NetAppend(), NetFileLock() and NetRecLock() to obtain different locks for a database open in shared mode.

Info

See also: NetAppend(), NetFileLock(), NetRecLock()
Category: Database functions, Network functions, xHarbour extensions
Header: TTable.ch
Source: rtl\ttable.prg
LIB: xhb.lib
DLL: xhbdll.dll

NetName()

Retrieves the current user name or the computer name.

Syntax

```
NetName( [<lInfo>] ) --> cComputerName | cUserName
```

Arguments

<lInfo>

If *<lInfo>* is set to .T. (true), the function returns the user account name, otherwise it returns the computer name.

Return

The function returns a character string containing either the computer name or the user name. If this information cannot be retrieved, an empty string ("") is returned.

Description

NetName() serves informational purposes and is used when different users log into the same computer, or when multiple computers run the same application in a network.

Info

See also: Os(), Version()
Category: Environment functions, Network functions
Source: rtl\net.c
LIB: xhb.lib
DLL: xhbdll.dll

NetOpenFiles()

Opens databases and associated index files.

Syntax

```
NetOpenFiles( <aFiles> ) --> lSuccess
```

Arguments

```
<aFiles> := { <cDatabase>, <cAlias>, { <cIndex,...> } }
```

This is a two dimensional array with three columns. Each array element contains an array with three elements:

Array elements

Element	Data type	Description
1	C	Name of the database file to open
2	C	Alias name for work area
3	A	Array with index file names

Return

The function returns zero when all files specified with <*aFiles*> are successfully opened, otherwise one of the following error codes is returned:

Error codes of NetOpenFiles()

Value	Description
-1	Database file not found
-2	Database file cannot be opened
-3	Index file not found

Description

NetOpenFiles() is a convenient network function which opens all files specified with <*aFiles*>. The function calls internally NetDbUse() and opens the specified index files when the database file is successfully opened.

Info

See also:	NetDbUse(), OrdListAdd()
Category:	Database functions, Network functions, xHarbour extensions
Source:	rtl\ttable.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

NetPrinter()

Tests if the current printer is a network printer.

Syntax

```
NetPrinter() --> lIsNetworkPrinter
```

Return

The function returns .T. (true) when the printer activated with SET PRINTER is a network printer, otherwise .F. (false).

Info

See also:	NetRedir(), NNetWork(), SET PRINTER
Category:	CT:Network, Network functions
Source:	ct\ctnet.c
LIB:	xhb.lib
DLL:	xhbdll.dll

NetRecall()

Recalls a record previousy marked for deletion.

Syntax

```
NetRecall() --> lSuccess
```

Return

The function returns .T. (true) when the deletion flag of the current database record is successfully removed, otherwise .F. (false) is returned.

Description

NetRecall() is a network function used in shared database access. The function removes the Deleted flag of the current database. If the flag cannot be removed due to the record being locked by another process, the function retries the operation for a maximum of SetNetDelay() seconds and prompts the user if the operation should be retried or aborted.

The function uses DbRecall() and implements additional default logic for recalling a record in a network when the deletion flag cannot be removed from the current record at the moment. Refer to DbRecall() for more information on removing the Deleted flag.

Info

See also:	DbDelete(), DbRecall(), DELETE, Deleted(), NetDelete(), RECALL
Category:	Database functions, Network functions, xHarbour extensions
Source:	rtl\ttable.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

NetRecLock()

Locks the current record for write access.

Syntax

```
NetRecLock( [<nSeconds>] ) --> lSuccess
```

Arguments

<nSeconds>

Optionally, a numeric value can be passed specifying the amount of seconds to retry the operation before the function returns without success. It defaults to the return value of SetNetDelay().

Return

The function returns .T. (true) when the current database record is successfully locked, otherwise .F. (false) is returned.

Description

NetRecLock() is a network function used to obtain a lock on the current database record when the database file is open in shared mode for multi-user access. If the current database record cannot be locked due to the record being locked by another process, the function retries the operation for a maximum of *<nSeconds>* seconds and prompts the user if the operation should be retried or aborted.

The function uses DbRLock() and implements additional default logic for record locking in a network when the record cannot be locked at the moment. Refer to DbRLock() for more information on locking a database record.

Note: call function DbRUnlock(), DbUnlock() or DbUnlockAll() to release the record lock after data is written to the current record.

Info

See also:	DbRLockList(), DbRUnlock(), DbUnlock(), DbUnlockAll(), NetFileLock(), RLock(), UNLOCK
Category:	Database functions, Network functions, xHarbour extensions
Source:	rtl\ttable.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

NetRedir()

Establishes a connection to a server.

Syntax

```
NetRedir( <cLocal>    , ;
          <cServer>   , ;
         [<cPassword>], ;
         [<lShowError>] ) --> lSuccess
```

Arguments

<cLocal>

This is a character string holding the name of the local device to redirect to a server (e.g. "K:").

<cServer>

This is a character string holding the name of the remote device to connect to (e.g. "\\SERVER05\MARKETING").

<cPassword>

Optionally, a character string holding the password for the server connection can be specified.

<lShowError>

This parameter defaults to .F. (false). When .T. (true) is passed, the function generates a runtime error if the connection fails.

Return

The function returns .T. (true) if the connection to the server is successfully established, otherwise .F. (false) is returned.

Info

See also:	NetCancel(), NetRmtName()
Category:	CT:Network, Network functions
Source:	ct\ctnet.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example connects the local drive "K:" to a server

   PROCEDURE Main
     // no error handling
     ? NetRedir( "Z:", "\\SERVER01\DEVELOPER" )

     // runtime error if connection fails
     ? NetRedir( "Z:", "\\SERVER01\DEVELOPER", "Password" , .T. )
   RETURN
```

NetRmtName()

Returns the remote name of a network device.

Syntax

```
NetRmtName( <cLocal> ) --> cRemote
```

Arguments

<cLocal>

This is a character string holding the name of the local device previously connected to a server (e.g. "K:").

Return

The function returns a character string holding the name of the remote network device, or a null string ("") in case of an error.

Info

See also:	NetRedir()
Category:	CT:Network, Network functions
Source:	ct\ctnet.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Network()

Tests for the existence of a network.

Syntax

```
Network() --> lIsNetwork
```

Return

The function returns .T. (true) when a PC LAN/MS-NET or a Novell network can be detected, otherwise .F. (false).

Info

See also:	NNetwork(), NetRedir()
Category:	CT:Network, Network functions
Source:	ct\ctnet.c
LIB:	xhb.lib
DLL:	xhbdll.dll

NextKey()

Returns the next pending key or mouse event.

Syntax

```
NextKey( [<nEventMask>] ) --> nNextInkeyCode
```

Arguments

<nEventMask>

A numeric value specifying the type of events NextKey() should retrieve. #define constants from INKEY.CH must be used for *<nEventMask>*. They are listed below:

Constants for *<nEventMask>*

Constant	Value	Events returned by NextKey()
INKEY_MOVE	1	Mouse pointer moved
INKEY_LDOWN	2	Left mouse button pressed
INKEY_LUP	4	Left mouse button released
INKEY_RDOWN	8	Right mouse button pressed
INKEY_RUP	16	Right mouse button released
INKEY_MMIDDLE	32	Middle mouse button pressed
INKEY_MWHEEL	64	Mouse wheel turned
INKEY_KEYBOARD	128	Key pressed
INKEY_ALL	255	All events are returned

IF *<nEventMask>* is omitted, the current SET EVENTMASK setting is used. If this is not issued, NextKey() returns only keyboard events.

Return

The function returns a numeric value identifying the next keyboard or mouse event that can be retrieved by the Inkey() function.

Description

The NextKey() function returns the next keyboard or mouse event that is pending in the internal input buffers to be retrieved by the Inkey() function. NextKey() does not remove events from the input buffers and does not update the LastKey() value. This way, the input buffers can be polled without actually removing an event.

If only a subset of the Inkey codes is required, NextKey() can be passed a parameter *<nEventMask>* that filters Inkey codes. Only Inkey codes matching *<nEventMask>* are returned from NextKey().

Note: the file INKEY.CH contains numerous symbolic #define constants that identify different key strokes or mouse input. It is recommended to use #define constants for prosessing Inkey codes, rather than using their numeric values.

Info

See also: Inkey(), KEYBOARD, LastKey(), SET EVENTMASK, SET TYPEAHEAD
Category: Keyboard functions, Mouse functions
Source: rtl\inkey.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// In the example, two characters are stuffed into the keyboard buffer
// and the results of Inkey(), LastKey() and NextKey() are shown

   #include "Inkey.ch"

   PROCEDURE Main
      KEYBOARD "19"

      ? Chr( NextKey() )              // result: 1
      ? Chr( LastKey() )              // result: Chr(0)

      ? Chr( Inkey()   )              // result: 1
      ? Chr( LastKey() )              // result: 1

      ? Chr( NextKey() )              // result: 9
      ? Chr( LastKey() )              // result: 1

      ? Chr( Inkey()   )              // result: 9
      ? Chr( LastKey() )              // result: 9
      ? Chr( NextKey() )              // result: Chr(0)
    RETURN
```

NNetwork()

Tests for the existence of a Novell network.

Syntax

```
NNetwork() --> lIsNovellNetwork
```

Return

The function returns .T. (true) when a Novell network can be detected, otherwise .F. (false).

Info

See also:	Network(), NetRedir()
Category:	CT:Network, Network functions
Source:	ct\ctnet.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Notify()

Resumes a single thread blocked by a particular Mutex.

Syntax

```
Notify( <pMutexHandle>, [<xValue>] ) --> NIL
```

Arguments

`<pMutexHandle>`

This is the handle of the Mutex to notify. It is the return value of HB_MutexCreate().

`<xValue>`

Optionally, an arbitrary value can be passed as second parameter. It defines the return value of Subscribe() and, thus, is passed to the thread that is blocked by the Mutex.

Return

The function returns always NIL.

Description

Function Notify() sends a notification to the next thread having subscribed to a Mutex. The second parameter *<xValue>* defines the return value for the function Subscribe() being executed in a second thread. If more than one thread have subscribed to the Mutex *<pMutexHandle>*, the operating system selects the thread to resume exexuting program code. It is not possible to notify a particular thread waiting for a Mutex.

Info

See also:	GetCurrentThread(), GetThreadID(), HB_MutexCreate(), HB_MutexLock(), NotifyAll(), StartThread(), Subscribe(), SubscribeNow()
Category:	Multi-threading functions, Mutex functions, xHarbour extensions
Source:	vm\thread.c
LIB:	xhbmt.lib
DLL:	xhbmtdll.dll

NotifyAll()

Resumes all threads blocked by a particular Mutex.

Syntax

```
NotifyAll( <pMutexHandle> ) --> NIL
```

Arguments

<pMutexHandle>

This is the handle of the Mutex to notify. It is the return value of HB_MutexCreate().

Return

The function returns always NIL.

Description

Function NotifyAll() sends a nodification to all threads having subscribed for a Mutex. As a result, all threads waiting for a notification on *<pMutexHandle>* resume program excution.

Info

See also:	GetCurrentThread(), GetThreadID(), HB_MutexCreate(), HB_MutexLock(), Notify(), StartThread(), Subscribe(), SubscribeNow()
Category:	Multi-threading functions, Mutex functions, xHarbour extensions
Source:	vm\thread.c
LIB:	xhbmt.lib
DLL:	xhbmtdll.dll

Example

```
// The example starts 5 threads which are suspended immediately after
// being started by Subscribe(). When the Main thread has created the
// threads, they are resumed after a key stroke. The Main thread then
// waits for the end of all threads before the program terminates.

   PROCEDURE Main
      LOCAL i, imax := 5
      LOCAL pMutex := HB_MutexCreate()

      CLS

      FOR i:=1 TO imax
         StartThread( "RunInThread", pMutex, 2*i )
         ThreadSleep( 50 )
      NEXT

      WAIT "Press a key to notify all threads..."

      NotifyAll( pMutex )

      WaitForThreads()
   RETURN


   PROCEDURE RunInThread( pMutex, nRow )
      LOCAL nCol

      @ nRow, 0 SAY "Thread #" + LTrim( Str(GetThreadID()) )
```

```
      nCol := Col()

      Subscribe( pMutex )

      DO WHILE nCol < MaxCol()
         DispoutAt( nRow, ++nCol, "-" )
         ThreadSleep(10)
      ENDDO
   RETURN
```

NtoC()

Converts an integer to a string of digits for a specified number base.

Syntax

```
NtoC( <nInteger>|<cHex>, ;
     [<nBase>]         , ;
     [<nLength>        , ;
     [<cPad>]            ) --> cString
```

Arguments

`<nInteger>`

A numeric 32-bit integer value can be specified as first parameter.

`<cHex>`

Alternatively, the integer can be passed as a hex-encoded character string (see NumToHex()).

`<nBase>`

This is a numeric value specifying the base for the conversion. It must be in the range between 2 and 36, and defaults to base 10.

`<nLength>`

Optionally, the length of the returned string can be specified as a numeric value.

`<cPad>`

If *<mLength>* is larger than the required string, the return string is padded on the left with *<cPad>* up to the desired length. The default value for *<cPad>* ia a blank space (Chr(32)).

Return

The function returns a character string. It holds the digit representation of *<nIneger>* for the specified number base. If there is an error, or *<nLength>* is specified too small, a character string filled with asterisks is returned.

Info

See also: CtoN(), NumToHex()
Category: CT:NumBits, Numbers and Bits
Source: ct\numconv.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays results of NtoC() for different
// number bases.

   PROCEDURE Main
      ? NtoC( 13 )         // result: 13
      ? NtoC( 13, 2  )     // result: 1101
      ? NtoC( 13, 16 )     // result: D
   RETURN
```

NtoCDoW()

Converts a numeric week day to its name.

Syntax

```
NtoCDoW( <nDayOfWeek> ) --> cDayName
```

Arguments

<nDayOfWeek>

A numeric value in the range between 1 and 7 specifies the day number. Numbering days of a week starts with Sunday = 1 (see Dow()).

Return

The function returns the name of the specified week day as a character string.

Info

See also:	CDow(), CtoDoW()
Category:	CT:DateTime, Date and time
Source:	ct\dattime2.c
LIB:	xhb.lib
DLL:	xhbdll.dll

NtoCMonth()

Converts a numeric month to its name.

Syntax

```
NtoCMonth( <nMonth> ) --> cMonthName
```

Arguments

<nMonth>

This is a numeric value in the range of 1 to 12 specifying the month.

Return

The function returns the name of the specified month as a character string.

Info

See also:	CMonth(), CtoMonth()
Category:	CT:DateTime, Date and time
Source:	ct\dattime2.c
LIB:	xhb.lib
DLL:	xhbdll.dll

NtoColor()

Converts a numeric color attribute to a color string.

Syntax

```
NtoColor( <nColor> , [<lAsChar>]) --> cColor
```

Arguments

<nColor>

This is a numeric color attribute in the range between 0 and 255.

<lAsChars>

If set to .T. (true), the function returns a SetColor() compliant character string. The default is .F. (false), i.e. the returned string encodes colors as digits.

Return

The function returns a color string encoding forground and background color as characters or digits.

Info

See also: ColorToN()
Category: CT:Video, Screen functions
Source: ct\color.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays all 256 colors.

   PROCEDURE Main
      LOCAL nRow, nCol, nColor, cColor

      nRow := 0
      nCol := 0

      FOR nColor := 0 TO 255
         cColor := NtoColor( nColor, .T. )

         @ nRow, nCol SAY PadC( cColor, 8 ) COLOR (cColor)

         IF ++nRow > MaxRow()
            nRow := 0
            nCol += 8
         ENDIF
      NEXT
   RETURN
```

Nul()

Returns a null string ("").

Syntax

```
Nul( [<xValue>] ) --> cNullString
```

Arguments

<xValue>

Any value can be passed

Return

The function returns a null string, i.e. a character string with zero characters.

Info

See also: Blank(), Empty()
Category: CT:Miscellaneous, Miscellaneous functions
Source: ct\misc2.c
LIB: xhb.lib
DLL: xhbdll.dll

NumAND()

Performs bitwise AND operations for a list of integer values.

Syntax

```
NumAND( <nInteger1|cHex1> , <nInteger2|cHex2> , ;
       [<...nIntegerN|cHexN>]) --> nInteger
```

Arguments

`<nInteger1>...<nIntegerN>`

A comma separated list of numeric 32-bit integer values must be passed. A minimum of two parameters is required.

`<cHex1>...<cHexN>`

Alternatively, the integers can be passed as hex-encoded character strings (see NumToHex()).

Return

The function performs a bitwise AND operation with the bits of all parameters, and returns the result as a numeric integer value.

Info

See also:	& (bitwise AND), IsBit(), NumAndX(), NumNOT(), NumOR(), NumXOR(), SetBit()
Category:	CT:NumBits, Bitwise functions, Numbers and Bits
Source:	ct\bit1.c
LIB:	xhb.lib
DLL:	xhbdll.dll

NumAndX()

Performs bitwise AND operations for a list of integer values.

Syntax

```
NumAND( <nMaxBits>, ;
        <nInteger1|cHex1>, <nInteger2|cHex2> , ;
       [<...nIntegerN|cHexN>]) --> nInteger
```

Arguments

<nMaxBits>

This numeric value specifies the number of bits to use in the operation, beginning with the least significant bit (LSB). The value of *<nMaxBits>* must be in the range between 1 and 32.

<nInteger1>...<nIntegerN>

A comma separated list of numeric 32-bit integer values must be passed. A minimum of two integers is required.

<cHex1>...<cHexN>

Alternatively, the integers can be passed as hex-encoded character strings (see NumToHex()).

Return

The function performs a bitwise AND operation with the bits of all parameters, and returns the result as a numeric integer value.

Info

See also:	& (bitwise AND), IsBit(), NumAnd(), NumNotX(), NumOrX(), NumXorX(), SetBit()
Category:	CT:NumBits, Bitwise functions, Numbers and Bits, xHarbour extensions
Source:	ct\bit3.c
LIB:	xhb.lib
DLL:	xhbdll.dll

NumAt()

Counts multiple occurrences of a substring within a string.

Syntax

```
NumAt( <cSearch>   , ;
       <cString>   , ;
      [<nSkipChars>] ) --> nCount
```

Arguments

<cSearch>

This is a character string to search for in *<cString>*.

<cString>

This is the character string to find *<cSearch>* in.

<nSkipChars>

This numeric parameter defaults to 0 so that the function begins searching with the first character of *<cString>*. Passing a value > 0 instructs the function to ignore the first *<nSkipChars>* characters in the search.

Return

The function returns the number of times *<cSearch>* is found in *<cString>* as a numeric value.

Info

See also:	AfterAtNum(), AtNum(), BeforAtNum(), CSetAtMuPa(), SetAtLike()
Category:	CT:String manipulation, Character functions
Source:	ct\numat.c
LIB:	xhb.lib
DLL:	xhbdll.dll

NumButtons()

Returns the number of mouse buttons.

Syntax

```
NumButtons() --> nButtonCount
```

Return

The function returns the number of mouse buttons available.

Description

NumButtons() is used to determine how many buttons a mouse is equipped with.

Info

See also:	MSetCursor(), SetMouse()
Category:	Mouse functions
Source:	rtl\mousex.c
LIB:	xhb.lib
DLL:	xhbdll.dll

NumCount()

Installs and/or increments an internal counter value.

Syntax

```
NumCount( [<nValue>], [<lIsStart>] ) --> nCounter
```

Arguments

<nValue>

This is an optional numeric value. It increments the internal counter when *<lIsStart>* is .F. (false), which is the default.

<lIsStart>

If .T. (true) is passed, *<nValue>* defines the start value for the internal counter.

Return

The function returns the internal counter as a numeric value. If the function is called without parameters, the return value is the current value of the internal counter.

Info

See also:	FOR EACH, HB_EnumIndex()
Category:	CT:NumBits, Numbers and Bits
Source:	ct\numcount.c
LIB:	xhb.lib
DLL:	xhbdll.dll

NumDiskL()

Returns the number of logical drives.

Syntax

```
NumDiskL() --> nLogicalDrives
```

Return

This function exists for compatibility reasons only. It returns always 26 as a numeric value.

Info

Category: CT:DiskUtil, Disks and Drives
Source: ct\disk.c
LIB: xhb.lib
DLL: xhbdll.dll

NumHigh()

Retrieves the high byte of a 16-bit integer.

Syntax

```
NumHigh( <nInteger>|<cHex> ) --> nHighByte
```

Arguments

<nInteger>

A numeric 16-bit integer value can be specified as first parameter.

<cHex>

Alternatively, the integer can be passed as a hex-encoded character string (see NumToHex()).

Return

The function returns the high byte of a 16-bit integer as a numeric value.

Info

See also: NumLow()
Category: CT:NumBits, Numbers and Bits
Source: ct\numlohi.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays the high byte of a 16-bit integer.

   PROCEDURE Main
      LOCAL nValue := HexToNum( "ABCD" )
      LOCAL nByte

      ? nValue                       // result: 43981
      ? nByte := NumHigh( nValue )   // result: 171
      ? NumToHex( nByte )            // result: AB

      ? nByte := NumLow( nValue )
      ? NumToHex( nByte )
   RETURN
```

NumLine()

Returns the number of lines in an ASCII formatted character string.

Syntax

```
NumLine( <cText>, [<nLineLength>] ) --> nLineCount
```

Arguments

`<cText>`

This is the ASCII formatted character string to process.

`<nLineLength>`

Optionally, the line length for formatting can be specified as a numeric value. It defaults to 80 characters per line.

Return

The function returns the number of lines needed to output an ASCII formatted text string (fixed font).

Note: the tab character (Chr(9)) is counted as a single character. If it exists in a text string, the text should be converted with TabExpand() before NumLine() is called.

Info

See also:	MaxLine(), MlCount(), TabExpand()
Category:	CT:String manipulation, Character functions
Source:	ct\numline.c
LIB:	xhb.lib
DLL:	xhbdll.dll

NumLow()

Retrieves the low byte of a 16 bit integer.

Syntax

```
NumLow( <nInteger>|<cHex> ) --> nLowByte
```

Arguments

`<nInteger>`

A numeric 16-bit integer value can be specified as first parameter.

`<cHex>`

Alternatively, the integer can be passed as a hex-encoded character string (see NumToHex()).

Return

The function returns the low byte of a 16-bit integer as a numeric value.

Info

See also: NumHigh()
Category: CT:NumBits, Numbers and Bits
Source: ct\numlohi.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays the low byte of a 16-bit integer.

   PROCEDURE Main
      LOCAL nValue := HexToNum( "ABCD" )
      LOCAL nByte

      ? nValue                       // result: 43981
      ? nByte := NumLow( nValue )    // result: 205
      ? NumToHex( nByte )            // result: CD

   RETURN
```

NumMirr()

Mirrors 8 or 16 bits of a 16-bit integer.

Syntax

```
NumMirr( <nInteger>|<cHex>, ;
        [<lEightBits>]      ) --> nInteger
```

Arguments

`<nInteger>`

A numeric 16-bit integer value in the range between 0 and 65535 can be specified as first parameter.

`<cHex>`

Alternatively, the integer can be passed as a hex-encoded character string (see NumToHex()).

`<lEightBits>`

If .T. (true) is passed for *<lEightBits>*, only the low 8 bits of *<nInteger>* are mirrored. The default value is .F. (false), i.e. all 16 bits are mirrored.

Return

The function returns a numeric integer value with 16 or 8 bits set in reversed order.

Info

See also: NumAND(), NumHigh(), NumLow(), NumMirrX(), NumNOT(), NumOR(), NumRoL()
Category: CT:NumBits, Numbers and Bits
Source: ct\bit1.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays results of NumMirr() and their binary
// representation.

   PROCEDURE Main
      LOCAL nValue := 4209

      ? nValue                                // result:  4209
      ? NtoC( nValue, 2, 16, "0" )
                                   // result: 0001000001110001

      ? NumMirr( nValue )                     // result: 36360
      ? NtoC( NumMirr( nValue ), 2, 16, "0" )
                                   // result: 1000111000001000

      ? NumMirr( nValue, .T. )                 // result: 4238
      ? NtoC( NumMirr( nValue, .T. ), 2, 16, "0" )
                                   // result: 0001000010001110
   RETURN
```

NumMirrX()

Mirrors bits of a numeric integer value.

Syntax

```
NumMirr( <nMaxBits>, ;
         <nInteger>|<cHex> ) --> nInteger
```

Arguments

<nMaxBits>

This numeric value specifies the number of bits to use in the operation, beginning with the least significant bit (LSB). The value of *<nMaxBits>* must be in the range between 1 and 32.

<nInteger>

A numeric integer value must be passed as second parameter.

<cHex>

Alternatively, the integer can be passed as a hex-encoded character string (see NumToHex()).

Return

The function returns a numeric integer value with the least significant *<nMaxBits>* bits set in reversed order.

Info

See also:	NumAndX(), NumMirr(), NumNotX(), NumOrX(), NumRolX()
Category:	CT:NumBits, Numbers and Bits, xHarbour extensions
Source:	ct\bit3.c
LIB:	xhb.lib
DLL:	xhbdll.dll

NumNOT()

Performs a bitwise NOT operation with a numeric integer value.

Syntax

```
NumNOT( <nInteger>|<cHex> ) --> nNOTed_Integer
```

Arguments

<nInteger>

A numeric 32-bit integer value can be specified as first parameter.

<cHex>

Alternatively, the integer can be passed as a hex-encoded character string (see NumToHex()).

Return

The function returns a numeric integer value with all bits inverted.

Info

See also:	IsBit(), NumAND(), NumNotX(), NumOR(), NumXOR(), SetBit()
Category:	CT:NumBits, Bitwise functions, Numbers and Bits
Source:	ct\bit1.c
LIB:	xhb.lib
DLL:	xhbdll.dll

NumNotX()

Performs a bitwise NOT operation with a numeric integer value.

Syntax

```
NumNOT( <nMaxBits>, ;
        <nInteger>|<cHex> ) --> nNOTed_Integer
```

Arguments

<nMaxBits>

This numeric value specifies the number of bits to use in the operation, beginning with the least significant bit (LSB). The value of *<nMaxBits>* must be in the range between 1 and 32.

<nInteger>

A numeric integer value is passed as second parameter.

<cHex>

Alternatively, the integer can be passed as a hex-encoded character string (see NumToHex()).

Return

The function returns a numeric integer value with *<nMaxBits>* bits inverted.

Info

See also:	IsBit(), NumAndX(), NumNot(), NumOrX(), NumXorX(), SetBit()
Category:	CT:NumBits, Bitwise functions, Numbers and Bits, xHarbour extensions
Source:	ct\bit3.c
LIB:	xhb.lib
DLL:	xhbdll.dll

NumOR()

Performs a bitwise OR for a list of integer values.

Syntax

```
NumOR( <nInteger1|cHex1> , <nInteger2|cHex2> , ;
       [<...nIntegerN|cHexN>]) --> nInteger
```

Arguments

`<nInteger1>...<nIntegerN>`

A comma separated list of numeric 32-bit integer values must be passed. A minimum of two parameters is required.

`<cHex1>...<cHexN>`

Alternatively, the integers can be passed as hex-encoded character strings (see NumToHex()).

Return

The function performs a bitwise OR operation with the bits of all parameters, and returns the result as a numeric integer value.

Info

See also:	| (bitwise OR), IsBit(), NumAND(), NumNOT(), NumOrX(), NumXOR(), SetBit()
Category:	CT:NumBits, Bitwise functions, Numbers and Bits
Source:	ct\bit1.c
LIB:	xhb.lib
DLL:	xhbdll.dll

NumOrX()

Performs a bitwise OR for a list of integer values.

Syntax

```
NumOR( <nMaxBits>, ;
       <nInteger1|cHex1> , <nInteger2|cHex2> , ;
       [<...nIntegerN|cHexN>]) --> nInteger
```

Arguments

<nMaxBits>

This numeric value specifies the number of bits to use in the operation, beginning with the least significant bit (LSB). The value of *<nMaxBits>* must be in the range between 1 and 32.

<nInteger1>...<nIntegerN>

A comma separated list of numeric integer values must be passed. A minimum of two integers is required.

<cHex1>...<cHexN>

Alternatively, the integers can be passed as hex-encoded character strings (see NumToHex()).

Return

The function performs a bitwise OR operation with *<nMaxBits>* bits of all parameters, and returns the result as a numeric integer value.

Info

See also:	| (bitwise OR), IsBit(), NumAndX(), NumNotX(), NumOR(), NumXorX(), SetBit()
Category:	CT:NumBits, Bitwise functions, Numbers and Bits, xHarbour extensions
Source:	ct\bit3.c
LIB:	xhb.lib
DLL:	xhbdll.dll

NumRoL()

Rotates bits of a numeric 16-bit integer value to the left.

Syntax

```
NumRoL( <nInteger>|<cHex>   , ;
        <nRotate>|<cRotate>, ;
       [<lEightBits>]        ) --> nInteger
```

Arguments

`<nInteger>`

A numeric 16-bit integer value in the range between 0 and 65535 can be specified as first parameter.

`<cHex>`

Alternatively, the integer can be passed as a hex-encoded character string (see NumToHex()).

`<nRotate>`

A numeric value in the range between 1 and 15 defines how many places the bits should be rotated. If *<lEightBits>* is set to .T. (true), the range is between 1 and 7.

`<cRotate>`

Instead of a numeric value, the parameter can be a hex-encoded character string.

`<lEightBits>`

If .T. (true) is passed for *<lEightBits>*, only the low 8 bits of *<nInteger>* are rotated. The default value is .F. (false), i.e. all 16 bits are rotated.

Return

The function returns the result of the bit rotation as a numeric value.

Info

See also: <<, >>, NumRolX(), NtoC()
Category: CT:NumBits, Bitwise functions, Numbers and Bits
Source: ct\bit1.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays the results of left bit rotation and their
// binary representation. Note that the most significant bit replaces
// the least significant bit in a rotation, while all other bits are
// shifted to the left.

   PROCEDURE Main
      LOCAL nValue := CtoN( "0010100001110000", 2 )

      ? nValue                       // result: 10352
      ? NtoC( nValue, 2, 16, "0" )
                        // result: 0010100001110000

      ? NumRol( nValue, 3 )          // result: 17281
      ? NtoC( NumRol( nValue, 3 ), 2, 16, "0" )
                        // result: 0100001110000001
```

```
   ? NumRol( nValue, 3, .T. )   // result: 10371
   ? NtoC( NumRol( nValue, 3, .T. ), 2, 16, "0" )
                  // result: 0010100010000011
RETURN
```

NumRolX()

Rotates bits of a numeric integer value to the left.

Syntax

```
NumRoL( <nMaxBits>, ;
        <nInteger>|<cHex>   , ;
        <nRotate>|<cRotate>   ) --> nInteger
```

Arguments

<nMaxBits>

This numeric value specifies the number of bits to use in the operation, beginning with the least significant bit (LSB). The value of *<nMaxBits>* must be in the range between 1 and 32.

<nInteger>

A numeric integer value is specified as second parameter.

<cHex>

Alternatively, the integer can be passed as a hex-encoded character string (see NumToHex()).

<nRotate>

A numeric value defines how many places the bits should be rotated.

<cRotate>

Instead of a numeric value, the parameter can be a hex-encoded character string.

Return

The function returns the result of rotating *<nMaxBits>* bits as a numeric value.

Info

See also:	<<, >>, NumROL(), NtoC()
Category:	CT:NumBits, Bitwise functions, Numbers and Bits, xHarbour extensions
Source:	ct\bit1.c
LIB:	xhb.lib
DLL:	xhbdll.dll

NumToHex()

Converts a numeric value or a pointer to a Hex string.

Syntax

```
NumToHex( <nNum>|<pPointer>, [<nLen>]) --> cHexString
```

Arguments

<nNum>|<pPointer>

The first parameter is eiher a numeric value or a pointer to convert to hexadecimal notation.

<nLen>

An optional numeric value specifying the length of the return string.

Return

The function returns a character string holding the passed value in hexadecimal notation.

Description

NumToHex() converts a numeric value or a pointer to a Hex formatted character string. It converts only integer values. If a number has a decimal fraction, it is ignored. The reverse function is HexToNum().

Info

See also:	CStr(), HexToNum(), HexToStr(), StrToHex(), StrZero(), Transform()
Category:	Numeric functions, Conversion functions, xHarbour extensions
Source:	rtl\hbhex2n.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates return values of NumToHex()

   PROCEDURE Main()
     ? NumToHex( 1 )          // result: 1
     ? NumToHex( 128 )        // result: 80

     ? NumToHex( 1   , 4 )    // result: 0001
     ? NumToHex( 128, 4 )     // result: 0080

     ? NumToHex( -128 )       // result: FFFFFFFFFFFFFF80
     ? NumToHex( -1 )         // result: FFFFFFFFFFFFFFFF

     ? NumToHex( -1 , 4 )     // result: FFFF

     ? NumToHex( SQrt(2) )    // result: 1
   RETURN
```

NumToken()

Returns the number of tokens in a string.

Syntax

```
NumToken( <cString>    , ;
         [<cDelimiter>], ;
         [<nSkipWidth>]  ) --> nCount
```

Arguments

`<cString>`

This is a character string to count tokens in.

`<cDelimiter>`

This character string holds a list of characters recognized as delimiters between tokens. The default list of delimiters consist of non-printable characters having the ASCII codes 0, 9, 10, 13, 26, 32, 138 and 141 and the following punctuation characters: ,.;:!?/\<>()#&%+-*

`<nSkipWidth>`

This optional numeric value defaults to zero. This causes the function to count empty tokens. To suppress this behavior, set *<nSkipWidth>* to 1.

Return

The function returns the number of tokens found in a string as a numeric value

Info

See also:	AtToken(), Token(), TokenLower(), TokenUpper()
Category:	CT:String manipulation, Character functions, Token functions
Source:	ct\token1.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays the number of tokens using different delimiters

   PROCEDURE Main
      LOCAL cDate := DtoC( StoD( "20061231" ) )
      LOCAL cTime := Time()
      LOCAL cFile := CurDrive()+ ":\" + CurDir() + "\Test.prg"

      ? NumToken( cDate, "/." )                // result: 3

      ? NumToken( cTime, ":"  )                // result: 3

      ? NumToken( cFile, ":\."  )              // result: 6
   RETURN
```

NumXOR()

Performs a bitwise XOR operation with two numeric 32-bit integer values.

Syntax

```
NumXOR( <nInteger1>|<cHex1>, ;
        <nInteger2>|<cHex2>  ) --> nInteger
```

Arguments

<nInteger>

A numeric 32-bit integer value can be specified for both parameters.

<cHex>

Alternatively, the two integers can be specified as hex-encoded character strings (see NumToHex()).

Return

The function returns the result of an exclusive OR operation with the bits of both parameters as a numeric value. The bits are set in the result where the bits of both parameters are different.

Info

See also:	IsBit(), NumAND(), NumNOT(), NumOR(), NumXorX(), SetBit()
Category:	CT:NumBits, Bitwise functions, Numbers and Bits
Source:	ct\bit1.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays the results of exclusive OR operations and
// their binary representation.

   PROCEDURE Main
      LOCAL nValue1 := 6
      LOCAL nValue2 := 23
      LOCAL nResult

      ? NtoC( nValue1, 2, 8, "0" ) // result: 00000110
      ? NtoC( nValue2, 2, 8, "0" ) // result: 00010111


      nResult := NumXOR( nValue1, nValue2 )
      ? nResult                    // result: 17
      ? NtoC( nResult, 2, 8, "0" ) // result: 00010001

      nResult := NumXOR( nValue1, nResult )
      ? nResult                    // result: 23
      ? NtoC( nResult, 2, 8, "0" ) // result: 00010111

      ? nResult == nValue2         // result: .T.
   RETURN
```

NumXorX()

Performs a bitwise XOR operation with two numeric integer values.

Syntax

```
NumXOR( <nMaxBits>, ;
        <nInteger1>|<cHex1>, ;
        <nInteger2>|<cHex2>  ) --> nInteger
```

Arguments

`<nMaxBits>`

This numeric value specifies the number of bits to use in the operation, beginning with the least significant bit (LSB). The value of *<nMaxBits>* must be in the range between 1 and 32.

`<nInteger1> and <nInteger2>`

Two numeric integer value can be specified for both parameters.

`<cHex1> and <cHex2>`

Alternatively, the two integers can be specified as hex-encoded character strings (see NumToHex()).

Return

The function returns the result of an exclusive OR operation with *<nMaxBits>* bits of both parameters as a numeric value. The bits are set in the result where the bits of both parameters are different.

Info

See also: IsBit(), NumAndX(), NumNotX(), NumOrX(), NumXOR(), SetBit()
Category: CT:NumBits, Bitwise functions, Numbers and Bits, xHarbour extensions
Source: ct\bit1.c
LIB: xhb.lib
DLL: xhbdll.dll

Occurs()

Counts the occurrance of a substring in a character string.

Syntax

```
Occurs( <cSubStr>, <cString> ) --> nCount
```

Arguments

<cSubStr>

This is a character string being searched in *<cString>*.

<cString>

This is a character string holding occurrances of *<cSubStr>*.

Return

The function scans *<cString>* and counts the occurrances of *<cSubStr>*. The number of occurrances is returned as a numeric value.

Info

See also:	At(), StrTran(), SubStr(), Token()
Category:	CT:String manipulation, Character functions
Source:	ct\util.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

OemToHtml()

Inserts HTML character entities into an OEM text string.

Syntax

```
OemToHtml( <cOemString> ) --> cHtmlString
```

Arguments

`<cOemString>`

This is an OEM character string.

Return

The function returns a HTML formatted text string.

Description

Function OemToHtml() inserts HTML character entities into *<cOemString>* and returns the HTML formatted text string. Note that the function processes only HTML character entities (e.g. >, < or &) that are interchangeable in the OEM and ANSI character sets. Refer to AnsiToHtml() for a complete list of HTML character entities recognized by the function.

Info

See also:	AnsiToHtml(), HtmlToAnsi(), HtmlToOem(), THtmlDocument()
Category:	Conversion functions, HTML functions, xHarbour extensions
Source:	tip\thtml.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows a result of OemToHtml()

   PROCEDURE Main
      LOCAL cText := 'Copyright © 2007 xHarbour Inc'

      ? OemToHtml( cText )  // result: Copyright &copy; 2007 xHarbour Inc

   RETURN
```

Ole2TxtError()

Returns the last OLE error code as character string.

Syntax

```
Ole2TxtError() --> cErrorCode
```

Return

The function returns a character string representing the string constant of the last numeric OLE error codeholding an error message. If no error occurred, the return value is "S_OK".

Description

Function Ole2TxtError() is used to obtain the string value of the last numeric OLE error code. It is returned form OleError().

Info

See also:	CreateObject(), OleError()
Category:	OLE Automation, xHarbour extensions
Source:	rtl\win32ole.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

OleError()

Returns the last OLE error code.

Syntax

```
OleError() --> nErrorCode
```

Return

The function returnes the error code of the OLE operation as a numeric value.

Description

When an OLE operation fails, the numeric error code of this operation can be retrieved OleError(). This error code is kept until a new OLE operation is performed. The value zero means "no error"

To obtain a human readable description of an OLE error, call Ole2TxtError().

Info

See also:	CreateObject(), Ole2TxtError()
Category:	OLE Automation, xHarbour extensions
Source:	rtl\win32ole.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

OrdBagExt()

Retrieves the default extension for index files.

Syntax

```
OrdBagExt() --> cFileExtension
```

Return

The function returns the default extension for index files used by the default RDD as a character string.

Description

OrdBagExt() returns the extension for index files used by the RDD that opens a database in the current work area. Different RDDs maintain different default file extensions. They are used when an index file is created and the file name is specified without extension.

Note: OrdBagExt() does not return the file extension of an open index.

Info

See also:	DbTableExt(), DbOrderInfo(), OrdBagName(), OrdCreate(), OrdFor(), OrdKey(), OrdName(), OrdNumber(), RddName()
Category:	Database functions, Index functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays default index file extensions of two
// RDDs available in xHarbour.

   REQUEST Dbfcdx

   PROCEDURE Main

      USE Customer VIA "DBFNTX"

      ? OrdBagExt()             // result: .ntx

      USE VfpCust VIA "DBFCDX"

      ? OrdBagExt()             // result: .cdx

      CLOSE ALL
   RETURN
```

OrdBagName()

Retrieves the file name of an open index.

Syntax

```
OrdBagName( [<nOrder> | <cIndexName>] ) --> cIndexFilename
```

Arguments

<nOrder>

A numeric value specifying the ordinal position of the index open in a work area. Indexes are numbered in the sequence of opening, beginning with 1. The value zero identifies the controlling index.

<cIndexName>

Alternatively, a character string holding the symbolic name of the open index can be passed. It is analogous to the alias name of a work area. Support for *<cIndexName>* depends on the RDD used to open the index. Usually, RDDs that are able to maintain multiple indexes in one index file support symbolic index names, such as DBFCDX, for example.

Return

The function returns the name of the index file that contains the specified index as a character string. If no parameter is passed, the function returns the file name of the controlling index. If no index file is open, an empty string ("") is returned.

Description

OrdBagName() retrieves the name of the index file in which an index is stored. The index can be specified by its numeric ordinal position, or by its symbolic name specified with the TAG option of the INDEX command.

Info

See also:	DbOrderInfo(), OrdBagExt(), OrdCreate(), OrdFor(), OrdKey(), OrdName(), OrdNumber(), RddName()
Category:	Database functions, Index functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example lists results of OrdBagName() using two different RDDs.

   REQUEST Dbfcdx

   PROCEDURE Main
      USE Customer VIA "DBFNTX"
      SET INDEX TO Cust01, Cust02

      ? OrdBagName()              // result: Cust01.ntx
      ? OrdBagName(1)             // result: Cust01.ntx
      ? OrdBagName(2)             // result: Cust02.ntx
      ? OrdBagName(3)             // result: (empty string)

      USE Invoice VIA "DBFCDX"

      SET INDEX TO Invoice
```

```
   ? OrdName() , OrdBagName()  // result: InvNo   Invoice.cdx
   ? OrdName(1), OrdBagName(1) // result: InvNo   Invoice.cdx
   ? OrdName(2), OrdBagName(2) // result: Custno  Invoice.cdx
   ? OrdName(3), OrdBagName(3) // result: (empty string)

   CLOSE ALL
RETURN
```

OrdCondSet()

Sets the conditions for index creation.

Syntax

```
OrdCondSet( [<cForCondition>]  , ;
            [<bForCondition>]  , ;
            [<lAllRecords>]    , ;
            [<bWhileCondition>], ;
            [<bEval>]          , ;
            [<nInterval>]      , ;
            [<nStart>]         , ;
            [<nNext>]          , ;
            [<nRecord>]        , ;
            [<lRest>]          , ;
            [<lDescend>]       , ;
            [<reserved>]       , ;
            [<lAdditive>]      , ;
            [<lCurrent>]       , ;
            [<lCustom>]        , ;
            [<lNoOptimize>]    , ;
            [<cWhileCondition>], ;
            [<lTemporary>]     , ;
            [<lUseFilter>]     , ;
            [<lExclusive>]       ) --> lSucces
```

Arguments

<cForCondition>

A character string holding the FOR expression for the index to create. This string is stored in the index file header and can later be retrieved using the OrdFor() function. Pass an empty string ("") if the index is created without a FOR condition.

The FOR expression cannot exceed 250 characters in length. RDDs that do not support a FOR condition when creating indexes generate a runtime error when this parameter is used.

<bForCondition>

A code block specifying the FOR expression, or NIL if no FOR expression is used. The code block expression must be identical with the expression specified as a character string *<cForCondition>*. The code block is evaluated for all records in the current work area. Those records where *<bForCondition>* yields .T. (true) are included in the index.

<lAll>

If .T. (true) is passed, all records in the current work area are indexed. The default value is .F. (false) so that the scope for indexing can be narrowed with the parameters *<nRecord>* or *<lRest>*.

<bWhileCondition>

A code block containing a logical expression, or NIL if no WHILE condition is used. The index creation continues while the code block returns .T. (true). Index creation is terminated as soon as *<bWhileCondition>* yields .F. (false).

<bWhileCondition> changes the default scope to *<lRest>*. The code block is only used during index creation and not stored in the index file.

<bEval>

A code block that is evaluated every *<nInterval>* records. It is normally used to inform the user about indexing progress and must return a logical value. The indexing operation continues as

long as Eval(<*bBlock*>) returns .T. (true). The operation is stopped when Eval(<*bBlock*>) returns .F. (false).

<nInterval>

This is a numeric value specifying the number of records to index before the code block <*bEval*> is called. The default value is 0, so that <*bEval*> is called for every record.

<nStart>

This is a numeric value specifying the record number to begin the index creation with. The default value is 0, i.e. indexing begins with the first record.

<nNext>

A numeric value restricting the number of records to evaluate during index creation to <*nNext*>. The default value is 0, i.e. there is no restriction.

<nRecord>

A numeric value specifying the record number of a single record to add to the index. The default value is 0, i.e. the single record scope is ignored.

<lRest>

If .T. (true) is passed, only the records from <*nStart*> to the end of file are processed. Passing .F. (false) processes all records. The default value is .T. (true).

<lDescend>

The default value for <*lDescend*> is .F. (false), resulting in records being added to the index in ascending order. If .T. (true) is passed, a descending index is created.

<reserved>

This parameter is reserved for future use.

<lAdditive>

If .T. (true) is passed, an index is created in addition to open indexes. The default value is .F. (false) causing all indexes open in the work area being closed before the new index is created.

<lCurrent>

Passing .T. (true) for <*lCurrent*> instructs the RDD to use the current logical order of records for navigating the database during index creation. The logical order is determined by the controlling index and the OrdScope() restriction. The default value is .F. (false) so that the records in the current work area are evaluated in physical order.

<lCustom>

If .T. (true) is passed, only an empty index is created that is custom built. Index entries must be added or deleted explicitely using functions OrdKeyAdd() and OrdKeyDel(). RDDs that support custom indexes do not add or delete keys automatically. The default value is .F. (false).

<lNoOptimize>

The parameter is only relevant when <*bForCondition*> is specified. Normally, the FOR condition is optimized. Passing .T. (true) suppresses optimization. The default value is .F. (false).

<cWhileCondition>

This is an optional character string holding the WHILE expression for the index to create.

<lTemporary>

If .T. (true) is passed, a temporary index is created which is automatically destroyed when the index is closed. The temporary index may be created in memory only or in a temporary file. This lies in the responsibility of the RDD used for index creation.

`<lUseFilter>`

Passing .T. (true) for *<lUseFilter>* instructs the RDD to recognize a filter condition active in the current work area that is set with SET FILTER or SET DELETED. In this case, filtered records are not included in the index. The default value is .F. (false) so that a filter condition is not recognized during index creation.

`<lExclusive>`

If .T. (true) is passed, the index file is created in EXCLUSIVE file access mode. The default value is .F. (false) so that the index file is created in SHARED mode.

Return

The function returns .T. (true) if the conditions for index creation are set, otherwise .F. (false) is returned.

Description

The OrdCondSet() function specifies the conditions for index creation in the current work area. The conditions override default values and remain active until an index is created. This is accomplished with function OrdCreate(). Consequently OrdCondSet() is immediately followed by a call to OrdCreate().

It is recommended to specify index creation conditions using the INDEX command. This command is preprocessed to OrdCondSet() and OrdCreate(). INDEX is much more comfortable to use for specifying the numerous function parameters.

Info

See also:	DbOrderInfo(), INDEX, OrdCreate()
Category:	Index functions, Database functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

OrdCount()

Determines the number of indexes open in a work area.

Syntax

```
OrdCount( [<cIndexFile>] ) --> nOpenIndexes
```

Arguments

`<cIndexFile>`

This is a character string with the name of the file that stores the indexes. When the file extension is omitted, it is determined by the database driver that manages the file.

Return

The function returns the number of open indexes stored in *<cIndexFile>*. If this parameter is omitted, the return value is the total number of indexes open in a work area.

Description

The function delivers information about the number of indexes open in a work area, or indexes stored in a particular index file. It can be used as loop counter limit when information about all open indexes is collected.

Info

See also:	DbOrderInfo(), OrdCreate(), OrdKey(), OrdNumber()
Category:	Index functions, Database functions, xHarbour extensions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows a user defined function which collects information
// about open indexes in all work areas.

   FUNCTION AllOrders()
      LOCAL nArea     := Select()
      LOCAL aWorkArea := {}
      LOCAL aOrders   := {}
      LOCAL i, imax, j, jmax

      SELECT 0
      imax := Select() - 1

      FOR i:=1 TO imax
         DbSelectArea( i )

         jmax    := OrdCount()
         aOrders := Array( jmax )
         AAdd( aWorkArea, { Alias(), aOrders } )

         FOR j:=1 TO jmax
            aOrders[j] := { OrdBagName(j), OrdName(j) , ;
                            OrdKey(j)    , OrdCustom(j) }
         NEXT
      NEXT
```

```
    DbSelectArea( nArea )
RETURN aWorkArea
```

OrdCreate()

Creates a new index.

Syntax

```
OrdCreate( <cIndexFile> , ;
          [<cIndexName>], ;
           <cIndexExpr> , ;
           <bIndexExpr> , ;
          [<lUnique>]     ) --> NIL
```

Arguments

<cIndexFile>

This is a character string with the name of the file that stores the new index. When the file extension is omitted, it is determined by the database driver that creates the file.

<cIndexName>

This is a character string holding the symbolic name of the index to create in an index file. It is analogous to the alias name of a work area. Support for *<cIndexName>* depends on the RDD used to create the index. Usually, RDDs that are able to maintain multiple indexes in one index file support symbolic index names, such as DBFCDX, for example.

<cIndexExpr>

This is a character expression which describes the index expression in textual form. The expression is evaluated for the records in the current work area. The return value of *<cIndexExpr>* determines the logical order of records when the index is the controlling index. The data type of the index may be Character, Date, Numeric or Logical. The maximum length of an index expression and its value is determined by the replaceable database driver used to create the index.

<bIndexExpr>

It is the same like *<cIndexExpr>* but is specified as a code block that can be evaluated. If omitted, the code block is created from *<cIndexExpr>*.

<lUnique>

If the optional parameter is set to .T. (true), it suppresses inclusion of records that yield duplicate index values. When an index value exists already in an index, a second record resulting in the same index value is not added to the index. When *<lUnique>* is omitted, the current SET UNIQUE setting is used as default.

Return

The function returns NIL.

Description

The index function OrdCreate() creates an index for the database open in the current work area. Use an aliased expression to create an index in a different work area.

Indexes are created by the RDD used to open the database, according to the conditions specified with function OrdCondSet(). If the RDD does not support multiple indexes per index file (DBFNTX, for example) an existing index file *<cIndexFile>* is overwritten. If multiple indexes are supported by an RDD (DBFCDX, for example) and *<cIndexFile>* is open, a new index *<cIndexName>* is added to the file. If *<cIndexName>* exists already in the list of open indexes, it is replaced with a new index having the expression *<cIndexExpr>*.

During index creation, the code block <*bIndexExpr*> is evaluated for each record of the work area and its return value is added to the index. When index creation is complete, the new index becomes the controlling index. As a result, records are viewed in logical index order and no longer in physical order.

Info

See also: DbCreateIndex(), DbOrderInfo(), INDEX, OrdCondSet(), OrdListAdd(), OrdListClear(), Set()
Category: Database functions, Index functions
Source: rdd\dbcmd.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example creates two indexes in one index file.

   REQUEST Dbfcdx

   PROCEDURE Main

      USE Customer ALIAS Cust NEW VIA "DBFCDX"

      OrdCreate( "Customer", "ID"   , "CUSTNO"   )
      OrdCreate( "Customer", "NAME", "Upper(LastName+FirstName)" )
      OrdListClear()

      OrdListAdd( "Customer" )

      ? OrdNumber()                // result: 1
      ? OrdName()                  // result: ID
      ? OrdKey()                   // result: CUSTNO

      ? OrdSetfocus( "NAME" )      // result: ID
      ? OrdNumber()                // result: 2
      ? OrdName()                  // result: NAME
      ? OrdKey()                   // result: Upper(LastName+FirstName)

      CLOSE Cust

   RETURN
```

OrdCustom()

Determines if an index is a custom index.

Syntax

```
OrdCustom( [<cIndexName> | <nOrder>], ;
           [<cIndexFile>]          , ;
           [<lNewCustomFlag>]        ) --> lOldCustomFlag
```

Arguments

`<cIndexName>`

This is a character string holding the symbolic name of the index to query. It is analogous to the alias name of a work area. Support for *<cIndexName>* depends on the RDD used to create the index. Usually, RDDs that are able to maintain multiple indexes in one index file support symbolic index names, such as DBFCDX, for example.

`<nOrder>`

Optionally, a numeric value specifying the ordinal position of the index open in a work area. Indexes are numbered in the sequence of opening, beginning with 1. The value zero identifies the controlling index.

`<cIndexFile>`

<cIndexFile> is a character string with the name of the file that stores the index. It is only required when multiple index files are open that contain indexes having the same *<cIndexName>*.

`<lNewCustomFlag>`

A logical value can be passed for *<lNewCustomFlag>*. It changes the custom flag of the index. When .T. (true) is specified, the index becomes a custom index that is not automatically updated by the RDD. Passing .F. (false) removes the custom flag.

Return

The function returns the custom flag set before the function is called.

Description

The OrdCustom() function is used to determine if an index is a custom built index, or changes the custom flag of an index at runtime. Custom built indexes are not automatically updated by the RDD. Instead, index keys must be added to or removed from an index programmatically using OrdKeyAdd() or OrdKeyDel().

Info

See also:	DbOrderInfo(), INDEX, OrdCondSet(), OrdDescend(), OrdKeyAdd(), OrdKeyDel()
Category:	Database functions, Index functions, xHarbour extensions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

OrdDescend()

Determines the navigational order of a work area.

Syntax

```
OrdDescend( [<cIndexName> | <nOrder>], ;
            [<cIndexFile>]           , ;
            [<lNewDescendFlag>]        ) --> lOldDescendFlag
```

Arguments

<cIndexName>

This is a character string holding the symbolic name of the index to query. It is analogous to the alias name of a work area. Support for *<cIndexName>* depends on the RDD used to create the index. Usually, RDDs that are able to maintain multiple indexes in one index file support symbolic index names, such as DBFCDX, for example.

<nOrder>

Optionally, a numeric value specifying the ordinal position of the index open in a work area. Indexes are numbered in the sequence of opening, beginning with 1. The value zero identifies the controlling index.

<cIndexFile>

<cIndexFile> is a character string with the name of the file that stores the index. It is only required when multiple index files are open that contain indexes having the same *<cIndexName>*.

<lNewDescendFlag>

A logical value can be passed for *<lNewDescendFlag>*. It changes the descend flag of the index. When .T. (true) is specified, the navigational order in the current work area is set to descending, .F. (false) sets the navigational order to ascending.

Return

The function returns the descend flag set before the function is called.

Description

The function OrdDescend() is used to change the descend flag dynamically at runtime. The navigational order in a work area can be changed from ascending to descending and back without having to create a corresponding index. When .T. (true) is passed to the function, the navigational order is reversed, i.e. all functions and commands which move the record pointer are reversed.

Descending navigation

Function	Description
Bof()	Returns .T. when the last record is reached.
Eof()	Returns .T. when the first record is reached.
DbSkip(1)	Moves the record pointer to the previous record.
DbSkip(-1)	Moves the record pointer to the next record.
DbGoTop()	Moves the record pointer to the last record.
DbGoBottom()	Moves the record pointer to the first record.

Note: the function changes the descending flag at runtime only. It does not change the actual index file.

Info

See also: DbOrderInfo(), INDEX, OrdCondSet(), OrdCustom()
Category: Database functions, Index functions
Source: rdd\dbcmd.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example creates an ascending index and demonstrates
// the effect when the navigational order is reversed.

   PROCEDURE Main
      USE Customer
      INDEX ON Upper(Lastname+Firstname) TO Cust01

      GO TOP
      ? LastName                // result: Alberts
      GO BOTTOM
      ? LastName                // result: Waters

      OrdDescend( ,, .T. )      // change navigational order

      SKIP -1                   // skipping backwards from the last
                                // record hits begin of file.
      ? Bof()                   // result: .T.

      GO BOTTOM
      ? Eof()                   // result: .F.
      ? LastName                // result: Alberts

      SKIP
      ? Eof()                   // Result: .T.

      GO TOP
      ? LastName                // result: Waters

      USE
   RETURN
```

OrdDestroy()

Deletes an index from an index file.

Syntax

```
OrdDestroy( <cIndexName> [, <cIndexFile>] )  --> NIL
```

Return

The function always returns NIL.

Description

OrdDestroy() deletes an open index and removes it from an index file. If the index file contains only one index, the file is erased from disc. When the controlling index is removed, no index is active after OrdDestroy() returns and the database is navigated in physical order. Function OrdNumber() returns 0 in that case, although other indexes may be open. Use OrdSetFocus() to activate another index.

Info

See also: DELETE TAG, OrdCreate()
Category: Database functions, Index functions
Source: rdd\dbcmd.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example creates two indexes and destroys the controlling
// index.

   PROCEDURE Main
      USE Customer
      OrdCreate( "Customer1", "ID"   , "CUSTNO"   )
      OrdCreate( "Customer2", "NAME", "Upper(LastName+FirstName)" )

      OrdSetFocus( "ID" )

      ? OrdNumber()               // result: 1
      ? OrdName()                 // result: ID
      ? OrdKey()                  // result: CUSTNO

      ? OrdSetfocus( "NAME" )     // result: ID

      ? OrdName()                 // result: NAME
      ? OrdNumber()               // result: 2
      ? OrdKey()                  // result: Upper(LastName+FirstName)

      OrdDestroy( "NAME" )

      ? OrdNumber()               // result: 0
      ? OrdName()                 // result: (empty string)
      ? OrdKey()                  // result: (empty string)

      USE
   RETURN
```

OrdFindRec()

Searches a record number in the controlling index.

Syntax

```
OrdFindRec( <nRecno>, [<lCurrentRec>] ) --> lFound
```

Arguments

<nRecno>

This is a numeric value specifying the phyiscal record number to find in the controlling index. Physical record numbers are in the range of 1 to Lastrec().

<lCurrentRec>

This parameter defaults to .F. (false) causing OrdFindRec() to begin the search with the first record included in the controlling index. When .T. (true) is passed, the function begins the search with the current record.

Return

The function returns .T. (true) if a record is found in the controlling index that has the record number *<nRecno>*, otherwise .F. (false) is returned.

Description

OrdFindRec() searches a record number in the controlling index. It is similar to DbSeek(), which searches an index key. If OrdFindRec() finds the record number *<nRecno>* in the controlling index, it positions the record pointer on the found record, and Function Found() returns .T. (true) until the record pointer is moved again. In addition, both functions, BoF() and EoF() return .F. (false).

When a matching record number is not found, the record pointer is positioned on the "ghost record" (Lastrec()+1), and the function Found() returns .F. (false), while Eof() returns .T. (true).

Info

See also:	DbOrderInfo(), DbSeek(), OrdKeyGoto(), OrdKeyVal(), OrdWildSeek()
Category:	Database functions, Index functions, xHarbour extensions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example navigates the record pointer via the controlling index
// using physical and logical record numbers,

   REQUEST Dbfcdx

   PROCEDURE Main
      USE Customer VIA "DBFCDX"
      INDEX ON Upper(LastName) TAG NAME TO Cust01

      OrdFindRec( 1 )                    // Physical first record
      ? Lastname, Recno(), OrdKeyNo()    // result: Miller    1  15

      OrdFindRec( LastRec() )            // Physical last record
      ? Lastname, Recno(), OrdKeyNo()    // result: Smith    22  19

      OrdKeyGoto( 1 )                    // Logical first record
      ? Lastname, Recno(), OrdKeyNo()    // result: Alberts  20   1
```

```
   OrdKeyGoto( OrdKeyCount() )        // Logical last record
   ? Lastname, Recno(), OrdKeyNo()    // result: Waters   15  22

   OrdFindRec( -1 )                   // Non-existent record
   ? Lastname, Recno(), OrdKeyNo()    // result:          23   0

   ? Bof(), Eof(), Found()            // result: .T.  .T.  .F.
   USE
RETURN
```

OrdFor()

Retrieves the FOR expression of an index.

Syntax

```
OrdFor( [<nOrder>|<cIndexName>][,<cIndexFile>] ) --> cForExpression
```

Arguments

<nOrder>

A numeric value specifying the ordinal position of the index open in a work area. Indexes are numbered in the sequence of opening, beginning with 1. The value zero identifies the controlling index.

<cIndexName>

Alternatively, a character string holding the symbolic name of the open index can be passed. It is analogous to the alias name of a work area. Support for *<cIndexName>* depends on the RDD used to open the index. Usually, RDDs that are able to maintain multiple indexes in one index file support symbolic index names, such as DBFCDX, for example.

<cIndexFile>

<cIndexFile> is a character string with the name of the file that stores the index. It is only required when multiple index files are open that contain indexes having the same *<cIndexName>*.

Return

The function returns a character string holding the FOR condition of the specified index. If the index is created without FOR condition, or if no index is open an empty string ("") is returned. If no parameter is passed to OrdFor(), the function returns the FOR condition of the controlling index.

Description

OrdFor() returns the FOR expression of an index as a character string. It is similar to OrdKey() which returns the index key expression.

Info

See also:	DbOrderInfo(), INDEX, OrdCondSet(), OrdCreate(), OrdKey(), OrdName(), OrdNumber()
Category:	Database functions, Index functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

OrdIsUnique()

Retrieves the UNIQUE flag of an index.

Syntax

```
OrdIsUnique( [<nOrder>|<cIndexName>][,<cIndexFile>] ) --> lUniqueFlag
```

Arguments

`<nOrder>`

A numeric value specifying the ordinal position of the index open in a work area. Indexes are numbered in the sequence of opening, beginning with 1. The value zero identifies the controlling index.

`<cIndexName>`

Alternatively, a character string holding the symbolic name of the open index can be passed. It is analogous to the alias name of a work area. Support for *<cIndexName>* depends on the RDD used to open the index. Usually, RDDs that are able to maintain multiple indexes in one index file support symbolic index names, such as DBFCDX, for example.

`<cIndexFile>`

<cIndexFile> is a character string with the name of the file that stores the index. It is only required when multiple index files are open that contain indexes having the same *<cIndexName>*.

Return

The function returns the UNIQUE flag of the specified index as a logical value. If no index is open, the function returns .F. (false). If no database is open, a runtime error is created.

Description

OrdIsUnique() queries the UNIQUE flag of an index. This flag is stored in the index file header and is set upon index creation with the INDEX command.

Info

See also:	DbOrderInfo(), INDEX, OrdCondSet(), OrdCreate(), OrdFor(), OrdKey(), OrdName()
Category:	Database functions, Index functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

OrdKey()

Returns the key expression of an index.

Syntax

```
OrdKey( [<nOrder>|<cIndexName>][,<cIndexFile>] ) --> cIndexKey
```

Arguments

<nOrder>

A numeric value specifying the ordinal position of the index open in a work area. Indexes are numbered in the sequence of opening, beginning with 1. The value zero identifies the controlling index.

<cIndexName>

Alternatively, a character string holding the symbolic name of the open index can be passed. It is analogous to the alias name of a work area. Support for *<cIndexName>* depends on the RDD used to open the index. Usually, RDDs that are able to maintain multiple indexes in one index file support symbolic index names, such as DBFCDX, for example.

<cIndexFile>

<cIndexFile> is a character string with the name of the file that stores the index. It is only required when multiple index files are open that contain indexes having the same *<cIndexName>*.

Return

The function returns the key expression of the specified index as a character string. If no parameters are passed, the index key of the controlling index is returned. If no index is open, the return value is an empty string ("").

Description

OrdKey() returns the index key expression of an index as a character string. It is similar to OrdFor() which returns the FOR expression of the index.

Info

See also:	DbOrderInfo(), INDEX, IndexKey(), OrdCreate(), OrdFor(), OrdKeyCount(), OrdKeyNo(), OrdKeyVal(), OrdName(), OrdNumber()
Category:	Database functions, Index functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example creates two indexes and displays their key expressions

   PROCEDURE Main
      USE Customer
      OrdCreate( "Customer1", "ID"   , "CUSTNO"   )
      OrdCreate( "Customer2", "NAME", "Upper(LastName+FirstName)" )

      OrdSetFocus( "ID" )

      ? OrdKey()                  // result: CUSTNO

      OrdSetfocus( "NAME" )
```

```
    ? OrdName()                  // result: NAME
    ? OrdNumber()                // result: 2
    ? OrdKey()                   // result: Upper(LastName+FirstName)

    ? OrdKey(1)                  // result: CUSTNO
    ? OrdKey( "NAME" )           // result: Upper(LastName+FirstName)

    USE
 RETURN
```

OrdKeyAdd()

Adds an index key to a custom built index.

Syntax

```
OrdKeyAdd( [<nOrder>|<cIndexName>], ;
           [<cIndexFile>]          , ;
           [<xIndexValue>]           ) --> lSuccess
```

Arguments

<nOrder>

A numeric value specifying the ordinal position of the custom index open in a work area. Indexes are numbered in the sequence of opening, beginning with 1. The value zero identifies the controlling index.

<cIndexName>

Alternatively, a character string holding the symbolic name of the open custom index can be passed. It is analogous to the alias name of a work area. Support for *<cIndexName>* depends on the RDD used to open the index. Usually, RDDs that are able to maintain multiple indexes in one index file support symbolic index names, such as DBFCDX, for example.

<cIndexFile>

<cIndexFile> is a character string with the name of the file that stores the custom index. It is only required when multiple index files are open that contain indexes having the same *<cIndexName>*.

<xIndexValue>

This is the index value to be added to the index for the current record. It must be of the same data type as the value returned by the index expression. If omitted, *<xIndexValue>* is obtained by evaluating the index expression with the data of the current record.

Return

The function returns .T. (true) if the current record is successfully included in the index, otherwise .F. (false) is returned.

Description

OrdKeyAdd() is used to build a custom index whose entries are programmatically added and deleted. Custom built indexes are not automatically updated by the RDD but are initially empty. OrdKeyAdd() adds the current record to the custom index and OrdKeyDel() removes it. It is possible to add multiple index values to the index for the same record, so that the same record is found when different search values are passed to DbSeek().

If no parameters are passed, OrdKeyAdd() evaluates the index expresssion with the data of the current record to obtain *<xIndexValue>*. The record is added to the index when it matches the FOR condition and scoping restrictions, if they are defined. When *<xIndexValue>* is specified, it must have the same data type as the value of the expression &(OrdKey()).

OrdKeyAdd() fails if the record pointer is positioned on Eof(), if the specified index is not a custom index, or if the specified index does not exist.

This is important when the custom index is the controlling index. Since a custome index is initially empty, relative database navigation with SKIP positions the record pointer always at Eof(). To include records to a controlling custom index, they must be physically navigated to using DbGoto().

The recommended way of creating a custom index is to use a non-custom index as the controlling index, skip through the database and specify <*cIndexName*> for OrdListAdd() when the current record meets the conditions for being included in the custom index.

Info

See also: DbOrderInfo(), DbGoto(), INDEX, OrdKey(), OrdKeyDel(), OrdKeyVal()
Category: Database functions, Index functions
Source: rdd\dbcmd.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example shows two approaches for building a custom index.
// In the first approach, the controlling index is a regular index
// which allows for relative database navigation. The second approach
// uses the custome index as controlling index and requires absolute
// database navigation. The first five logical records and physical records
// are added to the custome index.

   REQUEST Dbfcdx

   PROCEDURE Main
      USE Customer VIA "DBFCDX"
      INDEX ON Upper(LastName+FirstName) TAG NAME    TO Cust01
      INDEX ON Upper(LastName+FirstName) TAG NAMESET TO Cust01t CUSTOM

      // relative navigation with non-custom index
      OrdSetFocus( "NAME" )
      GO TOP
      FOR i:=1 TO 5
         OrdKeyAdd( "NAMESET" )
         SKIP
      NEXT

      GO TOP
      Browse()

      // absolute navigation with custom index
      OrdSetFocus( "NAMESET" )
      FOR i:=1 TO 5
         DbGoto( i )
         OrdKeyAdd( "NAMESET" )
      NEXT

      GO TOP
      Browse()

      USE
   RETURN
```

OrdKeyCount()

Retrieves the total number of keys included in an index.

Syntax

```
OrdKeyCount( [<nOrder>|<cIndexName>][,<cIndexFile>] ) --> nKeyCount
```

Arguments

`<nOrder>`

A numeric value specifying the ordinal position of the index open in a work area. Indexes are numbered in the sequence of opening, beginning with 1. The value zero identifies the controlling index.

`<cIndexName>`

Alternatively, a character string holding the symbolic name of the open index can be passed. It is analogous to the alias name of a work area. Support for *<cIndexName>* depends on the RDD used to open the index. Usually, RDDs that are able to maintain multiple indexes in one index file support symbolic index names, such as DBFCDX, for example.

`<cIndexFile>`

<cIndexFile> is a character string with the name of the file that stores the index. It is only required when multiple index files are open that contain indexes having the same *<cIndexName>*.

Return

The function returns the number of keys included in the specified index. If no parameter is passed, the controlling index is used. If the specified index does not exist, a runtime error is generated.

Description

The function counts the index keys present in the specified index. For this, it recognizes a possible FOR condition and/or scope set with OrdScope(). If no FOR condition and no scope is defined for the index, the return value of OrdKeyCount() is identical with LastRec(), which returns the physical number of records in a work area.

Scopes and FOR conditions restrict the logical visibility of records, so that OrdKeyCount() represents the number of records that can be navigated to using SKIP. This information is especially required for programming scroll bars in browse views.

Info

See also:	DbOrderInfo(), DbGoto(), DbSkip(), LastRec(), OrdKeyGoto(), OrdKeyNo(), OrdScope()
Category:	Database functions, Index functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example compares return values of LastRec() and OrdKeyCount()
// when the logical visibility of records is unrestricted vs. restricted
// with OrdScope().

   #include "Ord.ch"

   PROCEDURE Main
      USE Customer
```

```
    INDEX ON Upper(LastName+Firstname)  TAG Name TO Cust01

    ? LastRec()                       // result: 225
    ? OrdKeyCount()                   // result: 225

    OrdScope( TOPSCOPE   , "E" )
    OrdScope( BOTTOMSCOPE, "G" )

    ? LastRec()                       // result: 225
    ? OrdKeyCount()                   // result:  13

    USE
 RETURN
```

OrdKeyDel()

Removes an index key from a custom built index.

Syntax

```
OrdKeyDel( [<nOrder>|<cIndexName>], ;
           [<cIndexFile>]          , ;
           [<xIndexValue>]           ) --> lSuccess
```

Arguments

`<nOrder>`

A numeric value specifying the ordinal position of the custom index open in a work area. Indexes are numbered in the sequence of opening, beginning with 1. The value zero identifies the controlling index.

`<cIndexName>`

Alternatively, a character string holding the symbolic name of the open custom index can be passed. It is analogous to the alias name of a work area. Support for *<cIndexName>* depends on the RDD used to open the index. Usually, RDDs that are able to maintain multiple indexes in one index file support symbolic index names, such as DBFCDX, for example.

`<cIndexFile>`

<cIndexFile> is a character string with the name of the file that stores the custom index. It is only required when multiple index files are open that contain indexes having the same *<cIndexName>*.

`<xIndexValue>`

This is the index value to be added to the index for the current record. It must be of the same data type as the value returned by the index expression. If omitted, *<xIndexValue>* is obtained by evaluating the index expression with the data of the current record.

Return

The function returns .T. (true) if the current record is successfully removed from the index, otherwise .F. (false) is returned.

Description

OrdKeyDel() is used in conjunction with OrdKeyAdd() to maintain a custom built index programmatically. Custom built indexes are not automatically updated by the RDD but are initially empty. OrdKeyDel() removes the current record from a custom index, if it is included in the specified index.

If no parameters are passed, OrdKeyDel() evaluates the index expresssion with the data of the current record to obtain *<xIndexValue>*. The record is removed from the index when it is found.

OrdKeyDel() fails if the record pointer is positioned on Eof(), if the specified index is not a custom index, if the specified index does not exist, or if the current record is not included in the custom index.

Info

See also: DbOrderInfo(), DbGoto(), INDEX, OrdKey(), OrdKeyAdd(), OrdKeyVal()
Category: Database functions, Index functions
Source: rdd\dbcmd.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example creates a regular and a custom index holding five
// index keys. The last index key is removed from the custom index.

   REQUEST Dbfcdx

   PROCEDURE Main
      USE Customer VIA "DBFCDX"
      INDEX ON Upper(LastName) TAG NAME    TO Cust01
      INDEX ON Upper(LastName) TAG NAMESET TO Cust01 CUSTOM

      OrdSetFocus( "NAME" )
      GO BOTTOM
      ? OrdKeyVal()                    // result: WATERS

      GO TOP
      FOR i:=1 TO 5
         OrdKeyAdd( "NAMESET" )
         SKIP
      NEXT

      OrdSetFocus( "NAMESET" )
      GO BOTTOM
      ? OrdKeyVal()                    // result: CHAUCER

      ? OrdKeyDel( "NAMESET" )
      ? OrdKeyVal()                    // result: NIL

      USE
   RETURN
```

OrdKeyGoTo()

Moves the record pointer to a logical record number.

Syntax

```
OrdKeyGoTo( <nIndexKeyNo> ) --> lSuccess
```

Arguments

<nIndexKeyNo>

This is a numeric value specifying the logical record number to move the record pointer to. Logical record numbers are in the range of 1 to OrdKeyCount() and depend on the controlling index.

Return

The function returns .T. (true) if the record pointer is successfully positioned on the specified logical record number, otherwise .F. (false) is returned.

Description

The function OrdKeyGoTo() moves the record pointer based on its logical record number, which can be retrieved using function OrdKeyNo().

The logical record number represents the position of a record in the controlling index, while the physical record number Recno() represents a record's position in the database. Physical record movement is accomplished using function DbGoTo().

Info

See also:	DbOrderInfo(), DbGoto(), LastRec(), OrdFindRec(), OrdKeyNo(), OrdKeyRelPos(), OrdKeyCount(), RecNo()
Category:	Database functions, Index functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example outlines the difference between physical and logical
// record pointer movement.

   REQUEST Dbfcdx

   PROCEDURE Main
      USE Customer VIA "DBFCDX"
      INDEX ON Upper(LastName) TAG NAME TO Cust01

      DbGoto( 1 )                        // Physical first record
      ? Lastname, Recno(), OrdKeyNo()    // result: Miller    1  15

      DbGoto( LastRec() )                // Physical last record
      ? Lastname, Recno(), OrdKeyNo()    // result: Smith    22  19

      OrdKeyGoto( 1 )                    // Logical first record
      ? Lastname, Recno(), OrdKeyNo()    // result: Alberts  20   1

      OrdKeyGoto( OrdKeyCount() )        // Logical last record
      ? Lastname, Recno(), OrdKeyNo()    // result: Waters   15  22
```

```
   USE
RETURN
```

OrdKeyNo()

Returns the logical record number of the current record.

Syntax

```
OrdKeyNo( [<nOrder>|<cIndexName>], [<cIndexFile>] ) --> nOrdKeyNo
```

Arguments

<cIndexName>

This is a character string holding the symbolic name of the index to query. It is analogous to the alias name of a work area. Support for *<cIndexName>* depends on the RDD used to create the index. Usually, RDDs that are able to maintain multiple indexes in one index file support symbolic index names, such as DBFCDX, for example.

<nOrder>

Optionally, a numeric value specifying the ordinal position of the index open in a work area. Indexes are numbered in the sequence of opening, beginning with 1. The value zero identifies the controlling index.

<cIndexFile>

<cIndexFile> is a character string with the name of the file that stores the index. It is only required when multiple index files are open that contain indexes having the same *<cIndexName>*.

Return

The function returns the logical record number of the current record as a numeric value. Logical record numbers are in the range of 1 to OrdKeyCount() and depend on the controlling index. If the current record is not included in the index, the return value is zero.

Description

OrdKeyNo() is used to determine the logical record number of the current record. The logical record number represents the position of a record in the controlling index, while the physical record number Recno() represents a record's position in the database. Physical record movement is accomplished using function DbGoTo(), while logical record pointer movement is done with OrdKeyGoTo().

Info

See also:	DbOrderInfo(), INDEX, OrdKey(), OrdKeyCount(), OrdKeyGoTo(), OrdKeyRelPos(), OrdListAdd(), OrdNumber(), OrdScope(), Recno()
Category:	Database functions, Index functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example outlines the difference between physical and logical
// record numbers in an indexed database without and with scope.

   #include "Ord.ch"

   PROCEDURE Main
      USE Customer
      INDEX ON Upper(LastName+Firstname)  TAG Name TO Cust01

      ? LastRec()                         // result: 225
```

```
    ? OrdKeyCount()                    // result: 225

    GO TOP
    ? Recno()                          // result:  15
    ? OrdKeyNo()                       // result:   1

    GO BOTTOM
    ? Recno()                          // result: 123
    ? OrdKeyNo()                       // result: 225

    OrdScope( TOPSCOPE   , "E" )
    OrdScope( BOTTOMSCOPE, "G" )

    ? LastRec()                        // result: 225
    ? OrdKeyCount()                    // result:  13

    GO TOP
    ? Recno()                          // result:  87
    ? OrdKeyNo()                       // result:   1

    GO BOTTOM
    ? Recno()                          // result: 207
    ? OrdKeyNo()                       // result:  13

    USE
 RETURN
```

OrdKeyRelPos()

Returns or sets the relative position of the current record.

Syntax

```
OrdKeyRelPos( [<nNewPosition>] ) --> nRelativePosition
```

Arguments

<nNewPosition>

Optionally, a numeric value between 0 and 1 can be passed. It specifies the relative position to move the record pointer to.

Return

The function returns a value between 0.0 and 1.0 (inclusive), which is the relative position of the current record.

Description

The function is used to obtain the relative position of the record pointer, or to move it to a relative position. The relative position is specified as a value between 0 and 1. The return value multiplied with 100 is the relative position of the record pointer in percent of all records. It can be used to visualize the record pointer position with a scroll bar.

OrdKeyRelPos() can be used with both, indexed and non-index databases. If a controlling index is active, the function returns the relative position of the current record within the controlling index. If a scope is set for the controlling index using OrdScope(), the number of visible records is taken into account. Filter conditions that limit the logical visibility of records are ignored.

Note that the relative psoition returned by OrdKeyRelPos() is only an estimate whose accuracy grows with the number of records.

Info

See also:	DbOrderInfo(), OrdKeyCount(), OrdKeyGoTo(), OrdScope(), Recno()
Category:	Database functions, Index functions, xHarbour extensions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates the relative position of the first/last
// physical/logical record in an indexed database.

   PROCEDURE Main
      USE Customer

      INDEX ON Upper(LastName)
      SET DECIMALS TO 4

      GO TOP                         // Logical first record
      ? Lastname, OrdKeyRelPos()     // result: Alberts   0.0227

      DbGoto( 1 )                    // Physical first record
      ? Lastname, OrdKeyRelPos()     // result: Miller   0.6591

      GO BOTTOM                      // Logical last record
      ? Lastname, OrdKeyRelPos()     // result: Waters   0.9773
```

```
      DbGoto( LastRec() )           // Physical last record
      ? Lastname, OrdKeyRelPos()    // result: Smith    0.8864

   RETURN
```

OrdKeyVal()

Retrieves the index value of the current record from the controlling index.

Syntax

```
OrdKeyVal() --> xIndexValue
```

Return

The function returns the index value of the current record. The return value is NIL if the current record is not included in the controlling index or if there is no controlling index.

Description

The function retrieves the index value for the current record from the index file holding the controlling index, not from the database file. This can be advantageous when an index expression is comprised of multiple fields. Their values can be assigned to a memory variable with one function call, rather than accessing multiple field variables individually.

Info

See also:	DbOrderInfo(), INDEX, IndexKey(), OrdCreate(), OrdFor(), OrdKey(), OrdKeyNo(), OrdName(), OrdNumber()
Category:	Database functions, Index functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example uses the index value to display Lastname and Firstname
// instead of field variables.

   PROCEDURE Main
      LOCAL cName

      USE Customer
      INDEX ON Upper( LastName + FirstName ) TO Cust01

      ? FieldPos( "FirstName" )            // result: 2
      ? FieldPos( "MiddleName" )           // result: 3
      ? FieldPos( "LastName" )             // result: 4

      ? FieldLen( 2 )                      // result: 20
      ? FieldLen( 4 )                      // result: 20

      cName := OrdKeyVal()
      ? Trim( Right(cName, 20)), Trim( Left(cName,20) )

      USE
   RETURN
```

OrdListAdd()

Opens and optionally activates a new index in a work area.

Syntax

```
OrdListAdd( <cIndexFile> ) --> NIL
```

Arguments

<cIndexFile>

<cIndexFile> is a character string with the name of the file containing the index(es) to add to the list of open indexes in a work area. The file name can be specified including path information and extension. When the file extension is omitted, it is determined by the database driver that opened the database file in the work area.

Return

The function always returns NIL.

Description

OrdListAdd() opens an index file in a used work area and adds all contained indexes to the list of open indexes. If *<cIndexFile>* is the first index file opened in the work area, the first index stored in the file becomes the controlling index and the record pointer is moved to the first logical record. If there is a controlling index already active when OrdListAdd() is called, both, record pointer and controlling index, remain unchanged.

Info

See also:	DbClearIndex(), DbCreateIndex(), DbReindex(), DbSetOrder(), OrdCreate(), OrdListClear(), OrdListRebuild(), SET INDEX
Category:	Database functions, Index functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows the effect of OrdListAdd() when an index is opened
// a first index in a work area vs. the situation when an additional index
// is opened.

   PROCEDURE Main
      USE Customer
      INDEX ON Upper( LastName  ) TO Cust01
      INDEX ON Upper( FirstName ) TO Cust02

      // reopening the database closes all indexes
      USE Customer
      ? Recno(), LastName              // result:  1  Miller

      OrdListAdd( "Cust01" )
      ? Recno(), LastName              // result: 20  Alberts

      OrdListAdd( "Cust02" )
      ? Recno(), LastName              // result: 20  Alberts

      USE
   RETURN
```

OrdListClear()

Closes all indexes open in a work area.

Syntax

```
OrdListClear() --> NIL
```

Return

The function always returns NIL.

Description

OrdListClear() closes all indexes open in a work area. When indexes are closed, database records can only be navigated in their physical order, since a logical order is no longer present.

Info

See also: DbClearIndex(), OrdListAdd(), OrdListRebuild()
Category: Database functions, Index functions
Source: rdd\dbcmd.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example creates two indexes and demonstrates the effect of
// closing the indexes with OrdListClear()

   PROCEDURE Main
      USE Customer
      INDEX ON Upper( FirstName ) TO Cust01
      INDEX ON Upper( LastName  ) TO Cust02

      USE Customer
      ? Recno(), LastName, Firstname   // result:  1  Miller   Irene

      OrdListAdd( "Cust01" )

      ? OrdCount()                     // result:  1
      ? Recno(), LastName, Firstname   // result:  9  Feldman  Allen

      OrdListAdd( "Cust02" )
      OrdSetFocus( "Cust02" )
      DbGoTop()

      ? OrdCount()                     // result:  2
      ? Recno(), LastName, Firstname   // result: 20  Alberts  Cathy

      OrdListClear()
      ? OrdCount()                     // result:  0
      ? Recno(), LastName, Firstname   // result: 20  Alberts  Cathy

      USE
   RETURN
```

OrdListRebuild()

Rebuilds all indexes open in the current work area

Syntax

```
OrdListRebuild() --> NIL
```

Return

The function always returns NIL.

Description

OrdListRebuild() is used to reorganize all indexes open in the current work area. Use function OrdCreate() to reorganize a single index.

Info

See also:	DbClearIndex(), DbCreateIndex(), DbSetIndex(), DbSetOrder(), INDEX, OrdCreate(), OrdListAdd(), OrdListClear(), REINDEX
Category:	Database functions, Index functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example opens a database along with two index files. All indexes
// stored in the index files are rebuild.

   PROCEDURE Main
      USE Customer
      SET INDEX TO Cust01, Cust02

      OrdListRebuild()

      USE
   RETURN
```

OrdName()

Returns the symbolic name of an open index by its ordinal position.

Syntax

```
OrdName( [<nOrder>] [,<cIndexFile>] ) --> cIndexName
```

Arguments

<nOrder>

A numeric value specifying the ordinal position of the index open in a work area. Indexes are numbered in the sequence of opening, beginning with 1. The value zero identifies the controlling index. The default value is 0.

<cIndexFile>

<cIndexFile> is a character string with the name of the file that stores the index. It is only required when multiple index files are open that contain indexes having the same name.

Return

The function returns the index name at position *<nOrder>* in the list of open indexes as a character string. If no parameters are passed, the index name of the controlling index is returned. If no index is open, the return value is an empty string ("").

Description

OrdName() returns the symbolic name of an index as a character string. The name is specified with the TAG option of the INDEX command when the index is created. The reverse function of OrdName() is OrdNumber() which returns the ordinal position of an open index by its symbolic name.

Info

See also:	Alias(), INDEX, OrdFor(), OrdKey(), OrdNumber()
Category:	Database functions, Index functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example creates two indexes in one index file and displays.
// various information about the open indexes.

   REQUEST Dbfcdx

   PROCEDURE Main
      USE Customer VIA "DBFCDX"

      INDEX ON Upper(LastName+FirstName) TAG NAME  TO Cust01
      INDEX ON CustID                    TAG ID    TO Cust01

      ? OrdName()                 // result: ID
      ? OrdKey()                  // result: CustID

      ? OrdSetfocus( "NAME" )     // result: ID
      ? OrdName()                 // result: NAME
      ? OrdKey()                  // result: Upper(LastName+FirstName)

      ? OrdName(1)                // result: NAME
      ? OrdName(2)                // result: ID
```

```
   USE
RETURN
```

OrdNumber()

Returns the ordinal position of an open index by its symbolic name.

Syntax

```
OrdNumber( [<cIndexName>][, <cIndexFile>] ) --> nOrder
```

Arguments

<cIndexName>

This is a character string holding the symbolic name of the index whose ordinal position is searched. It is analogous to the alias name of a work area. Support for *<cIndexName>* depends on the RDD used to create the index. Usually, RDDs that are able to maintain multiple indexes in one index file support symbolic index names, such as DBFCDX, for example.

<cIndexFile>

<cIndexFile> is a character string with the name of the file that stores the index. It is only required when multiple index files are open that contain indexes having the same symbolic name.

Return

The function returns a numeric value specifying the ordinal position of the index *<cIndexName>*. Indexes are numbered in the sequence of opening, beginning with 1. If no parameter is passed, the ordinal position of the controlling index is returned. If no index is open, or if the name *<cIndexName>* does not exist, the return value is zero.

Description

OrdNumber() returns the ordinal position of an index based on its symbolic name. The name is specified with the TAG option of the INDEX command when the index is created. The reverse function of OrdNumber() is OrdName() which returns the symbolic name of an open index by its ordinal position.

Info

See also:	Alias(), INDEX, OrdFor(), OrdKey(), OrdName()
Category:	Database functions, Index functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example creates two indexes in one index file and displays.
// various information about the indexes.

   REQUEST Dbfcdx

   PROCEDURE Main
      USE Customer VIA "DBFCDX"
      ? OrdNumber()                // result: 0

      INDEX ON Upper(LastName+FirstName) TAG NAME  TO Cust01
      INDEX ON CustID                    TAG ID    TO Cust01

      ? OrdNumber()                // result: 2
      ? OrdKey()                   // result: CustID
```

```
   ? OrdNumber( "NAME" )        // result: 1
   ? OrdKey( "NAME" )           // result: Upper(LastName+FirstName)

   USE
RETURN
```

OrdScope()

Defines the top and/or bottom scope for navigating in the controlling index.

Syntax

```
OrdScope( <nScope>, [<xNewValue>] ) --> xOldValue
```

Arguments

`<nScope>`

The parameter is a numeric value identifying the top or bottom scope to set or clear. #define constants are available in Ord.ch that can be used for *<nScope>*.

#define constants for *<nScope>*

Constant	Value	Description
TOPSCOPE	0	Specifies the top scope
BOTTOMSCOPE	1	Specifies the bottom scope

`<xNewValue>`

The parameter is specifies the value to be used as top or bottom boundary for the scope of the controlling index. The value must be of the same data type as the the value of the index expression, or it can be a code block that returns a value of this data type.

If *<xNewValue>* is omitted or the value NIL is passed explicitely for *<xNewValue>*, a previously defined top or bottom scope is cleared.

Return

The function returns the value for the top or bottom scope that is defined before the function is called.

Description

The OrdScope() function defines or clears the top and bottom scope for indexed database navigation. The top and bottom scope values define the range of values valid for navigating the record pointer within the controlling index.

Top and bottom scope value are inclusive, i.e. they are part of the resulting subset of records. They can be defined as constant values or as code blocks. When code blocks are used for top and/or bottom scope, they must return a value having the data type of the index expression. The scope code blocks are evaluated during database navigation, so that scope boundaries can change dynamically while the record pointer is moved.

When a scope is set, the GO TOP command is equivalent to DbSeek(*<topScope>*), while GO BOTTOM is the same as DbSeek(*<bottomScope>*, .F., .T.).

Attempting to move the record pointer before the top scope value does not change the record pointer but causes function BoF() to return .T. (true).

When the record pointer is moved beyond the bottom scope value, it is advanced to the ghost record (Lastrec()+1) and function EoF() returns .T. (true).

Note: The current scope settings can be queried without altering them using function DbOrderInfo().

Info

See also: DbOrderInfo(), OrdCondSet(), OrdKey(), SET RELATION, SET SCOPE
Category: Database functions, Index functions
Header: Ord.ch
Source: rdd\dbcmd.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates evaluation of scope code blocks while a
// database is browsed. Although the code blocks return constant values,
// they could return the result of a function call as well.

   #include "Ord.ch"

   PROCEDURE Main
      USE Customer
      INDEX ON Upper(LastName+Firstname) TO Cust01

      OrdScope( TOPSCOPE   , {|| Tone(1000), "E" } )

      OrdScope( BOTTOMSCOPE, {|| Tone(500), "K" } )

      Browse()
   RETURN
```

OrdSetFocus()

Sets focus to the controlling index in a work area

Syntax

```
OrdSetFocus( [<nOrder>|<cIndexName>], ;
             [<cIndexFile>]           ) --> cOldIndexName
```

Arguments

`<nOrder>`

A numeric value specifying the ordinal position of the index open in a work area to select as the controling index. Indexes are numbered in the sequence of opening, beginning with 1. The value zero identifies the controlling index.

`<cIndexName>`

Alternatively, a character string holding the symbolic name of the index to select can be passed. It is analogous to the alias name of a work area. Support for *<cIndexName>* depends on the RDD used to open the index. Usually, RDDs that are able to maintain multiple indexes in one index file support symbolic index names, such as DBFCDX, for example.

`<cIndexFile>`

<cIndexFile> is a character string with the name of the file that stores the index. It is only required when multiple index files are open that contain indexes having the same *<cIndexName>*.

Return

The function returns the symbolic name of the controlling index that is selected before the function is called. If no index is open, the return value is an empty string ("").

Description

The OrdSetFocus() function selects an index open in a work area as the "controlling" index. Many indexes can be open in a work area, but only one index can have focus. This index determines the logical order how records stored in a database can be accessed and/or are visible in the current work area.

If an open index is specified with *<nOrder>* or *<cIndexName>*, this index is becomes the controlling index and the returnvalue is the name of the previous controlling index.

Calling OrdSetFocus() without a parameter yields the current controlling index.

Info

See also:	Alias(), DbSetOrder(), INDEX, OrdFor(), OrdKey(), OrdName(), OrdNumber(), SET INDEX
Category:	Database functions, Index functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example creates two indees in one index file and displays
// the result of OrdSetFocus(). The contents of the first logical
// records resulting from both indexes illustrates the index change.

   REQUEST DbfCdx
```

```
PROCEDURE Main
   USE Customer VIA "DbfCdx"
   INDEX ON Upper( LastName  ) TAG Last  TO Customer
   INDEX ON Upper( FirstName ) TAG First TO Customer

   ? OrdSetFocus()             // result: FIRST

   DbGotop()
   ? Lastname, Firstname       // result: Feldman   Allen

   ? OrdSetFocus(1)            // result: FIRST

   DbGotop()
   ? Lastname, Firstname       // result: Alberts   Cathy

   ? OrdSetFocus( "First" )    // result: LAST

   DbGotop()
   ? Lastname, Firstname       // result: Feldman   Allen

   USE
RETURN
```

OrdSetRelation()

Defines a scoped relation between child work area and parent work area.

Syntax

```
OrdSetRelation( <nArea> | <cAlias>, ;
                <bRelation>       , ;
               [<cRelation>]        )  --> NIL
```

Arguments

<nArea>

The numeric ordinal position of the child work area to relate to the current work area.

<cAlias>

Optionally, the child work area can be specified as a character string holding the symbolic alias name of the child work area.

<bRelation>

A code block containing the relation expression.

<cRelation>

The relation expression in form of a character string.

Return

The function always returns NIL.

Description

The function OrdSetRelation() creates a relation between the current work area (parent) and a secondary work area (Child) specified with *<nArea>* or *<cAlias>*. The relation is scoped in the child work area, i.e. when the child area is selected as the current work area, only those records matching the *<bRelation>* expression can be navigated to. All other records are not visible.

OrdSetRelation() is identical with DbSetRelation() when the fourth parameter *<lScoped>* of this function is set to .T. (true). Refer to DbSetRelation() for more information on creating links between parent and child work areas.

Info

See also:	DbSetRelation()
Category:	Database functions, Index functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

OrdSkipRaw()

Moves the record pointer via the controlling index.

Syntax

```
OrdSkipRaw( [<nRecords>] ) --> NIL
```

Arguments

<nRecords>

This is a numeric value indicating the number of records to move the record pointer within the controlling index. Positive values for *<nRecords>* move the record pointer forwards (towards the last logical record), negative values move it backwards. The default value is 1, i.e. calling OrdSkipRaw() with no parameter advances the record pointer to the next logical record.

Return

The return value is always NIL.

Description

OrdSkipRaw() moves the record pointer using information from the controlling index only, and ignores any filter condition set with SET DELETED or SET FILTER. All records included in the controlling index can be navigated to with OrdSkipRaw().

Use function DbSkip() when a filter condition should be recognized for record pointer movement.

Info

See also:	DbSkip(), DbOrderInfo(), OrdKeyCount(), OrdKeyGoTo(), OrdScope(), Recno()
Category:	Database functions, Index functions, xHarbour extensions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates movement of the record pointer using
// DbSkip() versus OrdSkipRaw(). Note that Bof() returns  .F.
// with OrdSkipRaw() although the filter condition is not met.

   PROCEDURE Main
      USE Customer
      INDEX ON Upper(LastName) TO Cust01

      ? Bof(), Lastname     // result:  .F.  Alberts

      SET FILTER TO Lastname >= "K"

      GO TOP
      ? Bof(), Lastname     // result:  .F.  Keller

      DbSkip( -1 )
      ? Bof(), Lastname     // result:  .T.  Keller

      OrdSkipRaw( -1 )
      ? Bof(), Lastname     // result:  .F.  Henreid

      USE
   RETURN
```

OrdSkipUnique()

Navigates to the next or previous record that has another index value.

Syntax

```
OrdSkipUnique( [<nDirection>] ) --> lSuccess
```

Arguments

`<nDirection>`

This parameter specifies the direction to move the record pointer. If <*nDirection*> is a negative numeric value, the record pointer is moved up (towards the begin of file). If <*nDirection*> is NIL, 0 or a positive numeric value, the record pointer is moved down (towards the end of file).

Return

The function returns .T. (true) when the record pointer is moved to the next/previous record having a different index value.

Description

OrdSkipUnique() navigates the record pointer in an indexed database as if the UNIQUE flag was set for the controlling index. If the controlling index is a regular index without the UNIQUE flag, it may contain identical index values for multiple records of the database. As a result, records that have duplicate index values can be navigated to with DbSkip().

If logical database navigation should ignore records having duplicate index values, function OrdSkipUnique() can be used. This function navigates the record pointer from the current record to the first/previous record whose index value differs from the current one.

When OrdSkipUnique() is used for record pointer movement, the visibility of records is the same as if an INDEX is created with the UNIQUE flag.

Info

See also:	DbOrderInfo(), INDEX, OrdCreate(), OrdDescend(), OrdIsUnique()
Category:	Database functions, Index functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example creates a database and fills it with data from
// the string "cNames", thus creating duplicate index keys. The
// database is scanned from the first to the last record, but
// the next record is navigated to using OrdSkipUnique(). As
// a result, only four records of 14 show up in the result array.

   PROCEDURE Main
      LOCAL i, aResult
      LOCAL cNames := "AADDDDDCCCBBBB"
      LOCAL aStruct := { {"NAME", "C", 10, 0 } }

      // create database and index
      DbCreate( "TEST.DBF", aStruct )
      USE Test EXCLUSIVE
      INDEX ON Name TO Test

      // fill database using string data
```

```
    FOR i:=1 TO Len( cNames )
       APPEND BLANK
       REPLACE FIELD->NAME WITH Replicate( cNames[i], 3 )
    NEXT

    // collect result of OrdSkipUnique() navigation
    // in array "aResult"
    GO TOP
    aResult := Array(0)

    DO WHILE .NOT. Eof()
       AAdd( aResult, Trim( FIELD->NAME ) )
       OrdSkipUnique()
    ENDDO

    // Length of input data
    ? Len( cNames )                      // result: 14
    ? LastRec()                          // result: 14

    // Length of output data
    ? Len( aResult )                     // result: 4

    AEVal( aResult, {|c| QOut(c) } )

    // output:
    // AAA
    // BBB
    // CCC
    // DDD

    USE
 RETURN
```

OrdWildSeek()

Searches a value in the controlling index using wild card characters.

Syntax

```
OrdWildSeek( <cWildCardString>, ;
            [<lCurrentRec>]   , ;
            [<lBackwards>]      ) --> lFound
```

Arguments

<cWildCardString>

This is a character string to search in the controlling index. It may include the wild card characters "?" and "*". The question mark matches a single character, while the asterisk matches one or more characters.

<lCurrentRec>

This parameter defaults to .F. (false) causing OrdWildSeek() to begin the search with the first record included in the controlling index. When .T. (true) is passed, the function begins the search with the current record.

<lBackwards>

If .T. (true) is passed, OrdWildSeek() searches *<cWildCardString>* towards the begin of file. The default value is .F. (false), i.e. the function searches towards the end of file.

Return

The function returns .T. (true) if a record matching *<cWildCardString>* is found in the controlling index, otherwise .F. (false) is returned.

Description

OrdWildSeek() searches a character string that may include wild card characters in the controlling index. This allows for collecting subsets of records based on an approximate search string. Records matching the search string are found in the controlling index, and the record pointer is positioned on the found record.

When a matching record is found, the function Found() returns .T. (true) until the record pointer is moved again. In addition, both functions, BoF() and EoF() return .F. (false).

If the searched value is not found, OrdWildSeek() positions the record pointer on the "ghost record" (Lastrec()+1), and the function Found() returns .F. (false), while Eof() returns .T. (true). The SET SOFTSEEK setting is ignored by OrdWildSeek().

Info

See also:	DbSeek(), LOCATE, OrdFindRec(), OrdKeyGoto(), WildMatch()
Category:	Database functions, Index functions, xHarbour extensions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example uses two wildcard search strings to show
// possible search results of OrdWildSeek()

   PROCEDURE Main
      LOCAL aCust := {}
```

```
    USE Customer
    INDEX ON Upper(LastName) TO Cust01

    DO WHILE OrdWildSeek( "*MAN?", .T. )
       AAdd( aCust, FIELD->Lastname )
    ENDDO

    AEval( aCust, {|c| QOut(c) } )
    // Found records:
    //   Dormann
    //   Feldman

    GO TOP
    aCust := {}
    DO WHILE OrdWildSeek( "*EL*", .T. )
       AAdd( aCust, FIELD->Lastname )
    ENDDO

    AEval( aCust, {|c| QOut(c) } )
    // Found records:
    //   Feldman
    //   Hellstrom
    //   Keller
    //   Reichel
    USE
 RETURN
```

Os()

Returns the name of the operating system.

Syntax

```
OS() --> cOsName
```

Return

The function returns a character string holding the operating system name.

Description

Os() is an informational function used to query the name of the operating system an xHarbour application is running on.

Info

See also: Getenv(), HB_BuildDate(), HB_BuildInfo(), Os_IsWinXP(), Version()
Category: Environment functions
Source: rtl\version.c
LIB: xhb.lib
DLL: xhbdll.dll

Os_IsWin2000()

Checks if the application is running on Windows 2000.

Syntax

```
Os_IsWin2000() --> lIsWin2000
```

Return

The function returns .T. (true) if the application runs on Windows 2000, otherwise .F. (false) is returned.

Description

Os_IsWin2000() belongs to a group of environment functions that test if an xHarbour application is running on a particular Windows operating system version.

Info

See also:	Os(), Os_IsWinNT(), Os_IsWin9X(), Os_IsWin95(), Os_IsWin98(), Os_IsWinME(), Os_IsWinNT351(), Os_IsWinNT4(), Os_IsWinXP(), Os_IsWin2003(), Os_IsWtsClient(), Os_VersionInfo()
Category:	Environment functions, xHarbour extensions
Source:	rtl\winos.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Os_IsWin2000_Or_Later()

Checks if the application is running on Windows version 2000 or later

Syntax

```
Os_IsWin2000_Or_Later() --> lIsWin2000_Or_Later
```

Return

The function returns .T. (true) if the application runs on Windows version 2000 or later, otherwise .F. (false) is returned.

Description

Os_IsWin2000_Or_Later() belongs to a group of environment functions that test if an xHarbour application is running on a particular Windows operating system version.

Info

See also:	Os(), Os_IsWinNT(), Os_IsWin9X(), Os_IsWin95(), Os_IsWin98(), Os_IsWinME(), Os_IsWinNT351(), Os_IsWinNT4(), Os_IsWinXP(), Os_IsWin2003(), Os_IsWtsClient(), Os_VersionInfo()
Category:	Environment functions, xHarbour extensions
Source:	rtl\winos.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Os_IsWin2003()

Checks if the application is running on Windows 2003.

Syntax

```
Os_IsWin2003() --> lIsWin2003
```

Return

The function returns .T. (true) if the application runs on Windows 2003, otherwise .F. (false) is returned.

Description

Os_IsWin2003() belongs to a group of environment functions that test if an xHarbour application is running on a particular Windows operating system version.

Info

See also:	Os(), Os_IsWinNT(), Os_IsWin9X(), Os_IsWin95(), Os_IsWin98(), Os_IsWinME(), Os_IsWinNT351(), Os_IsWinNT4(), Os_IsWinXP(), Os_IsWin2000(), Os_IsWtsClient(), Os_VersionInfo()
Category:	Environment functions, xHarbour extensions
Source:	rtl\winos.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Os_IsWin95()

Checks if the application is running on Windows 95.

Syntax

```
Os_IsWin95() --> lIsWin95
```

Return

The function returns .T. (true) if the application runs on Windows 95, otherwise .F. (false) is returned.

Description

Os_IsWin95() belongs to a group of environment functions that test if an xHarbour application is running on a particular Windows operating system version.

Info

See also:	Os(), Os_IsWinNT(), Os_IsWin9X(), Os_IsWin98(), Os_IsWinME(), Os_IsWinNT351(), Os_IsWinNT4(), Os_IsWin2000(), Os_IsWinXP(), Os_IsWin2003(), Os_IsWtsClient(), Os_VersionInfo()
Category:	Environment functions, xHarbour extensions
Source:	rtl\winos.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Os_IsWin98()

Checks if the application is running on Windows 98.

Syntax

```
Os_IsWin98() --> lIsWin98
```

Return

The function returns .T. (true) if the application runs on Windows 98, otherwise .F. (false) is returned.

Description

Os_IsWin98() belongs to a group of environment functions that test if an xHarbour application is running on a particular Windows operating system version.

Info

See also:	Os(), Os_IsWinNT(), Os_IsWin9X(), Os_IsWin95(), Os_IsWinME(), Os_IsWinNT351(), Os_IsWinNT4(), Os_IsWin2000(), Os_IsWinXP(), Os_IsWin2003(), Os_IsWtsClient(), Os_VersionInfo()
Category:	Environment functions, xHarbour extensions
Source:	rtl\winos.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Os_IsWin9X()

Checks if the application is running on a Windows 9x platform.

Syntax

```
Os_IsWin9X() --> lIsWin9X
```

Return

The function returns .T. (true) if the application runs on a Windows 9x platform, otherwise .F. (false) is returned. The 9x platform includes Windows 95, 98 and ME.

Description

Os_IsWin9x() belongs to a group of environment functions that test if an xHarbour application is running on a particular Windows platform.

Info

See also:	Os(), Os_IsWinNT(), Os_IsWin95(), Os_IsWin98(), Os_IsWinME(), Os_IsWinNT351(), Os_IsWinNT4(), Os_IsWin2000(), Os_IsWinXP(), Os_IsWin2003(), Os_IsWtsClient(), Os_VersionInfo()
Category:	Environment functions, xHarbour extensions
Source:	rtl\winos.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Os_IsWinME()

Checks if the application is running on Windows ME.

Syntax

```
Os_IsWinME() --> lIsWinME
```

Return

The function returns .T. (true) if the application runs on Windows ME, otherwise .F. (false) is returned.

Description

Os_IsWinME() belongs to a group of environment functions that test if an xHarbour application is running on a particular Windows operating system version.

Info

See also:	Os(), Os_IsWinNT(), Os_IsWin9X(), Os_IsWin95(), Os_IsWin98(), Os_IsWinNT351(), Os_IsWinNT4(), Os_IsWin2000(), Os_IsWinXP(), Os_IsWin2003(), Os_IsWtsClient(), Os_VersionInfo()
Category:	Environment functions, xHarbour extensions
Source:	rtl\winos.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Os_IsWinNT()

Checks if the application is running on a Windows NT platform.

Syntax

```
Os_IsWinNT() --> lIsWinNT
```

Return

The function returns .T. (true) if the application runs on a Windows NT platform, otherwise .F. (false) is returned. The NT platform includes Windows NT, 2000, XP and 2003 Server.

Description

Os_IsWin95() belongs to a group of environment functions that test if an xHarbour application is running on a particular Windows platform.

Info

See also:	Os(), Os_IsWin9x(), Os_IsWin95(), Os_IsWin98(), Os_IsWinME(), Os_IsWinNT351(), Os_IsWinNT4(), Os_IsWin2000(), Os_IsWinXP(), Os_IsWin2003(), Os_IsWtsClient(), Os_VersionInfo()
Category:	Environment functions, xHarbour extensions
Source:	rtl\winos.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Os_IsWinNT351()

Checks if the application is running on Windows NT 3.51.

Syntax

```
Os_IsWinNT351() --> lIsWinNT351
```

Return

The function returns .T. (true) if the application runs on Windows NT 3.51, otherwise .F. (false) is returned.

Description

Os_IsWinNT351() belongs to a group of environment functions that test if an xHarbour application is running on a particular Windows operating system version.

Info

See also:	Os(), Os_IsWinNT(), Os_IsWin9X(), Os_IsWin95(), Os_IsWin98(), Os_IsWinME(), Os_IsWinNT4(), Os_IsWin2000(), Os_IsWinXP(), Os_IsWin2003(), Os_IsWtsClient(), Os_VersionInfo()
Category:	Environment functions, xHarbour extensions
Source:	rtl\winos.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Os_IsWinNT4()

Checks if the application is running on Windows NT 4.0.

Syntax

```
Os_IsWinNT4() --> lIsWinNT40
```

Return

The function returns .T. (true) if the application runs on Windows NT 4.0, otherwise .F. (false) is returned.

Description

Os_IsWinNT4() belongs to a group of environment functions that test if an xHarbour application is running on a particular Windows operating system version.

Info

See also:	Os(), Os_IsWinNT(), Os_IsWin9X(), Os_IsWin95(), Os_IsWin98(), Os_IsWinME(), Os_IsWinNT351(), Os_IsWin2000(), Os_IsWinXP(), Os_IsWin2003(), Os_IsWtsClient(), Os_VersionInfo()
Category:	Environment functions, xHarbour extensions
Source:	rtl\winos.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Os_IsWinVista()

Checks if the application is running on Windows Vista.

Syntax

```
Os_IsWinVISTA() --> lIsWinVista
```

Return

The function returns .T. (true) if the application runs on Windows Vista, otherwise .F. (false) is returned.

Description

Os_IsWinVista() belongs to a group of environment functions that test if an xHarbour application is running on a particular Windows operating system version.

Info

See also:	Os(), Os_IsWinNT(), Os_IsWin9X(), Os_IsWin95(), Os_IsWin98(), Os_IsWinME(), Os_IsWinNT351(), Os_IsWinNT4(), Os_IsWinXP(), Os_IsWin2003(), Os_IsWtsClient(), Os_VersionInfo()
Category:	Environment functions, xHarbour extensions
Source:	rtl\winos.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Os_IsWinXP()

Checks if the application is running on Windows XP.

Syntax

```
Os_IsWinXP() --> lIsWinXP
```

Return

The function returns .T. (true) if the application runs on Windows XP, otherwise .F. (false) is returned.

Description

Os_IsWinXP() belongs to a group of environment functions that test if an xHarbour application is running on a particular Windows operating system version.

Info

See also:	Os(), Os_IsWinNT(), Os_IsWin9X(), Os_IsWin95(), Os_IsWin98(), Os_IsWinME(), Os_IsWinNT351(), Os_IsWinNT4(), Os_IsWin2000(), Os_IsWin2003(), Os_IsWtsClient(), Os_VersionInfo()
Category:	Environment functions, xHarbour extensions
Source:	rtl\winos.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Os_IsWtsClient()

Checks if the application is running on a Windows Terminal Server client.

Syntax

```
Os_IsWtsClient() --> lIsWtsClient
```

Return

The function returns .T. (true) if the application runs on a Windows Terminal Server client, otherwise .F. (false) is returned.

Description

Os_IsWtsClient() belongs to a group of environment functions that test if an xHarbour application is running on a particular Windows operating system version.

Info

See also:	Os(), Os_IsWinNT(), Os_IsWin9X(), Os_IsWin95(), Os_IsWin98(), Os_IsWinME(), Os_IsWinNT351(), Os_IsWinNT4(), Os_IsWinXP(), Os_IsWin2000(), Os_IsWin2003(), Os_VersionInfo()
Category:	Environment functions, xHarbour extensions
Source:	rtl\winos.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Os_NetRegOk()

Checks for correct network registry settings on Windows platforms.

Syntax

```
Os_NetRegOk( [<lFixIt>] ) --> lIsOK
```

Arguments

<lFixIt>

This parameter defaults to .F. (false) which leaves the registry unchanged. Passing .T. (true) changes registry settings for correct networking operations.

Return

The function returns .T. (true) when the registry settings on Windows platforms are correct for networking operations, otherwise .F. (false) is returned.

Info

See also: OS_NetVRedirOk(), QueryRegistry()
Category: Network functions, xHarbour extensions
Source: rtl\winos.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example outlines how the registry settings can be checked for
// correct values in network operations.

   PROCEDURE Main()
      LOCAL lFixIt := .F.
      LOCAL nError := 0

      CLS
      ? 'Checking network settings for Windows'
      ? '-----------------------------------------'
      ?
      IF OS_IsWTSClient() .AND. .NOT. OS_NetRegOK()
         ? 'Registry on WTS server is not set correctly for networking.'
      ELSEIF OS_NetRegOK( lFixIt )
         ? 'Registry set correctly for networking'
      ELSE
         ? 'Failed to set registry - May need "Administrator" rights'
      ENDIF
      ?
      IF .NOT. OS_NetVRedirOk( @nError )
         ? 'Invalid RVREDIR.VXD file installed'
         IF nError = 950
            ? 'You need file VREDRUPD.EXE if your vredir.vxd is dated "09:50:00"'
         ELSEIF nError == 1111
            ? 'You need file VRDRUPD.EXE if your vredir.vxd is dated "11:11:10"'
         ENDIF
       ENDIF

   RETURN
```

OS_NetVRedirOk()

Checks for the correct VREDIR.VXD file.

Syntax

```
OS_NetVRedirOk( [@<nErrorCode>] ) --> lIsOK
```

Arguments

<nErrorCode>

If passed by reference, this parameter gets assigned a numeric value indicating if a correct VREDIR.VXD file is installed.

Return

The function returns .T. (true) if the file VREDIR.VXD for Windows 9x is OK or when the operating system is not Windows 9x. Otherwise .F. (false) is returned and the VREDIR patch should be applied for reliable networking. The reference parameter can have the following values:

Error codes

Value	Description
0	Everything is OK
950	Microsoft patch file is VREDRUPD.EXE
1111	Microsoft patch file is VRDRUPD.EXE

Description

The function checks if the Windows version is Windows 9x and compares the time stamp and file size of VREDIR.VXD against known faulty versions.

Info

See also: Os_NetRegOk(), QueryRegistry()
Category: Network functions, xHarbour extensions
Source: rtl\winos.prg
LIB: xhb.lib
DLL: xhbdll.dll

Os_VersionInfo()

Retrieves specific version information about the operating system.

Syntax

```
Os_VersionInfo() --> aVersionInfo
```

Return

The function returns a one-dimensional array with five elements holding version information data of the operating system. If the operating system does not provide version information, the return value is NIL.

Description

The elements of the returned array contain the following data

Version information array

Element	Description
1	Major version number 3 = Windows NT 3.51 4 = Windows 95, 98, ME or NT 4.0 5 = Windows 2000, XP, 2003 Server
2	Minor version number 0 = Windows 95 10 = Windows 98 90 = Windows Me 51 = Windows NT 3.51 0 = Windows NT 4.0 0 = Windows 2000 1 = Windows XP 2 = Windows Server 2000
3	Build number of the operating system
4	Platform identifier 0 = VER_PLATFORM_WIN32s 1 = VER_PLATFORM_WIN32_WINDOWS 2 = VER_PLATFORM_WIN32_NT
5	Service Pack number or additional information about the OS

Info

See also: HB_BuildInfo(), Os(), Os_IsWinNT(), Os_IsWin9X(), Os_IsWin95(), Os_IsWin98(), Os_IsWinME(), Os_IsWinNT351(), Os_IsWinNT4(), Os_IsWinXP(), Os_IsWin2000(), Os_IsWin2003(), Os_IsWtsClient(), Version()
Category: Environment functions, xHarbour extensions
Source: rtl\winos.prg
LIB: xhbdll.lib
DLL: xhbdll.dll

Example

```
// The example lists the version information of the operating system

   PROCEDURE Main
      AEval( OS_VersionInfo(),  {|x| QOut(x)} )
   RETURN
```

OutErr()

Writes values to the standard error device.

Syntax

```
OutErr( <expression,...> ) --> NIL
```

Arguments

<expression,...>

This is a comma separated list of expressions whose values are output.

Return

The return value is always NIL.

Description

OutErr() evaluates *<expression,...>* and writes the resulting values to the standard error device. This allows for collecting output in a file when program output is redirected on the command line to the error device (c:\xhb\program.exe 2> logfile.txt).

OutErr() works identical as OutStd() except for using a different output channel.

Info

See also:	DispOut(), OutStd()
Category:	Output functions
Source:	rtl\console.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The examlple illustrates how output sent to the standard devices
// STDOUT and STDERR can be redirected into two different files. To
// run the example, use this command line syntax:
//
//  c:\xhb\Test.exe > regular.txt 2> error.txt

   PROCEDURE Main

      OutStd( "Regular log info", Date(), Time() )

      OutErr( "Error log info", Date(), Time() )

   RETURN
```

OutStd()

Writes values to the standard output device.

Syntax

```
OutStd( <expression,...> ) --> NIL
```

Arguments

```
<expression,...>
```

This is a comma separated list of expressions whose values are output.

Return

The return value is always NIL.

Description

OutStd() evaluates *<expression,...>* and writes the resulting values to the standard output device. This allows for collecting output in a file when program output is redirected on the command line to the error device (c:\xhb\program.exe > logfile.txt).

Note: if an xHarbour console application does not require sophisticated screen output, the commands ? and ?? can be replaced with OutStd() by including the file SIMPLEIO.CH.

Info

See also: DispOut(), FWrite(), OutErr(), QOut() | QQOut()
Category: Output functions
Source: rtl\console.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The examlple illustrates how output sent to the standard devices
// STDOUT and STDERR can be redirected into two different files. To
// run the example, use this command line syntax:
//
//  c:\xhb\Test.exe > regular.txt 2> error.txt

   PROCEDURE Main

      OutStd( "Regular log info", Date(), Time() )

      OutErr( "Error log info", Date(), Time() )

   RETURN
```

PadC() | PadL() | PadR()

Pads values of data type Character, Date and Numeric with a fill character.

Syntax

```
PadC( <expression>, <nLength>, [<cFillChar>] ) --> cPaddedString
PadL( <expression>, <nLength>, [<cFillChar>] ) --> cPaddedString
PadR( <expression>, <nLength>, [<cFillChar>] ) --> cPaddedString
```

Arguments

`<expression>`

An expression yielding a Character, Date or Numeric value.

`<nLength>`

A numeric value indicating the length of the returned string.

`<cFillChar>`

A single character used to pad the value of *<expression>* with. It defaults to a space character (Chr(32)).

Return

The function returns a character string of *<nLength>* characters. It contains the value of *<expression>* padded with *<cFillChar>*. If the value of *<expression>* contains more than *<nLength>* characters, the value in the returned string is truncated to *<nLength>* characters.

Description

PadC(), PadL(), and PadR() are conversion functions that convert the value of an expression to a character string, padded with a fill character. The functions accept for *<expression>* values of data type Character, Date and Numeric.

PadC() The fill character is added on the left and the right side, so that the value of *<expression>* appears centered in the result string.

PadL() The fill character is added on the left side, so that the value of *<expression>* appears right justified in the result string.

PadL() The fill character is added on the right side, so that the value of *<expression>* appears left justified in the result string.

Note: function Pad() is a synonym for PadR().

Info

See also:	Alltrim(), LTrim(), RTrim(), StrTran(), Stuff(), SubStr()
Category:	Character functions, Conversion functions
Source:	rtl/pad.c, rtl/padc.c, rtl/padl.c, rtl/padr.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows the results of creating padded character strings

   PROCEDURE Main
      ? PadC( "xHarbour", 12, "_" )   // result: __xHarbour__
      ? PadL( "xHarbour", 12, "_" )   // result: ____xHarbour
      ? PadR( "xHarbour", 12, "_" )   // result: xHarbour____
```

```
   ? "0x" + PadL( 2, 4, "0" )       // result: 0x0002
RETURN
```

PadLeft()

Pads a character string on the left.

Syntax

```
PadLeft( <cString>, ;
         <nLength>, ;
        [<xChar>]   ) --> cResult
```

Arguments

<cString>

This is the string to process.

<nLength>

A numeric value indicating the length of the returned string.

<xChar>

This is a single character or its numeric ASCII code that is added at the beginning of *<cString>*. It defaults to a blank space (Chr(32)).

Return

The function fills *<cString>* on the left side with *<xChar>* up to the length *<nLength>* and returns the result. If *<nLength>* is smaller than Len(*<cString>*) the string is truncated on the left side.

Info

See also: Center(), PadC() | PadL() | PadR(), PadRight()
Category: CT:String manipulation, Character functions
Source: ct\ctpad.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example outlines the difference between PadL() and PadLeft()
// when the padding length is smaller that the string length.

   PROCEDURE Main
      LOCAL cString := "12345"

      ? PadL( cString, 3 )           // result: 123
      ? PadLeft( cString, 3 )        // result: 345

      ? PadL( cString, 7, "_" )      // result: __12345
      ? PadLeft( cString, 7, "_" )   // result: __12345
   RETURN
```

PadRight()

Pads a character string on the right.

Syntax

```
PadLeft( <cString>, ;
         <nLength>, ;
        [<xChar>]   ) --> cResult
```

Arguments

`<cString>`

This is the string to process.

`<nLength>`

A numeric value indicating the length of the returned string.

`<xChar>`

This is a single character or its numeric ASCII code that is added at the end of *<cString>*. It defaults to a blank space (Chr(32)).

Return

The function fills *<cString>* on the right side with *<xChar>* up to the length *<nLength>* and returns the result. If *<nLength>* is smaller than Len(*<cString>*) the string is truncated on the right side.

Info

See also:	Center(), PadC() | PadL() | PadR(), PadLeft()
Category:	CT:String manipulation, Character functions
Source:	ct\ctpad.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example outlines the difference between PadR() and PadRight()
// PadR() ignores a numeric third parameter

   PROCEDURE Main
      LOCAL cString := "12345"

      ? PadRight( cString, 3 )             // result: 123
      ? PadR( cString, 3 )                 // result: 123

      ? ">"+PadRight( cString, 7, 65 )+"<" // result: >12345AA<
      ? ">"+PadR( cString, 7, 65 )+"<"     // result: >12345  <

   RETURN
```

Payment()

Calculates a periodical payment for loans.

Syntax

```
Payment( <nLoan>         , ;
         <nInterestRate> , ;
         <nPeriods>        ) --> nPayment
```

Arguments

<nLoan>

A numeric value defining the amount of borrowed capital.

<nInterestRate>

This parameter defines a constant interest rate per period during the periods of paying back a loan in the range of 0 to 1 (1 equals 100% interest rate).

<nPeriods>

This is a numeric value defining the number of periods to pay back the loan.

Return

The function returns a numeric value. It is the constant payment necessary to pay back a loan over *<nPeriods>* amounts of time at a fixed interest rate.

Info

See also: Fv(), Periods(), Pv(), Rate()
Category: CT:Math, Financial functions, Mathematical functions
Source: ct\finan.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example calculates the monthly payment required to pay back
// a loan of 10000 units of money over a period of 36 months at an
// interest rate of 8% per year.

   PROCEDURE Main
      LOCAL nLoan       := 10000      // borrowed capital
      LOCAL nInterest   := 0.08/12    // monthly interest rate
      LOCAL nMonths     := 36         // total payback period

      ? Payment( nLoan, nInterest, nMonths )  // result: 313.36

   RETURN
```

PCol()

Returns the column position of the print head.

Syntax

```
PCol() --> nColumn
```

Return

The function returns a numeric value indicating the current column position of the print head.

Description

PCol() reports the current column position of the print head when console output is directed to the printer after SET DEVICE TO PRINTER or SET PRINTER ON is issued. During printer output, PCol() maintains an internal counter which is initialized with zero. The counter is incremented by 1 for each character sent to the printer, and reset to zero when a form feed occurs with the EJECT command, or when a carriage return character (Chr(13)) is sent to the printer.

The function increments the counter for all characters, including printer control characters, or escape sequences, that do not print. If a control character is sent to the printer, the current print head position must be saved using PCol() and PRow(), and restored with SetPrc().

Note that PCol() cannot be used with proportional fonts.

Info

See also:	?|??, @...SAY, Col(), EJECT, IsPrinter(), PadC() | PadL() | PadR(), PRow(), QOut() | QQOut(), Row(), SET DEVICE, SET PRINTER, SetPrc()
Category:	Environment functions
Source:	rtl\console.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example prints a telephone list from an address database.

   PROCEDURE Main
      LOCAL nPageSize := 60
      LOCAL nCurPage  := 0
      LOCAL nCurLine  := 99

      USE Customer
      SET DEVICE TO PRINTER

      DO WHILE .NOT. EoF()
         IF nCurLine > nPageSize
            EJECT
            nCurLine := 1
            nCurPage ++
            @ nCurLine, 60 SAY "Page: " + LTrim(Str(nCurPage))
            nCurLine += 2
         ENDIF

         @ nCurLine, 0          SAY Trim(FIELD->LastName)+","
         @ nCurLine, PCol() + 1 SAY Trim(FIELD->FirstName)
         @ nCurLine, 40         SAY FIELD->PHONE PICTURE "@R (999) 999-9999"

         SKIP
         nCurLine++
```

```
   ENDDO

   SET DEVICE TO SCREEN

   CLOSE Customer
RETURN
```

PCount()

Returns the number of passed parameters.

Syntax

```
PCount() --> nParamCount
```

Return

The function returns a numeric value indicating the number of parameters passed to a function, procedure or method.

Description

PCount() reports the number of parameters passed to a called function, procedure or method. Note that if parameters are omitted in a call, they are initialized with NIL in the called routine. PCount() reports the last parameter passed to a called routine (see the example).

Info

See also: FUNCTION, HB_ArgC(), PValue(), PROCEDURE, Valtype()
Category: Debug functions, Environment functions
Source: vm\pcount.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example shows the result of PCount() when a function is
// called with different numbers of parameters. Omitted parameters
// are implicitely passed as NIL values.

   PROCEDURE Main
      ? Test()                          // result: 0

      ? Test( "A" )                     // result: 1
      ? Test( NIL )                     // result: 1

      ? Test( "A", "B" )                // result: 2
      ? Test( "A",     )                // result: 2
      ? Test( "A", NIL )                // result: 2
      ? Test(    , "B" )                // result: 2
      ? Test(    ,     )                // result: 2

      ? Test( "A", "B", "C" )           // result: 3
      ? Test( "A",    , "C" )           // result: 3
      ? Test(    ,    , "C" )           // result: 3
      ? Test(    ,    , NIL )           // result: 3
      ? Test(    ,    ,     )           // result: 3
   RETURN

   FUNCTION Test
   RETURN PCount()
```

Periods()

Calculates the duration for paying back a loan at fixed interest rate and payments.

Syntax

```
Periods( <nLoan>        , ;
         <nPayBack>     , ;
         <nInterestRate> ) --> nDuration
```

Arguments

<nLoan>

A numeric value defining the amount of borrowed capital.

<nPayBack>

A numeric value defining the constant, payback amount.

<nInterestRate>

This parameter defines a constant interest rate per period during the periods of paying back a loan in the range of 0 to 1 (1 equals 100% interest rate).

Return

The function returns a numeric value. It is the number or periods (the duration) a loan must be payed back at fixed payments and fixed interest rate.

Info

See also: Fv(), Payment(), Pv(), Rate()
Category: CT:Math, Financial functions, Mathematical functions
Source: ct\finan.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example calculates the duration of a payback period for
// a loan in months. Monthly payment to pay back a loan of 10000
// units of money is set at 200 units of money. Interest rate is
// 8% per year.

   PROCEDURE Main
      LOCAL nLoan     := 10000      // borrowed capital
      LOCAL nPayback  := 200        // payback units of money per month
      LOCAL nInterest := 0.08/12    // monthly interest rate

      ? Periods( nLoan, nPayback, nInterest )  // result: 61.02 (months)

   RETURN
```

Pi()

Returns Pi with highest accuracy.

Syntax

```
Pi() --> nPi
```

Return

The function returns the number *Pi* with highest possible precision.

Info

See also:	Cos(), Sin(), Tan()
Category:	CT:Math, Mathematical functions, Trigonometric functions
Source:	ct\trig.c
LIB:	xhb.lib
DLL:	xhbdll.dll

PosAlpha()

Returns the position of the first alphabetic characters in a character string.

Syntax

```
PosAlpha( <cString>   , ;
         [<lNoAlpha>] , ;
         [<nSkipChars>] ) --> nPos
```

Arguments

`<cString>`

This is the character string to search for the first matching charatcer.

`<lNoAlpha>`

This parameter defaults to .F. (false) so that the first alphabetic character is searched. Passing .T. (true) instructs the function to search for the first non-alphabetic character.

`<nSkipChars>`

This numeric parameter defaults to 0 so that the function begins searching with the first character of *<cString>*. Passing a value > 0 instructs the function to ignore the first *<nSkipChars>* characters in the search.

Return

Depending on *<lNoAlpha>*, the function returns the position of the first (non)alphabetic character in *<cString>* as a numeric value.

Info

See also:	PosLower(), PosRange(), PosUpper()
Category:	CT:String manipulation, Character functions
Source:	ct\pos1.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays results of PosAlpha()

   PROCEDURE Main
      LOCAL cString := "January 18th, 2007"
      LOCAL nPos, cChr

      ? nPos := PosAlpha( cString )             // result:   1
      ? cChr := cString[ nPos ], ;              // result:   J   74
        Asc( cChr )

      ? nPos := PosAlpha( cString, .T. )        // result:   8
      ? cChr := cString[ nPos ], ;              // result:       32
        Asc( cChr )

      ? nPos := PosAlpha( cString, .F., nPos ) // result:  11
      ? cChr := cString[ nPos ], ;              // result:   t  116
        Asc( cChr )

   RETURN
```

PosChar()

Replaces a single character at a specified position in a string.

Syntax

```
PosChar( <cString>, <xChar>, [<nPos>] ) --> cResult
```

Arguments

<cString>

This is the character string to replace a single character in.

<xChar>

The replacment character can be specified as a character or its numeric ASCII code.

<nPos>

Optionally, the position of the character to replace can be specified as a numeric value. By default, *<nPos>* specifies the last character (Len(*<cString>*))

Return

The function substitues the character at the specified position and returns the result. If CSetRef() is set to .T. (true), pass *<cString>* be reference to avoid the return value.

Note: the function exists for compatibility reasons. The [ ] operator can be applied to character strings in xHarbour and is more efficient for replacing a single character.

Info

See also:	[ ] (string), CharRepl(), CSetRef(), Stuff()
Category:	CT:String manipulation, Character functions
Source:	ct\pos2.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows different ways of replacing a
// single character in a string.

   PROCEDURE Main
      LOCAL cString := "xHarbour"

      // compatibility
      CSetRef( .F. )
      ? PosChar( cString, "Y" )      // result: xHarbouY
      ? PosChar( cString, "Z", 1 )   // result: ZHarbour
      ? cString                      // result: xHarbour

      CSetRef( .T. )
      ? PosChar( @cString, "Y" )     // result: NIL
      ? PosChar( @cString, "Z", 1 ) // result: NIL
      ? cString                      // result: ZHarbouY

      // xHarbour [] operator
      ? cString := "xHarbour"        // result: xHarbour
      ? cString[-1] := "Y"           // result: Y
      ? cString[ 1] := "Z"           // result: Z
      ? cString                      // result: ZHarbouY
```

```
RETURN
```

PosDel()

Deletes character(s) at a specified position in a string.

Syntax

```
PosDel( <cString>, [<nPos>], [<nCount>] ) --> cResult
```

Arguments

<cString>

This is the character string to delete one or more characters from.

<nPos>

This is the numeric ordinal position of the first character in *<cString>* to start deleting from. It defaults to the last character (Len(*<cString>*)).

<nCount>

A numeric value specifies the number of characters to delete. The default value is 1 character.

Return

The function removes the requested numnber of characters and returns the modified string.

Info

See also: AtRepl(), PosIns(), PosRange(), Stuff()
Category: CT:String manipulation, Character functions
Source: ct\pos2.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays results of PosDel()

   PROCEDURE Main
      LOCAL cString := "xHarbour"

      ? PosDel( cString )          // result: xHarbou

      ? PosDel( cString, 3, 2 )    // result: xHbour

      ? PosDel( cString,  , 3 )    // result: xHarb
   RETURN
```

PosDiff()

Searches the first position where two character strings differ.

Syntax

```
PosDiff( <cString1>, <cString2>, [<nSkipChars>]) --> nPos
```

Arguments

`<cString1> and <cString2>`

These are two character strings whose characters are compared.

`<nSkipChars>`

This numeric parameter defaults to 0 so that the function begins searching with the first character of *<cString1>*. Passing a value > 0 instructs the function to ignore the first *<nSkipChars>* characters in the search.

Return

The function returns the ordinal position of the first character that differs in both input strings, or 0 if both strings are identical.

Info

See also:	PosEqual()
Category:	CT:String manipulation, Character functions
Source:	ct\posdiff.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays results of PosDiff()

   PROCEDURE Main
      LOCAL cStr1 := "xHarbour compiler"
      LOCAL cStr2 := "xHarbour company"

      ? PosDiff( cStr1, cStr2 )         // result: 14

      ? PosDiff( cStr1, Upper(cStr2) )  // result:  1
   RETURN
```

PosEqual()

Searches the first position where two character strings are equal.

Syntax

```
PosEqual( <cString1>  , ;
          <cString2>  , ;
         [<nCompare>] , ;
         [<nSkipChars>] ) --> nPos
```

Arguments

`<cString1> and <cString2>`

These are two character strings whose characters are compared.

`<nCompare>`

A numeric parameter specifies the number of characters to be equal in both input strings. It defaults to the expression: Min(Len(<*cString1*>), Len(<*cString2*>)).

`<nSkipChars>`

This numeric parameter defaults to 0 so that the function begins searching with the first character of <*cString1*>. Passing a value > 0 instructs the function to ignore the first <*nSkipChars*> characters in the search.

Return

The function returns the ordinal position of the first character that is the same in both input strings, or 0 if both strings are completely different.

Info

See also:	PosEqual()
Category:	CT:String manipulation, Character functions
Source:	ct\posdiff.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays results of PosEqual()

   PROCEDURE Main
      LOCAL cStr1 := "xHarbour compiler"
      LOCAL cStr2 := " Clipper compiler"

      // check if strings are equal
      ? PosEqual( cStr1, cStr2 )         // result:  0

      // find a minimum of 4 equal characters
      ? Posequal( cStr1, cStr2, 4 )      // result:  8
   RETURN
```

PosIns()

Inserts character(s) at a specified position in a string.

Syntax

```
PosIns( <cString>, ;
        <cInsert>, ;
       [<nPos>]    ) --> cResult
```

Arguments

<cString>

This is the character string to delete one or more characters from.

<cInsert>

This is a character string of one or more characters to be inserted.

<nPos>

This is the numeric ordinal position of the first character in *<cString>* to start deleting from. It defaults to Len(*<cString>*)+1, i.e. *<cInsert>* is appended to *<cString>*.

Return

The function inserts *<cInsert>* at the specified position into *<cString>* and returns the modified string.

Info

See also:	PosDel(), PosRepl(), Stuff()
Category:	CT:String manipulation, Character functions
Source:	ct\pos2.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example display results of PosIns()

   PROCEDURE Main
      LOCAL cStr := "This compiler"

      cStr := PosIns( cStr, "is the ", 6 )

      ? cStr          // result: This is the compiler

      cStr := PosIns( cStr, "xHarbour ", 13 )

      ? cStr          // result: This is the xHarbour compiler
   RETURN
```

PosLower()

Returns the position of the first lower case letter in a character string.

Syntax

```
PosLower( <cString>   , ;
         [<lNoLower>] , ;
         [<nSkipChars>] ) --> nPos
```

Arguments

<cString>

This is the character string to search for the first lower case charatcer.

<lNoLower>

This parameter defaults to .F. (false) so that the first lower case character is searched. Passing .T. (true) instructs the function to search for the first non-lower case character.

<nSkipChars>

This numeric parameter defaults to 0 so that the function begins searching with the first character of *<cString>*. Passing a value > 0 instructs the function to ignore the first *<nSkipChars>* characters in the search.

Return

Depending on *<lNoLower>*, the function returns the position of the first (non)lower case character in *<cString>* as a numeric value.

Info

See also: PosAlpha(), PosRange(), PosUpper()
Category: CT:String manipulation, Character functions
Source: ct\pos1.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays results of PosLower()

   PROCEDURE Main
      LOCAL cString := "xHarbour"
      LOCAL nPos, cChr

      ? nPos := PosLower( cString )             // result:   1
      ? cChr := cString[ nPos ], ;              // result:   x  120
        Asc( cChr )

      ? nPos := PosLower( cString, .T. )        // result:   2
      ? cChr := cString[ nPos ], ;              // result:   H   72
        Asc( cChr )

      ? nPos := PosLower( cString, .F., nPos ) // result:   3
      ? cChr := cString[ nPos ], ;              // result:   a   97
        Asc( cChr )
   RETURN
```

PosRange()

Retrieves the position of the first character out of a range found in a string.

Syntax

```
PosRange( <cRangeStart> , ;
          <cRangeEnd>   , ;
          <cString>     , ;
         [<lNotInRange>], ;
         [<nSkipChars>]   ) --> nPos
```

Arguments

`<cRangeStart> and <cRangeEnd>`

These are two single characters marking the begin and end of the range of characters to be included in the search.

`<cString>`

This is the character string to search for the first charatcer out of a range of characters.

`<lNotInRange>`

This parameter defaults to .F. (false) so that the first character within the defined range of characters is searched. Passing .T. (true) instructs the function to search for the first character not within the range.

`<nSkipChars>`

This numeric parameter defaults to 0 so that the function begins searching with the first character of *<cString>*. Passing a value > 0 instructs the function to ignore the first *<nSkipChars>* characters in the search.

Return

Depending on *<lNotInRange>*, the function returns the position of the first character in *<cString>*, that is (not) within the defined range, as a numeric value.

Note: the function is designed for easily finding non-printable characters in a character string. In this case Chr(0) and Chr(31) mark the start and end of the range of characters to search for.

Info

See also:	PosAlpha(), PosLower(), PosUpper(), RangeRem(), RangeRepl()
Category:	CT:String manipulation, Character functions
Source:	ct\pos1.c
LIB:	xhb.lib
DLL:	xhbdll.dll

PosRepl()

Replaces characters in a string beginning at a specified position.

Syntax

```
PosRepl( <cString> , ;
         <cReplace>, ;
        [<nPos>]     ) --> cResult
```

Arguments

<cString>

This is the character string to replace one or more characters in.

<cReplace>

This is a character string of one or more characters that replaces characters in *<cString>*.

<nPos>

This is the numeric ordinal position of the first character in *<cString>* to start deleting from. It defaults to Len(*<cString>*)-Len(*<cReplace>*), i.e. *<cReplace>* replaces the characters at the end of *<cString>*.

Return

The function replaces *<cReplace>* at the specified position in *<cString>* and returns the modified string.

Info

See also: PosDel(), PosIns(), PosRange()
Category: CT:String manipulation, Character functions
Source: ct\pos2.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays results of PosRepl()

   PROCEDURE Maine
      LOCAL cString := "12345678"

      ? PosRepl( cString, "ABC" )    // result: 12345ABC

      ? PosRepl( cString, "ABC", 2 ) // result: 1ABC5678

      ? PosRepl( cString, "ABC", 7 ) // result: 123456ABC
   RETURN
```

PosUpper()

Returns the position of the first upper case letter in a character string.

Syntax

```
PosUpper( <cString>   , ;
         [<lNoUpper>] , ;
         [<nSkipChars>] ) --> nPos
```

Arguments

<cString>

This is the character string to search for the first upper case charatcer.

<lNoLower>

This parameter defaults to .F. (false) so that the first upper case character is searched. Passing .T. (true) instructs the function to search for the first non-upper case character.

<nSkipChars>

This numeric parameter defaults to 0 so that the function begins searching with the first character of *<cString>*. Passing a value > 0 instructs the function to ignore the first *<nSkipChars>* characters in the search.

Return

Depending on *<lNoUpper>*, the function returns the position of the first (non)upper case character in *<cString>* as a numeric value.

Info

See also: PosAlpha(), PosLower(), PosRange()
Category: CT:String manipulation, Character functions
Source: ct\pos1.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays results of PosUpper()

   PROCEDURE Main
      LOCAL cString := "xHarbour Compiler"
      LOCAL nPos, cChr

      ? nPos := PosUpper( cString )            // result:   2
      ? cChr := cString[ nPos ], ;             // result:   H   72
        Asc( cChr )

      ? nPos := PosUpper( cString, .F., nPos ) // result:  10
      ? cChr := cString[ nPos ], ;             // result:   C   67
        Asc( cChr )

      ? nPos := PosUpper( cString, .T. )       // result:   1
      ? cChr := cString[ nPos ], ;             // result:   x  120
        Asc( cChr )

   RETURN
```

PrgExpToVal()

Converts a character string obtained from ValToPrgExp() back to the original data type.

Syntax

```
Prgexptoval( <cString> ) --> xValue
```

Arguments

<cString>

This is a character string previously obtained from ValToPrgExp().

Return

The function returns the value of the expression encoded in *<cString>*.

Description

PrgExpToVal() is the reverse function of ValToPrgExp(). It exists mainly for completeness, since it merely invokes the macro compiler and returns the result of the expression represented by *<cString>*.

Info

See also:	&, CStr(), HB_Serialize(), Valtype(), ValToPrg(), ValToPrgExp()
Category:	Conversion functions, xHarbour extensions
Source:	rtl\cstr.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

PrinterExists()

Checks if a particular printer is installed.

Syntax

```
PrinterExists( <cPrinterName> ) --> lInstalled
```

Arguments

```
<cPrinterName>
```

This is a character string holding the name of a printer to check.

Return

The function returns .T. (true) if the specified printer is installed, otherwise .F. (false) is returned.

Description

This function checks whether a particular printer is installed or not. The printer name is case sensitive and must be spelled in the exact same way as listed in the control panel.

Info

See also:	GetPrinters(), GetDefaultPrinter(), IsPrinter(), PrinterPortToName()
Category:	Printer functions, xHarbour extensions
Source:	rtl\tprinter.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example outlines case sensitivity of printer names.

   PROCEDURE Main()
      ? PrinterExists( "HP LaserJet 1200 Series PCL" ) // result: .T.
      ? PrinterExists( "HP LASERJET 1200 SERIES PCL" ) // result: .F.
   RETURN
```

PrinterPortToName()

Retrieves the name of the printer connected to a printer port.

Syntax

```
PrinterPortToName( <cPortName> ) --> cPrinterName
```

Arguments

<cPortName>

This is a character string holding the port name where a printer is connected to.

Return

The function returns a character string holding the name of a printer, or an empty string ("") if the port name does not exist or no printer is connected to the specified port.

Description

Function PrinterPortToName() retrieves the name of the printer which is connected to the port *<cPortName>*. The port name must be specified including the colon, like "LPT1:".

Info

See also: GetPrinters(), GetDefaultPrinter(), IsPrinter(), PrinterExists()
Category: Printer functions, xHarbour extensions
Source: rtl\tprinter.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example queries the printer ports LPT1: to LPT5: and
// lists the names of connected printers

   PROCEDURE Main()
      LOCAL aPrinters := {}
      LOCAL i, cPrinter, cPort

      FOR i:=1 TO 5
         cPort    := "LPT" + LTrim( Str(i) ) + ":"
         cPrinter := PrinterPortToName( cPort )
         IF .NOT. Empty( cPrinter )
            AAdd( aPrinters, { cPort, cPrinter } )
         ENDIF
      NEXT

      IF Empty( aPrinters )
         ? "No printer found"
      ELSE
         AEval( aPrinters, {|a| Qout( a[1], a[2] ) } )
      ENDIF

   RETURN
```

PrintFileRaw()

Prints a file to a Windows printer in RAW mode.

Syntax

```
PrintFileRaw( <cPrinterName>, ;
              <cFileName>   , ;
             [<cJobTitle>]    ) --> nErrorCode
```

Arguments

`<cPrinter>`

This is a character string holding the name of the printer to use for printing.

`<cFileName>`

The file to print must be specified as a character string, including path and extension. If the path is omitted, the file is searched in the current directory.

`<cJobTitle>`

This is an optional character string which appears as print job description in the spooler. It defaults to *<cFileName>*.

Return

The function returns 1 on success or a value less than zero on error. See the example for error codes.

Description

Function PrintFileRaw() prints a file to a Windows printer in RAW mode. This restricts file types for printing to enhanced metafiles (EMF), ASCII text, and raw data, which includes all printer specific file types such as PostScript and PCL. The file is sent to the Windows spooler which processes the print job. The function returns zero when the file is successfully transferred to the spooler. Note that this is no guarantee for a printout. If the physical printer is out of paper, for example, the spooler reports an error to the user. This type of failure cannot be detected by PrintFileRaw().

Info

See also: GetPrinters(), GetDefaultPrinter(), PrinterExists(), PrinterPortToName()
Category: Printer functions, xHarbour extensions
Source: rtl\tprinter.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example prints a file in RAW mode and demonstrates
// the possible return values of PrintFileRaw().

   PROCEDURE Main()
      LOCAL cPrinter := GetDefaultPrinter()
      LOCAL cFile    := "MyFile.Txt"
      LOCAL nResult  := -1
      LOCAL cMsg     := "PrintFileRaw(): "

      CLS
      IF Empty( cPrinter )
         ? "No default printer found"
         QUIT
      ENDIF
```

```
   nResult := PrintFileRaw( cPrinter, cFile, "Test for PrintFileRaw()" )

   SWITCH nResult
   CASE -1
      cMsg += "Invalid parameters passed to function" ; EXIT
   CASE -2
      cMsg += "WinAPI OpenPrinter() call failed"      ; EXIT
   CASE -3
      cMsg += "WinAPI StartDocPrinter() call failed"  ; EXIT
   CASE -4
      cMsg += "WinAPI StartPagePrinter() call failed" ; EXIT
   CASE -5
      cMsg += "WinAPI malloc() of memory failed"      ; EXIT
   CASE -6
      cMsg += "File " + cFile + " not found"          ; EXIT
   DEFAULT
      cMsg += cFile + " PRINTED OK!!!"
   END

   ? cMsg
RETURN
```

PrintReady()

Tests if a printer connected to a specified port is ready.

Syntax

```
PrintReady( [<nLPT>] ) --> lPrinterIsReady
```

Arguments

<nLPT>

This is a numeric value specifying the printer port to check. It defaults to 1.

Return

The function exists for compatibility reasons. It returns always .F. (false). Use IsPrinter() to test the availability of a printer.

Info

See also: IsPrinter()
Category: CT:Printer, Printer functions
Source: ct\print.c
LIB: xhb.lib
DLL: xhbdll.dll

PrintSend()

Sends a string or a single character to the printer.

Syntax

```
PrintSend( <cString> ,[<nLPTport>] ) --> nCharsNotSend
```

Arguments

`<cString>`

A character string to send to the printer.

`<nLPTport>`

This is a numeric value specifying the printer port. It defaults to 1.

Return

The function exists for compatibility reasons. It returns always 0. Use DevOut() to send a string to the printer.

Info

Category:	CT:Printer, Printer functions
Source:	ct\print.c
LIB:	xhb.lib
DLL:	xhbdll.dll

PrintStat()

Returns the status of a printer.

Syntax

```
PrintStat( [<nLPT>] ) --> nPrinterStatus
```

Arguments

<nLPT>

This is a numeric value specifying the printer port. It defaults to 1.

Return

The function exists for compatibility reasons. It returns always 0. Use IsPrinter() to check the availability of a printer.

Info

See also:	IsPrinter()
Category:	CT:Printer, Printer functions
Source:	ct\print.c
LIB:	xhb.lib
DLL:	xhbdll.dll

ProcFile()

Determines the current PRG source code file.

Syntax

```
ProcFile( [<nCallStack>] ) --> cPrgFilename
```

Arguments

<nCallStack>

This is a numeric value >= 0 indicating the call stack position of the currently executed routine whose source code file should be returned. The default value is zero, which indicates the currently executed routine.

Return

The function returns a character string holding the name of the PRG file in which the queried routine is implemented. If *<nCallStack>* is larger than the number of entries in the call stack, or if the file name is not included as debug information in the linked executable file, an empty string is returned ("").

Description

ProcFile() is a debug function used to determine the PRG file name in which a routine activated in the call stack is implemented. It is mainly used in conjunction with error handling routines that identify the location of runtime errors. See function ErrorBlock() for a discussion of error handling.

Info

See also:	ErrorBlock(), ProcLine(), ProcName()
Category:	Debug functions, xHarbour extensions
Source:	vm\proc.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example implements a user defined function that collects call stack
// information in an array.

   FUNCTION GetCallStack()
      LOCAL aStack := {}
      LOCAL nStack := 1  // Skip the GetCallStack() function in the result

      DO WHILE .NOT. Empty( ProcName(nStack) )
         AAdd( aStack, { ProcFile(nStack), ProcName(nStack), ProcLine(nStack) }
         nStack ++
      ENDDO

   RETURN aStack
```

ProcLine()

Determines the current line number executed in a PRG source code file.

Syntax

```
ProcLine( [<nCallStack>] ) --> nPrgLineNumber
```

Arguments

<nCallStack>

This is a numeric value >= 0 indicating the call stack position of the currently executed routine whose source code line number should be returned. The default value is zero, which indicates the currently executed routine.

Return

The function returns a numeric value indicating the line number in the PRG source code file that is currently being executed. If *<nCallStack>* is larger than the number of entries in the call stack, or if the executable is created with the /l compiler switch, the return value is zero.

Description

ProcLine() is a debug function used to determine the line in the PRG source code file that is currently being executed. It is mainly used in conjunction with error handling routines that identify the location of runtime errors. See function ErrorBlock() for a discussion of error handling.

Info

See also: ErrorBlock(), ProcFile(), ProcName()
Category: Debug functions
Source: vm\proc.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example implements a user defined function that collects call stack
// information in an array.

   FUNCTION GetCallStack()
      LOCAL aStack := {}
      LOCAL nStack := 1  // Skip the GetCallStack() function in the result

      DO WHILE .NOT. Empty( ProcName(nStack) )
         AAdd( aStack, { ProcFile(nStack), ProcName(nStack), ProcLine(nStack) }
         nStack ++
      ENDDO

   RETURN aStack
```

ProcName()

Determines the symbolic name of the currently executed function, method or procedure.

Syntax

```
ProcName( [<nCallStack>] ) --> nProcName
```

Arguments

<nCallStack>

This is a numeric value >= 0 indicating the call stack position of the currently executed routine whose symbolic name should be returned. The default value is zero, which indicates the currently executed routine.

Return

The function returns a character string holding the name of the routine that is currently being executed. If *<nCallStack>* is larger than the number of entries in the call stack, or if the symbolic name is not included as debug information in the linked executable file, an empty string is returned ("").

Description

ProcName() is a debug function used to determine the symbolic name of the routine currently being executed. It is mainly used in conjunction with error handling routines that identify the location of runtime errors. See function ErrorBlock() for a discussion of error handling.

Info

See also:	ErrorBlock(), ProcFile(), ProcLine()
Category:	Debug functions
Source:	vm\proc.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example implements a user defined function that collects call stack
// information in an array.

   FUNCTION GetCallStack()
      LOCAL aStack := {}
      LOCAL nStack := 1  // Skip the GetCallStack() function in the result

      DO WHILE .NOT. Empty( ProcName(nStack) )
         AAdd( aStack, { ProcFile(nStack), ProcName(nStack), ProcLine(nStack) }
         nStack ++
      ENDDO

   RETURN aStack
```

PRow()

Returns the row position of the print head.

Syntax

```
PRow() --> nRow
```

Return

The function returns a numeric value indicating the current row position of the print head.

Description

PRow() reports the current row position of the print head when console output is directed to the printer after SET DEVICE TO PRINTER or SET PRINTER ON is issued. During printer output, PRow() maintains an internal counter which is initialized with zero. The counter is incremented by 1 when output begins in a new line, and reset to zero with the EJECT command.

The function does not increment the counter for printer control characters that advance the print head to a new line, like line feed (Chr(10)) or form feed (Chr(12)). If such a control character is sent to the printer, the current print head position must be programmatically adjusted with the SetPrc() function.

Info

See also: ?|??, @...SAY, Col(), EJECT, IsPrinter(), PadC() | PadL() | PadR(), PCol(), QOut() | QQOut(), Row(), SET DEVICE, SET PRINTER, SetPrc()
Category: Environment functions
Source: rtl\console.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// This example prints address labels using PRow() and SetPrc()

   PROCEDURE Main
      USE Customer
      INDEX ON Upper(Lastname) TO Cust01

      SET DEVICE TO PRINTER
      SetPrc( 2, 0 )

      DO WHILE .NOT. Eof()
         @ PRow()+1 , 0        SAY Trim( FIELD->Firstname )
         @ PRow()   , PCol()+1 SAY Trim( FIELD->Lastname )
         @ PRow()+1 , 0        SAY Trim( FIELD->Street )
         @ PRow()+1 , 0        SAY Trim( FIELD->City )
         @ PRow()+1 , 0        SAY Trim( FIELD->State )
         @ PRow()   , PCol()+1 SAY FIELD->Zip
         @ PRow()+1 , 0        SAY ""

         SKIP
         SetPrc( 2, 0 )
      ENDDO

      SET DEVICE TO SCREEN
      CLOSE ALL
   RETURN
```

Pv()

Calculates the present value of future capital, based on periodic investments.

Syntax

```
Pv( <nInvestment>   , ;
    <nInterestRate> , ;
    <nPeriods>        ) --> nPresentValue
```

Arguments

`<nInvestment>`

A numeric value defining the periodic investment.

`<nInterestRate>`

This parameter defines a constant interest rate per period during the periods of the investment in the range of 0 to 1 (1 equals 100% interest rate).

`<nPeriods>`

This is a numeric value defining the number of periods to include in the calculation.

Return

The function returns the present value of a periodic, constant investment at a constant interest rate over a period of time as a numeric value.

Info

See also:	Fv(), Payment(), Periods(), Rate()
Category:	CT:Math, Financial functions, Mathematical functions
Source:	ct\finan.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example calculates the present value of monthly investments of 100 units
// of money over a period of 48 months at an interest rate of 5% per year.
// (potential savings with interest)

   PROCEDURE Main
      LOCAL nInvestment := 100        // monthly savings
      LOCAL nInterest   := 0.05/12    // monthly interest rate
      LOCAL nMonths     := 48         // total savings period

      ? Pv( nInvestment, nInterest, nMonths )  // result: 4342.30

   RETURN
```

PValue()

Retrieves the value of a parameter passed to a function, method or procedure.

Syntax

```
PValue( <nParamPos> ) --> xParamValue
```

Arguments

<nParamPos>

This is a numeric value indicating the ordinal position of the parameter to check.

Return

The function returns the value of the parameter at position *<nParamPos>* passed to a function, method or procedure. If *<nParamPos>* is larger than PCount(), the return value is NIL.

Description

PValue() is useful to retrieve the value of a parameter when a function, method or procedure is called with more arguments than are declared in the formal argument list.

Info

See also:	PCount(), HB_AParams(), HB_ArgV(), Valtype()
Category:	Debug functions, Environment functions, xHarbour extensions
Source:	vm\pvalue.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example declares a function with an unknown number
// of parameters. The value of each parameter is retrieved
// with PValue().

   PROCEDURE Main

      ? Average( 10, 20 )                  // result: 15.00
      ? Average( 1,2,3,4,5,6,7,8,9,10 )    // result: 5.50

   RETURN


   FUNCTION Average( ... )
      LOCAL nSum := 0, i, imax

      imax := PCount()                     // number of passed arguments

      FOR i:=1 TO imax                     // sum data
         nSum += PValue( i )
      NEXT

   RETURN ( nSum / imax )
```

QOut() | QQOut()

Displays values of expressions to the console window.

Syntax

```
QOut( [<expression,...>] ) --> NIL
QQOut( [<expression,...>]) --> NIL
```

Arguments

`<expression>`

<expression> is an optional, comma separated list of expressions whose values are output. When no expression is specified, the QOut() function outputs a new line while QQOut() outputs nothing.

Return

The return value is always NIL.

Description

QOut() and QQOut() are text mode functions that display the result of a comma separated list of expressions to the currently selected console output device. This can be the screen, or console window, or the printer.

The difference between QOut() and QQOut() is that QOut() first outputs a carriage-return/line-feed pair so that the output of *<expression,...>* always begins at a new line, while QQOut() outputs the values of *<expression,...>* at the current cursor or printhead position.

The QOut() or QQOut() function first locates the current cursor or printhead position and shifts it one to the right. Row() and Col() are updated with the new cursor position if SET PRINTER is OFF, otherwise they are updated with the new printhead position.

Should the output of QOut() or QQOut() reach the right border of the screen (defined by MaxCol()), it will wrap to the next line. Should the output of QOut() or QQOut() reach the bottom border of the screen (defined by MaxRow()), the screen will scroll up one line.

It is possible to output the expression(s) to the printer by using SET PRINTER ON before calling QOut() or QQOut(). It is also possible to output to a text file using the SET ALTERNATE TO command, followed by the SET ALTERNATE ON command. The SET CONSOLE OFF command will prevent display on the screen without interrupting the output to the printer or text file.

When the expression(s) need formatting, use the Transform() function or any user-defined function. For padding, use any of the Pad() functions to center, left align or right align the expression(s).

Info

See also:	?|??, @...SAY, PadC() | PadL() | PadR(), SET ALTERNATE, SET CONSOLE, SET PRINTER, TEXT, Transform()
Category:	Output functions
Source:	rtl\console.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// This example displays a list of expressions:

   PROCEDURE Main
      LOCAL var1 := 1, var2 := Date(), var3 := .T.
```

```
      QOut( "This line will be displayed on the console." )
      QOut( "This line will be displayed beneath the previous line." )
      QQOut( "No carriage return / linefeed for this expression!" )
      QOut( var1, var2, var3, "These were variables" )
   RETURN
```

Quarter()

Returns the quarter a date belongs to.

Syntax

```
Quarter( [<dDate>] ) --> nQuarter
```

Arguments

`<dDate>`

An expression returning a Date value. It defaults to Date().

Return

The function returns a numeric value between 1 and 4 indicating the quarter *<dDate>* belongs to, or zero on error.

Info

See also:	Day(), Month(), Year()
Category:	CT:DateTime, Date and time
Source:	ct\dattime2.c
LIB:	xhb.lib
DLL:	xhbdll.dll

QueryRegistry()

Checks if a particular registry key with specified value exists.

Syntax

```
QueryRegistry( <nHKEY>   , ;
               <cRegPath>, ;
               <cRegKey> , ;
               <xValue>  , ;
              [<lCreate>]  ) --> lExists
```

Arguments

<nHKEY>

This numeric parameter specifies the root entry in the Windows registry to search for a registry key. #define constants are available in the Winreg.ch file that can be used for *<nHKEY>*. They begin the the prefix HKEY_.

<cRegPath>

This is a character string holding the search path in the registry to locate *<cRegKey>*. The path must include a backslash as delimiter, if required, but may neither begin or end with a backslash.

<cRegKey>

This is a character string holding the name of the registry key to check and/or assign a value to. *<cRegKey>* must be located underneath *<cRegPath>*.

<xValue>

This is a value of data type Character, Date, Logical or Numeric to compare with the value of *<cRegKey>*.

<lCreate>

This parameter defaults to .F. (false), so that *<cRegKey>* is only searched and its value compared with *<xValue>*. When *<lCreate>* is .T. (true) a non existent key is created, and the value *<xValue>* is assigned to the registry key.

Return

The function returns .T. (true), when the specified registry key exists and has the value *<xValue>*. Otherwise .F. (false) is returned.

Description

Function QueryRegistry() checks if a registry key of a particular value exists in the registry and/or creates this key/value pair.

By default, the function searches the key/value pair beginning with the root *<nHKEY>*, following the search path *<cRegPath>*. If the key/value pair exists, the return value is .T. (true). When it does not, the return value is .F. (false), unless the optional parameter *<lCreate>* isset to .T. (true). In this case, a non existent key is created. The value *<xValue>* is assigned to an existing key.

Winreg.ch

Winreg.ch adds quite some overhead to an application program by adding structure definitions. If this is not required, Winreg.ch does not need to be #included. QueryRegistry() recognizes the following values for *<nHKEY>* in addition to the HKEY_* #define constants:

Registry keys

Registry Key	Equivalent value
HKEY_LOCAL_MACHINE	0
HKEY_CLASSES_ROOT	1
HKEY_CURRENT_USER	2
HKEY_CURRENT_CONFIG	3
HKEY_LOCAL_MACHINE	4
HKEY_USERS	5

Note: on Windows NT, 2000, XP or later, the user may need certain security rights in order to be able to read and/or change the registry.

Info

See also: SetRegistry(), GetRegistry()
Category: Registry functions, xHarbour extensions
Header: Winreg.ch
Source: rtl\winreg.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example queries the registry for entries as they exist
// after the installation of xHarbour builder of October 2006.
// One registry entry does not exist and is created.

   * #include "Winreg.ch"           // not needed for this example

   #define HKEY_CURRENT_USER  0     // use alternative #define constant

   PROCEDURE Main
      LOCAL nHKey    := 0
      LOCAL cRegPath := "SOFTWARE\xHarbour.com\xHarbour Builder"

      ? QueryRegistry( nHKey, cRegPath, "Edition", "Enterprise" )
                                 // result: .T.

      ? QueryRegistry( nHKey, cRegPath, "rootdir", "C:\xhb" )
                                 // result: .T.

      // Query non existent registry key ...
      IF .NOT. QueryRegistry( nHKey, cRegPath, ;
                          "xhb build last", "October 2006" )

         // ... and create it
         QueryRegistry( nHKey, cRegPath, ;
                          "xhb build last", "October 2006", .T. )
      ENDIF

      ? GetRegistry( nHKey, cRegPath, "xhb build last" )
                                 // result: October 2006
   RETURN
```

RangeRem()

Deletes character(s) within a specified range of characters.

Syntax

```
RangeRem( <xRangeStart>, ;
          <xRangeEnd>  , ;
          <cString>      ) --> cResult
```

Arguments

`<xRangeStart> and <xRangeEnd>`

These are two single characters, or their numeric ASCII codes, marking the begin and end of the range of characters to be removed from *<cString>*. If *<xRangeStart>* is larger than *<xRangeEnd>*, the range is defined as *<xRangeStart>* to 255 and 0 to *<xRangeEnd>*.

`<cString>`

This is the character string to delete a range of characters from.

Return

The function removes all characters within the specified range and returns the modified string.

Info

See also: PosRange(), RangeRepl()
Category: CT:String manipulation, Character functions
Source: ct\range.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates results of RangeRem()

   PROCEDURE Main
      LOCAL cString := "The xHarbour Compiler"

      // remove lower case characters from "o" to "z"
      ? RangeRem( "o", "z", cString ) // result: The Hab Cmile

      // remove upper case characters from "A" to "Z"
      ? RangeRem( "A", "Z", cString ) // result: he xarbour ompiler
   RETURN
```

RangeRepl()

Repleaces character(s) within a specified range of characters.

Syntax

```
RangeRepl( <cRangeStart>, ;
           <cRangeEnd>  , ;
           <cString>    , ;
           <cReplace>     ) --> cResult
```

Arguments

<cReplace>

This is a single character which replaces all characters within the specified range.

Return

The function replaces all characters with in the specified range with *<cReplace>* and returns the modified string.

Info

See also: CharRepl(), PosRepl(), PosRange(), RangeRem()
Category: CT:String manipulation, Character functions
Source: ct\range.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates results of RangeRepl()

   PROCEDURE Main
      LOCAL cString := "The xHarbour Compiler"

      // replace lower case characters from "o" to "z"
      ? RangeRepl( "o", "z", cString, "_" )
                          // result: The _Ha_b___ C_m_ile_

      // remove upper case characters from "A" to "Z"
      ? RangeRepl( "A", "Z", cString, "?" )
                          // result: ?he x?arbour ?ompiler
   RETURN
```

RAscan()

Searches a value in an array beginning with the last element.

Syntax

```
RAscan( <aArray>  , ;
         <xbSearch>, ;
        [<nStart>] , ;
        [<nCount>] , ;
        [<lExact>] , ;
        [<lASCII>]   ) --> nElement
```

Arguments

<aArray>

This is the array to iterate.

<xbSearch>

<xbSearch> is either the value to search in *<aArray>* or a code block containing the search condition. The code block receives a single parameter which is the value stored in the current array element. The code block must return a logical value. When the return value is .T. (true), the function stops searching and returns the position of the corresponding array element.

<nStart>

This is a numeric expression indicating the first element in the array to begin the search with. It defaults to Len(*<aArray>*), the last element of the array.

<nCount>

A numeric expression specifying the number of elements to search. It defaults to 1+Len(*<aArray>*)-*<nStart>*.

<lExact>

This parameter influences the comparison for searching character strings in *<aArray>*. If .T. (true) is passed, an exact string comparison is performed. When omitted or if .F. (false) is passed, string comparison follows the SET EXACT rules.

<lASCII>

This parameter is only relevant for arrays that contain single characters in their elements. A single character is treated like a numeric ASCII value when *<lASCII>* is .T. (true). The default is.F. (false).

Return

RAScan() returns the numeric ordinal position of the array element that contains the searched value. When no match is found, the return value is 0.

Description

The array function RAscan() traverses the array *<aArray>* for the value specified with *<xbSearch>*. If *<xbSearch>* is not a code block, RAscan() compares the values of each array element for being equal with the searched value. If this comparison yields .T. (true), the function stops searching and returns the numeric position of the array element containing the searched value.

If a code block is passed for *<xbSearch>*, it is evaluated and receives as parameter the value of the current array element. The code block must contain a comparison rule that yields a logical value. AScan() considers the value as being found when the codeblock returns .T. (true). Otherwise the function proceeds with the next array element.

Note: use function AScan() to search an array beginning with the first element.

Info

See also: AEval(), Asc(), Ascan(), ASort(), ATail(), Eval(), IN, SET EXACT
Category: Array functions, xHarbour extensions
Source: vm\arrayshb.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates the difference between AScan() and RAscan().

   PROCEDURE Main()
      LOCAL aArray := { "A", 1, Date(), 1, .F. }

      ? Ascan( aArray, 1 )              // result: 2
      ? RAScan( aArray, 1 )             // result: 4

   RETURN
```

RAt()

Locates the position of a substring within a character string.

Syntax

```
RAt( <cSearch>, <cString>, [<nStart>], [<nEnd>] ) --> nPos
```

Arguments

<cSearch>

<cSearch> is the substring to search for.

<cString>

<cString> is the searched character string.

<nStart>

A numeric expression indicating the position of the first character in *<cString>* to begin the search with. It defaults to Len(*<cString>*).

<nEnd>

A numeric expression indicating the position of the last character in *<cString>* to include in the search. It defaults to 1.

Return

The function returns a numeric value which is the position in *<cString>* where *<cSubString>* is found. The return value is zero when *<cSubString>* is not found.

Description

This function RAt() searches the string *<cString>* from right to left for the character string *<cSearch>*. The search begins at position *<nStart>* and is case-sensitive. If *<cString>* does not contain *<cSearch>*, the function returns 0.

Note: use function At() to search *<cString>* from left to right.

Info

See also:	$, At(), IN, Left(), Right(), StrTran(), SubStr()
Category:	Character functions
Source:	rtl\rat.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates various results of RAt().

   PROCEDURE Main()
      LOCAL cString := "xHarbour"

      ? RAt( "Ha", cString )           // result: 2

      ? RAt( "ARB" , cString )         // result: 0

      ? RAt( "ARB" , Upper(cString) )  // result: 3

      ? RAt( "r" , cString )           // result: 8

      ? RAt( "r" , cString, 7 )        // result: 4
```

```
   ? RAt( "r" , cString, 5, 7 )      // result: 0
RETURN
```

Rate()

Calculates the interest rate for a loan.

Syntax

```
Rate( <nLoan>, <nPayment>, <nPeriods> ) --> nInterestRate
```

Arguments

`<nLoan>`

A numeric value defining the amount of the loan.

`<nPayment>`

A numeric value defining the constant, payback amount.

`<nPeriods>`

This is a numeric value defining the number of periods during which the loan is payed back.

Return

The function returns the interest rate for a loan at constant payments over a period of time as a numeric value.

Info

See also: Fv(), Payment(), Periods(), Pv()
Category: CT:Math, Financial functions, Mathematical functions
Source: ct\finan.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example calculates the effective interest rate for a loan of 10000
// units of money that is payed back in 36 months with maonthly payments of
// 312 units of money.

   PROCEDURE Main
      LOCAL nLoan     := 10000      // borrowed capital
      LOCAL nPayment  := 312        // payback units of money per month
      LOCAL nMonths   := 36         // months to pay

      ? Rate( nLoan, nPayment, nMonths ) * 12  // result: 0.08 (8% interest)

   RETURN
```

RddInfo()

Queries and/or changes configuration data of RDDs.

Syntax

```
RddInfo( <nDefine>, [<xNewSetting>], [<cRddName>] ) --> xOldSetting
```

Arguments

`<nDefine>`

This is a numeric parameter for which #define constants exist in the file DbInfo.ch. They identify the numerous settings that can be queried or changed for replacable database drivers (RDD9 and begin with the prefix RDDI_ (see below).

`<xNewSetting>`

<xNewSetting> is an optional argument specifying a new value for the RDD setting identified by *<nDefine>*. The data type for *<xNewSetting>* depends on the RDD setting to change (see below). If the RDD setting is READONLY, the parameter is ignored.

`<cRddName>`

<cRddName> is an optional character string with the name of the RDD to query or configure. It defaults to the return value of RddSetDefault().

Return

The function returns the value of the specified RDD setting which is set before the function is called.

Description

RddInfo() is a universal function managing numerous RDD settings available in xHarbour. If an RDD does not support a setting, it may create a runtime error or simply ignore the function call upon its own discretion. In the latter case, the return value of RddInfo() is NIL.

The constants that can be used for *<nDefine>* are listed below:

RDDI_BLOB_SUPPORT	--> lIsBlobSupported (READONLY) The return value is .T. (true) when the RDD supports binary large objects (BLOBs), otherwise .F. (false) is returned. BLOBs are supported by the DBFBLOB RDD. Refer to function BlobDirectImport() for an example of BLOB usage.
RDDI_CANPUTREC	--> lCanWriteData (READONLY) See also: function RddList(). The return value is .T. (true) when data can be written record-wise instead of field-wise to the file maintained by the RDD, otherwise .F. (false) is returned. Record-wise data storage is an internal optimization of an RDD.
RDDI_ISDBF	--> lIsDbfSupported (READONLY) The return value is .T. (true) when the RDD supports databases in DBF file format, otherwise .F. (false) is returned.
RDDI_LOCKSCHEME	[*<nNewLockScheme>*] --> nOldLockScheme See also: command SET DBFLOCKSCHEME. This setting queries or changes the locking scheme for SHARED database access in a multi user environment. Refer to the SET DBFLOCKSCHEME command for a description of locking schemes. The values that can

	be used for *<nNewLockScheme>* are the same #define constants as described with the command.
RDDI_MEMOBLOCKSIZE	[*<nNewBlockSize>*] --> nOldBlockSize See also: command SET MEMOBLOCK. This setting queries or changes the default block size for memo fields to use by an RDD. If an RDD does not support memo fields, the return value is zero.
RDDI_MEMOEXT	[*<nNewFileExtension>*] --> nOldFileExtension See also: command SET MFILEEXT. This setting queries or changes the default extension for memo files to use by an RDD. If an RDD does not support memo fields, the return value is an empty string ("").
RDDI_MEMOTYPE	[*<nNewMemoType>*] --> nOldMemoType This setting queries or configures the default memo file format to use by an RDD when it creates a new database containing memo fields with the DbCreate() function. To change the default memo file format, #define constants from Dbinfo.ch must be used for *<nNewMemoType>*. They begin with the prefix DB_MEMO_. Supported memo file formats are DBT, FPT and SMT.
RDDI_MEMOVERSION	[*<nNewMemoVersion>*] --> nOldMemoVersion This setting queries or configures the a subtype, or version, for FPT memo files. It can be set in addition to the memo type RddInfo(RDDI_MEMOTYPE, DB_MEMO_FPT). To change the subtype of the FPT memo file format, #define constants from Dbinfo.ch must be used for *<nNewMemoVersion>*. They begin with the prefix DB_MEMOVER_. Supported memo file subtypes are FoxPro, SIX and FLEX.
RDDI_ORDBAGEXT	[*<cNewFileExtension>*] --> cOldFileExtension See also: function OrdBagExt(). This setting queries or changes the default extension for index files to use by an RDD. If an RDD does not support indexes, the return value is an empty string ("").
RDDI_TABLEEXT	[*<cNewFileExtension>*] --> cOldFileExtension See also: function DbTableExt(). This setting queries or changes the default extension for database files to use by an RDD. If an RDD does not support database files, the return value is an empty string ("").

Info

See also:	DbInfo(), DbOrderInfo(), DbRecordInfo(), RddList(), RddName(), RddSetDefault()
Category:	Database drivers, Info functions, xHarbour extensions
Header:	Dbinfo.ch
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

RddList()

Retrieves information about available Replaceable Database Drivers (RDDs).

Syntax

```
RddList( [<nRddType>] ) --> aRDDNames
```

Arguments

<nRddType>

A numeric value indicating the type of RDDs to list names for. #define constants contained in RDDSYS.CH are used for this parameter.

Types of RDDs

Constant	Value	Description
RDT_FULL *)	1	RDDs with full functionality
RDT_TRANSFER	2	RDDs having only import/export capabilities

**) default*

Return

The function returns a one-dimensional array whose elements contain the names of replaceable database drivers as character strings.

Description

RddList() is an informational RDD function used to obtain the names of RDDs available at runtime of an xHarbour application.

Info

See also: RddInfo(), RddName(), RddSetDefault()
Category: Database drivers
Header: rddsys.ch
Source: rdd\dbcmd.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example requests various RDDs to be linked and lists
// the RDD names.

   #include "RDDSYS.CH"

   REQUEST DBFCDX
   REQUEST SDF
   REQUEST DELIM

   PROCEDURE Main
      LOCAL aNames

      aNames := RddList( RDT_FULL )

      ? "RDT_FULL"
      AEval( aNames, {|c| Qout(c) } )

      aNames := RddList( RDT_TRANSFER )
```

```
   ?
   ? "RDT_TRANSFER"
   AEval( aNames, {|c| Qout(c) } )
RETURN
```

RddName()

Retrieves the name of an RDD used to open files in a work area

Syntax

```
RddName() --> cRDDName
```

Return

The function returns the name of the replaceable database driver managing the files in a work area as character string. If no file is open in a work area, an empty string ("") is returned.

Description

RddName() is an informational function that retrieves the name of the RDD specified with the USE command to open files in a workarea.

Info

See also:	RddInfo(), RddList(), RddSetDefault()
Category:	Database drivers
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows the result of RddName() calls in the current and
// in aliased work areas.

   REQUEST DBFCDX

   PROCEDURE Main
      USE Customer ALIAS Cust
      USE Orders   ALIAS Ord NEW VIA "DBFCDX"

      ? RddMame()                        // result: DBFCDX
      ? Cust->( RddName() )              // result: DBFNTX
      ? Ord-> ( RddName() )              // result: DBFCDX

      CLOSE ALL
   RETURN
```

RddRegister()

Registers a user defined Replaceable Database Driver (RDD).

Syntax

```
RddRegister( <cRddName>, <nRddType> ) --> nErrorCode
```

Arguments

`<cRddName>`

This is a character string holding the name of the RDD to register in xHarbour's RDD system.

`<nRddType>`

This is a numeric value indicating the type of the RDD to register. #define constants contained in RDDSYS.CH are used for this parameter.

Types of RDDs

Constant	Value	Description
RDT_FULL	1	RDDs with full functionality
RDT_TRANSFER	2	RDDs having only import/export capabilities

Return

The function returns a numeric error code indicating a successfull RDD registration.

Return values of RddRegister()

Value	Description
0	RDD is successfully registered
1	RDD is already registered
>1	RDD is not registered due to error

Description

RddRegister() registers a user-defined RDD in xHarbour's RDD system. The easiest way of registration is to call the function from within an INIT PROCEDURE, so that the registration occurs automatically at program start.

Refer to the files in \source\rdd\usrrdd\rdds for examples of RDD registration.

Info

See also: RddInfo(), UsrRdd_ID()
Category: Database drivers, User-defined RDD, xHarbour extensions
Header: usrrdd.ch
Source: rdd\dbcmd.c
LIB: lib\xhb.lib
DLL: dll\xhbdll.dll

RddSetDefault()

Retrieves or changes the default replaceable database driver.

Syntax

```
RddSetDefault( [<cNewRDD>] ) --> cOldRDD
```

Arguments

<cNewRDD>

This is a character string holding the name of the RDD to select as the new default RDD for opening files in a work area.

Return

The function returns a character string holding the name of the RDD selected as default RDD before the function is called. If *<cNewRDD>* is specified and an RDD of this name does not exist, a runtime error is raised.

Description

RddSetDefault() queries or changes the default replaceable database driver. The default RDD is used to open or create database and index files in a work area when the VIA clause of the USE command is omitted.

Note: some RDDs, such as DBFCDX, for example, must be linked explicitly to an xHarbour application using the REQUEST statement.

Info

See also: RddInfo(), RddList(), RddName(), REQUEST
Category: Database drivers
Source: rdd\dbcmd.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example REQUESTs the DBFCDX driver, opens a database
// with DBFNTX and then changes the default RDD.

   REQUEST DBFCDX

   PROCEDURE Main
      ? RddSetDefault()                 // result: DBFNTX

      USE Customer ALIAS Cust

      ? RddSetDefault( "DBFCDX" )       // result: DBFNTX
      USE Orders   ALIAS Ord NEW

      ? RddSetDefault()                 // result: DBFCDX

      ? RddMame()                       // result: DBFCDX
      ? Cust->( RddName() )             // result: DBFNTX
      ? Ord-> ( RddName() )             // result: DBFCDX

      CLOSE ALL
   RETURN
```

ReadExit()

Enables or disables up/down arrow keys as exit keys for READ

Syntax

```
ReadExit( [<lNewMode>] ) --> lOldMode
```

Arguments

`<lNewMode>`

A logical value can be passed that defines the new value for the exit keys flag of the READ command.

Return

The function returns the previous flag as a logical value.

Description

ReadExit() retrieves or changes a logical flag of the Get system which indicates if the up/down arrow keys can be used to terminate the READ command. When .T. (true) is passed, the READ command is terminated when the first (last) Get object has input focus and the user presses the up (down) arrow key. Setting the flag to .F. (false) requires the user to press the Enter key on the last Get to terminate READ.

Info

See also:	@...GET, READ, ReadInsert()
Category:	Get system
Source:	rtl\readexit.c
LIB:	xhb.lib
DLL:	xhbdll.dll

ReadInsert()

Queries or changes the insert/overstrike mode during READ.

Syntax

```
ReadInsert [<lNewMode>] ) --> lOldMode
```

Arguments

<lNewMode>

A logical value can be passed that defines the new value for the insert/overstrike mode of the READ command.

Return

The function returns the previous mode as a logical value.

Description

The function retrieves or changes the insert/overstrike mode recognized by the READ command and MemoEdit(). When the mode is set to .T. (true), characters are inserted in the edit buffer of the current Get entry field or text. If set to .F. (false), characters are overwritten.

Info

See also:	READ, ReadExit()
Category:	Get system
Source:	rtl\readins.c
LIB:	xhb.lib
DLL:	xhbdll.dll

ReadKey()

Returns the key code of the key that has terminated READ.

Syntax

```
Readkey() --> nKeyCode
```

Return

The function returns a numeric value indicating the key that has terminated the READ command.

Termination keys

Value	Termination key
4	Up arrow
5	Down arrow
6	Page Up
7	Page Down
12	Esc
14	Ctrl+W
15	Enter or edit buffer is full
34	Ctrl+Page Up
35	Ctrl+Page Down

Note: the number 256 is added to the key code when Updated() returns .T. (true), i.e. when the user has changed any Get entry field.

Info

See also: ReadExit(), LastKey(), NextKey()
Category: Get system
Source: rtl\readkey.prg
LIB: xhb.lib
DLL: xhbdll.dll

ReadKill()

Queries or changes the READ termination flag.

Syntax

```
ReadKill( [<lNewFlag>] ) --> lOldFlag
```

Arguments

`<lNewFlag>`

A logical value can be passed that defines the new value for the termination flag of the READ command.

Return

The function returns the previous flag as a logical value.

Description

ReadKill() is a utility function of the Get system. It maintains a logical value used as termination flag of the READ command. When .T. (true) is passed, the READ command is unconditionally terminated.

Info

See also: CLEAR GETS, READ, ReadExit(), ReadUpdated()
Category: Get system
Source: rtl\getsys.prg
LIB: xhb.lib
DLL: xhbdll.dll

ReadModal()

Activates editing of @...GET entry fields in text mode.

Syntax

```
ReadModal( <GetList>      , ;
          [<nStartGet>]   , ;
          [<oTopBarMenu>], ;
          [<nMsgRow>]     , ;
          [<nMsgLeft>]    , ;
          [<nMsgRight>]   , ;
          [<cMsgColor>]    ) --> lUpDated
```

Arguments

<GetList>

This is the *GetList* array used to collet Get() objects created with the @...GET command.

<nStartGet>

This is a numeric value indicating the ordinal position of the first Get entry field held in <*GetList*> to receive input focus. It defaults to 1.

<oTopBarMenu>

Optionally, a TopBarMenu() object can be specified. In this case, the menu can be activated during ReadModal().

<nMsgRow>

This is a numeric value specifying the screen row for displaying messages of individual Get objects or the menu <*oTopBarMenu*>. The range for <*MsgnRow*> is between 0 and MaxRow().

<nMsgLeft>

This is a numeric value specifying the left screen coordinate for displaying status messages. Usually, <*nMsgLeft*> is set to the value 0.

<nMsgRight>

This is a numeric value specifying the right screen coordinate for displaying status messages. Usually, <*nMsgRight*> is set to the value of MaxCol().

<cMsgColor>

The parameter <*cMsgColor*> is an optional character string defining the color for the messages to display. It defaults to SetColor().

Return

The return value is .T. (true) if any Get object in <*GetList*> is edited and changed by the user, otherwise .F. (false) is returned.

Description

ReadModal() is the functional equivalent of the READ command. The only parameter processed in addition to READ is <*nStartGet*>. It specifies the first Get to begin editing with.

Refer to the READ command for more information.

Info

See also:	@...GET, @...SAY, Get(), LastKey(), READ, TopBarMenu()
Category:	Get system, UI functions
Source:	rtl\getsys.prg, rtl\tgetlist.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

ReadUpdated()

Queries or changes the updated flag of the READ command.

Syntax

```
ReadUpdated( [<lNewFlag>] ) --> lOldFlag
```

Return

The function returns the previous flag as a logical value.

Description

ReadUpdated() is a utility function of the Get system. It maintains a logical value indicating the Updated() status of the READ command. ReadUpdated() returns this status and allows to change it.

Info

See also: @...GET, Get(), READ, Updated()
Category: Get system
Source: rtl\getsys.prg
LIB: xhb.lib
DLL: xhbdll.dll

ReadVar()

Returns name of the current GET or MENU TO variable.

Syntax

```
ReadVar() --> cVarName
```

Return

The function returns the name of the variable in the current Get field or MENU TO command as a character string.

Info

See also:	@...GET, @...PROMPT, MENU TO, READ, SetKey()
Category:	Get system
Source:	rtl\readvar.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

RecCount()

Returns the number of records available in a work area.

Syntax

```
RecCount() --> nRecords
```

Return

The function returns the number of records available in a work area as a numeric value. If the work area is not used or if a database is empty, the return value is zero.

Description

RecCount() is identical with LastRec(). Refer to LastRec() for a description.

Info

See also:	LastRec()
Category:	Database functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

RecNo()

Retrieves the number of the current record in a work area.

Syntax

```
RecNo() --> nRecord
```

Return

The function returns a numeric value which is the record number of the current record in a work area. If no database is open in the work area, RecNo() returns zero.

Description

RecNo() retrieves the record number of the current record in a work area. The record number identifies a database record unambiguously and can be used with the DbGoto() function to navigate the record pointer of a work area to a particular record.

Record numbers can be used for physical navigation in a database, they do not represent the logical order of records in an indexed database.

Info

See also: Bof(), DbGoto(), DbInfo(), DbSkip(), Eof(), GO, LastRec(), OrdKeyGoto(), RecSize(), SKIP
Category: Database functions
Source: rdd\dbcmd.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example outlines the difference between physical and logical
// records.

   REQUEST Dbfcdx

   PROCEDURE Main
      USE Customer VIA "DBFCDX"
      INDEX ON Upper(LastName) TAG NAME TO Cust01

      DbGoto( 1 )                        // Physical first record
      ? Lastname, Recno()                // result: Miller    1

      DbGoto( LastRec() )                // Physical last record
      ? Lastname, Recno()                // result: Smith    22

      OrdKeyGoto( 1 )                    // Logical first record
      ? Lastname, Recno()                // result: Alberts   20

      OrdKeyGoto( LastRec() )            // Logical last record
      ? Lastname, Recno()                // result: Waters    15

      USE
   RETURN
```

RecSize()

Retrieves the number of bytes required to store a database record.

Syntax

```
RecSize() --> nBytes
```

Return

The function returns a numeric value indicating the record length, or number of bytes required to store a record in a database open in a work area. If no database is open in the work area, the return value is zero.

Description

RecSize() is an informational database function used to determine the record length of the database open in a work area. The record length refers to the number of bytes a database file grows when a new record is added to the database with APPEND BLANK.

Info

See also:	DbInfo(), DiskSpace(), FCount(), FieldName(), Header(), LastRec(), RecNo()
Category:	Database functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example implements a user-defined function which calculates
// the file size of a DBF file.

   FUNCTION DbfFileSize()
   RETURN ( (RecSize() * LastRec()) + Header() + 1 )
```

RemAll()

Deletes a specified character from both sides of a string.

Syntax

```
RemAll( <cString>, [<xChar>] ) --> cResult
```

Arguments

`<cString>`

This is the character string to delete characters from on both sides.

`<xChar>`

A single character or its numeric ASCII code can be passed. The default value is a space character (Chr(32)).

Return

The function removes <*xChar*> at the beginning and the end of <*cString*> and returns the modified string. If no character is specified for <*xChar*> the result is the same as AllTrim().

Info

See also:	RangeRem(), RemLeft(), RemRight()
Category:	CT:String manipulation, Character functions
Source:	ct\remove.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays results of RemAll()

   PROCEDURE Main
      LOCAL cStr1 := "  xHarbour  "
      LOCAL cStr2 := "00012345,000"

      ? ">"+RemAll( cStr1 )+"<"        // result: >xHarbour<

      ? ">"+RemAll( cStr2, "0" )+"<"   // result: >12345,<
   RETURN
```

RemLeft()

Deletes a specified character from the left side of a string.

Syntax

```
RemLeft( <cString>, [<xChar>] ) --> cResult
```

Arguments

<cString>

This is the character string to delete characters from on the left side.

<xChar>

A single character or its numeric ASCII code can be passed. The default value is a space character (Chr(32)).

Return

The function removes *<xChar>* on the left side of *<cString>* and returns the modified string. If no character is specified for *<xChar>* the result is the same as LTrim().

Info

See also:	RangeRem(), RemAll(), RemRight()
Category:	CT:String manipulation, Character functions
Source:	ct\remove.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays results of RemLeft()

   PROCEDURE Main
      LOCAL cStr1 := "  xHarbour  "
      LOCAL cStr2 := "00012345,000"

      ? ">"+RemLeft( cStr1 )+"<"        // result: >xHarbour  <

      ? ">"+RemLeft( cStr2, "0" )+"<"   // result: >12345,000<
   RETURN
```

RemRight()

Deletes a specified character from the right side of a string.

Syntax

```
RemRight( <cString>, [<xChar>] ) --> cResult
```

Arguments

<cString>

This is the character string to delete characters from on the right side.

<xChar>

A single character or its numeric ASCII code can be passed. The default value is a space character (Chr(32)).

Return

The function removes *<xChar>* on the right side of *<cString>* and returns the modified string. If no character is specified for *<xChar>* the result is the same as RTrim().

Info

See also: RangeRem(), RemAll(), RemRight()
Category: CT:String manipulation, Character functions
Source: ct\remove.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays results of RemRight()

   PROCEDURE Main
      LOCAL cStr1 := "   xHarbour   "
      LOCAL cStr2 := "00012345,000"

      ? ">"+RemRight( cStr1 )+"<"        // result: >   xHarbour<

      ? ">"+RemRight( cStr2, "0" )+"<"   // result: >00012345,<
   RETURN
```

RenameFile()

Renames a file and handles errors.

Syntax

```
RenameFile( <cOldFile>, <cNewFile> ) --> nErrorCode
```

Arguments

<cOldFile>

This is a character string holding the name of the file to rename. It must include path and file extension. The path can be omitted from *<cOldFile>* when the file resides in the current directory.

<cNewFile>

This is a character string with the new file name including file extension. Drive and/or path are optional.

Return

The function returns zero on success or a numeric error code on failure.

Info

See also: DeleteFile(), FileMove()
Category: CT:DiskUtil, File functions
Source: ct\diskutil.prg
LIB: xhb.lib
DLL: xhbdll.dll

ReplAll()

Replaces a specified character on both sides of a string.

Syntax

```
ReplAll( <cString> , ;
         <xReplace>, ;
        [<xSearch>]  ) --> cResult
```

Arguments

`<cString>`

This is the character string to replace characters on both sides.

`<xReplace>`

This is a single character or its numeric ASCII code that replaces *<xSearch>* on both sides of *<cString>*.

`<xSearch>`

A single character or its numeric ASCII code can be passed. This character is replaced by *<xReplace>*. The default value is a space character (Chr(32)).

Return

The function replaces characters specified with *<xSearch>* at the beginning and end of *<cString>* with *<xReplace>* and returns the modified string.

Info

See also:	ReplLeft(), ReplRight(), StrTran()
Category:	CT:String manipulation, Character functions
Source:	ct\replace.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays results of ReplAll()

   PROCEDURE Main
      LOCAL cStr1 := "  xHarbour  "
      LOCAL cStr2 := "00012345,000"

      ? ">"+ReplAll( cStr1, "_" )+"<"        // result: >__xHarbour__<

      ? ">"+ReplAll( cStr2, "x", "0" )+"<"   // result: >xxx12345,xxx<
   RETURN
```

Replicate()

Creates a character string by replicating an input string.

Syntax

```
Replicate( <cString>, <nCount> ) --> cReplicatedString
```

Arguments

`<cString>`

This parameter is the input string to replicate <*nCount*> times.

`<nCount>`

A numeric value indicating the number or times to replicate <*cString*>.

Return

The function returns a character string holding the input string <*cString*> <*nCount*> number of times.

Description

Replicate() provides a convenient way for creating an output string consisting of multiple copies of an input string.

A similar function is Space() which creates a character string consisting of blank space characters only.

Info

See also:	PadC() | PadL() | PadR(), Space()
Category:	Character functions
Source:	rtl\replic.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows results of Replicate()

   PROCEDURE Main
      ? Replicate( "-", 17 )           // result: -----------------
      ? Replicate( "xHarbour ", 2 )    // result: xHarbour xHarbour
      ? Replicate( "!", 17 )           // result: !!!!!!!!!!!!!!!!!
   RETURN
```

ReplLeft()

Replaces a specified character on the left side of a string.

Syntax

```
ReplLeft( <cString> , ;
          <xReplace>, ;
         [<xSearch>]  ) --> cResult
```

Arguments

<cString>

This is the character string to replace characters on the left side.

<xReplace>

This is a single character or its numeric ASCII code that replaces *<xSearch>* on the left side of *<cString>*.

<xSearch>

A single character or its numeric ASCII code can be passed. This character is replaced by *<xReplace>*. The default value is a space character (Chr(32)).

Return

The function replaces characters specified with *<xSearch>* at the beginning of *<cString>* with *<xReplace>* and returns the modified string.

Info

See also: ReplAll(), ReplRight(), StrTran()
Category: CT:String manipulation, Character functions
Source: ct\replace.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays results of ReplLeft()

   PROCEDURE Main
      LOCAL cStr1 := "  xHarbour  "
      LOCAL cStr2 := "00012345,000"

      ? ">"+ReplLeft( cStr1, "_" )+"<"        // result: >__xHarbour  <

      ? ">"+ReplLeft( cStr2, "x", "0" )+"<"   // result: >xxx12345,000<
   RETURN
```

ReplRight()

Replaces a specified character on the right side of a string.

Syntax

```
ReplRight( <cString> , ;
           <xReplace>, ;
          [<xSearch>]  ) --> cResult
```

Arguments

<cString>

This is the character string to replace characters on the right side.

<xReplace>

This is a single character or its numeric ASCII code that replaces *<xSearch>* on the right side of *<cString>*.

<xSearch>

A single character or its numeric ASCII code can be passed. This character is replaced by *<xReplace>*. The default value is a space character (Chr(32)).

Return

The function replaces characters specified with *<xSearch>* at the end of *<cString>* with *<xReplace>* and returns the modified string.

Info

See also: ReplAll(), ReplRight(), StrTran()
Category: CT:String manipulation, Character functions
Source: ct\replace.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays results of ReplRight()

   PROCEDURE Main
      LOCAL cStr1 := "  xHarbour  "
      LOCAL cStr2 := "00012345,000"

      ? ">"+ReplRight( cStr1, "_" )+"<"        // result: >  xHarbour__<

      ? ">"+ReplRight( cStr2, "x", "0" )+"<"   // result: >00012345,xxx<
   RETURN
```

RestCursor()

Restores shape and position of the screen cursor.

Syntax

```
RestCursor( <nSaved> ) --> cNull
```

Arguments

<nSaved>

This is a numeric value as returned from SaveCursor().

Return

The function restores the cursor position and its shape, and returns a null string ("").

Info

See also:	SaveCursor(), SetCursor(), SetPos()
Category:	CT:Video, Screen functions
Source:	ct\screen3.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

RestGets()

Restores a previously saved *Getlist* array.

Syntax

```
RestGets( <aSavedGets> ) --> lSuccess
```

Arguments

<aSavedGets>

This is an array previously obtained from SaveGets().

Return

The function accepts a previously saved *Getlist* array and assigns it to the PUBLIC variable *Getlist*. The logical return value indicates a successful operation.

Info

See also:	@...GET, SaveGets()
Category:	CT:GetSys, Get system
Source:	ct\ctmisc.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

RestScreen()

Displays a SaveScreen() string.

Syntax

```
RestScreen( [<nTop>]   , ;
            [<nLeft>]  , ;
            [<nBottom>], ;
            [<nRight>] , ;
            <cScreen>    ) --> NIL
```

Arguments

<nTop>

A numeric value indicating the screen coordinate for the top row of the rectangle to display the SaveScreen() string. The default value is 0.

<nLeft>

A numeric values indicating the screen coordinate for the left column of the rectangle to display the SaveScreen() string. The default value is 0.

<nBottom>

A numeric value indicating the screen coordinate for the bottom row of the rectangle to display the SaveScreen() string. The default value is MaxRow().

<nRight>

A numeric values indicating the screen coordinate for the right column of the rectangle to display the SaveScreen() string. The default value is MaxCol().

<cScreen>

A character string previously returned from function SaveScreen().

Return

The function returns NIL.

Description

RestScreen() displays the contents of a console window previously saved with the SaveScreen() function. The coordinates *<nTop>*, *<nLeft>*, *<nBottom>* and *<nRight>* do not necessarily have to be the same coordinates used with SaveScreen(). They must, however, describe a rectangular area of the exact same size. If the rectangle defined for SaveScreen() differs from the rectangle defined for RestScreen(), the result of RestSCreen() is unpredictable.

Info

See also:	DispBegin(), SaveScreen()
Category:	Screen functions
Source:	rtl\saverest.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays a frame moving diagonal in a console window.
// The SaveScreen() string is created only once, while the frame is
// displayed and erased at different screen coordinates.
```

```
PROCEDURE Main
   LOCAL nTop:=1, nLeft:=1, nBottom:=10, nRight:=40, cScreen, i

   CLS
   cScreen := SaveScreen( nTop, nLeft, nBottom, nRight )

   FOR i:=1 TO 20
      DispBox( nTop, nLeft, nBottom, nRight, 2, "W+/B" )

      RestScreen( nTop, nLeft, nBottom, nRight, cScreen )
      Inkey(0.2)
      nTop ++
      nLeft ++
      nBottom ++
      nRight ++
   NEXT

RETURN
```

RestSetKey()

Restores SetKey() settings and associated code blocks.

Syntax

```
RestSetkey( <aSavedSetKey> ) --> lRestored
```

Arguments

<aSavedSetKey>

This is an array previousley returned from SaveSetKey().

Return

The functions scans *<aSavedSetKey>* and restores previously saved SetKey() settings. The returned logical value indicates a successful operation.

Info

See also:	SaveSetKey(), SetKey()
Category:	CT:GetSys, Get system
Source:	ct\setkeys.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

RestToken()

Restores the global environment of the incremental tokenizer.

Syntax

```
RestToken( <cSavedTokenEnv> ) --> cNullString
```

Arguments

<cSavedTokenEnv>

This is the character string previously returned from SaveToken(), or the fourth parameter of TokenInit().

Return

The function assigns a previously saved tokenizer environment to the global tokenizer environment and returns a null string ("").

Note: the tokenizer environment must be saved at runtime. It is not possible to restore the environment from a file.

Info

See also:	SaveToken(), TokenInit(), TokenNext()
Category:	CT:String manipulation, Character functions, Token functions
Source:	ct\token2.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Right()

Extracts characters from the right side of a string

Syntax

```
Right( <cString>, <nCount> ) --> cSubString
```

Arguments

`<cString>`

A character string to extract a substring from.

`<nCount>`

A numeric value specifying the number of characters to extract from the right side of *<cString>*.

Return

The function returns a character string containing Min(*<nCount>*, Len(*<cString>*)) characters.

Description

The character function Right() extracts a substring from the right side of *<cString>*. The returned substring contains *<nCount>* characters. If *<nCount>* is larger than Len(*<cString>*), the return value is a copy of *<cString>*.

The counterpart of Right() is Left(), which extracts a substring from the left side of a string.

Info

See also:	At(), HB_ATokens(), Left(), LTrim(), RAt(), RTrim(), Stuff(), SubStr()
Category:	Character functions
Source:	rtl\right.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example illustrates return values of function Right()

   PROCEDURE Main
      ? CDoW(Date())                  // result: Friday
      ? Right( CDoW(Date()), 3)       // result: day

      ? Time()                        // result: 14:45:12
      ? Right( Time(), 5)             // result: 45:12

      ? Right("xHarbour", 7)           // result: Harbour
   RETURN
```

RLock()

Locks the current record for concurrent write access in a work area.

Syntax

```
RLOCK() --> lIsLocked
```

Return

The function returns .T. (true) if the current record is successfully locked, otherwise .F. (false) is returned.

Description

The network function RLock() locks the current record in a work area for concurrent write access. If the record is successfully locked, it still can be read by other applications, but it can only be changed by the application which has obtained the record lock.

RLock() releases all pending record locks prior to attempting to lock the current record. If this attempt fails, the record is currently locked by another application in a network.

Record locks are required when a database is opened in SHARED mode and data must be written to the database file using the REPLACE command, or when the deleted flagof the current record is changed using DELETE or RECALL.

After the record is changed, the record lock must be released using UNLOCK or DbUnlock().

Info

See also:	APPEND BLANK, DbRLock(), DbRLockList(), DbUnlock(), DbUseArea(), FLock(), SET EXCLUSIVE, UNLOCK, USE
Category:	Database functions, Network functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates a typical coding pattern for updating a
// record in a networked environment.

   PROCEDURE Main
      USE Customer ALIAS Cust SHARED
      INDEX ON Upper( LastName+FirstName) TO Cust01

      SEEK "WALTERS"

      IF Found()
         IF RLock()
            REPLACE FIELD->Lastname WITH "Waters"
            DbUnlock()
         ELSE
            Alert( "Record is currently locked" )
         ENDIF
      ENDIF

      USE
   RETURN
```

Round()

Rounds a numeric value to a specified number of digits

Syntax

```
Round( <nNumber>, <nDecimals> ) --> nRounded
```

Arguments

<nNumber>

This is the numeric value to round.

<nDecimals>

If the parameter is a positive number, it specifies the number of decimal places to retain after the decimal point. If specified as negative value, Round() operates on the digits before the decimal point, thus rounding integer numbers.

Return

The function returns the rounded numeric value.

Description

Round() is a numeric function used to round numbers to a given number of decimal places. Digits 5 to 9 are rounded up, while digits 0 to 4 round down.

Info

See also:	Abs(), Int(), SET DECIMALS, SET FIXED, Str(), Val()
Category:	Numeric functions
Source:	rtl\round.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates results of Round() and how SET FIXED
// influences the display of rounded numbers.

   PROCEDURE Main

      SET DECIMALS TO 4
      SET FIXED ON

      ? Round( 1234.5678, 0)            // result: 1235.0000
      ? Round( 1234.5678, 1)            // result: 1234.6000
      ? Round( 1234.5678, 2)            // result: 1234.5700
      ? Round( 1234.5678, 3)            // result: 1234.5680

      ? Round( 1234.5678,-1)            // result: 1230.0000
      ? Round( 1234.5678,-2)            // result: 1200.0000
      ? Round( 1234.5678,-3)            // result: 1000.0000

      SET FIXED OFF

      ? Round( 1234.5678, 0)            // result: 1235
      ? Round( 1234.5678, 1)            // result: 1234.6
      ? Round( 1234.5678, 2)            // result: 1234.57
      ? Round( 1234.5678, 3)            // result: 1234.568
```

```
   ? Round( 1234.5678,-1)            // result: 1230
   ? Round( 1234.5678,-2)            // result: 1200
   ? Round( 1234.5678,-3)            // result: 1000

RETURN
```

Row()

Returns the current row position of the screen cursor.

Syntax

```
Row() --> nRowPos
```

Return

The function returns a numeric value indicating the current row position of the screen cursor. The top row has position 0 and the bottom position is identified by MaxRow().

Description

The function returns the current cursor row position within a console window (text-mode). The cursor position changes with screen output using console output commands and functions.

Use function SetPos() to position the screen cursor at a defined row and column coordinate.

Info

See also:	?|??, @...GET, @...SAY, Col(), MaxRow(), PCol(), PRow(), SETPOS()
Category:	Screen functions
Source:	rtl\setpos.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The examples displays and changes the screen cursor position.

   PROCEDURE Main
      CLS

      ? Row(), Col()                   // result: 0  0

      ? "xHarbour"

      ? Row(), Col()                   // result: 2  8
   RETURN
```

RtoD()

Converts angles from radians to degrees.

Syntax

```
RtoD( <nRadians> ) --> nDegrees
```

Arguments

<nRadians>

A numeric value specifies the radians as a fraction (or multiple) of Pi().

Return

The numeric return value is the angle in degrees (one full circle = 360 degrees).

Info

See also: DtoR()
Category: CT:Math, Mathematical functions, Trigonometric functions
Source: ct\trig.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// the example displays fractions of Pi in degrees
   PROCEDURE Main
      ? RtoD(   0      )   // result:    0.00
      ? RtoD(  Pi()/8 )   // result:   22.50
      ? RtoD(  Pi()/6 )   // result:   30.00
      ? RtoD(  Pi()/4 )   // result:   45.00
      ? RtoD(  Pi()/2 )   // result:   90.00
      ? RtoD(  Pi()   )   // result:  180.00
   RETURN
```

RTrim()

Removes trailing blank spaces from a character string.

Syntax

```
RTrim( <cString> [,<lAllWhiteSpace>] ) --> cTrimmedString
Trim( <cString> [,<lAllWhiteSpace>] ) --> cTrimmedString
```

Arguments

`<cString>`

A character string which is copied without trailing blank space characters.

`<lAllWhiteSpace>`

This parameter defaults to .F. (false). If .T. (true) is passed, function RTrim() treats the white-space characters TAB (Chr(9)) and CRLF (Chr(13)+Chr(10)) like blank spaces and removes them as well.

Return

The function returns a copy of *<cString>* without blank spaces at the end of the string.

Description

RTrim() is used for formatting character strings whose ending characters consist of blank spaces (Chr(32)). The function creates a copy of *<cString>* but ignores blank spaces at the end of the input string. It is frequently used to format character strings stored in database fields. Such strings are padded with blank paces up to the field length.

Note() function Trim() is a synonym for RTrim().

Info

See also: Alltrim(), LTrim(), PadC() | PadL() | PadR(), SubStr()
Category: Character functions
Source: rtl\trim.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example shows various possibilities for trimming a string.

   #define CRLF     Chr(13)+Chr(10)

   PROCEDURE Main
      LOCAL cStr  := "  xHarbour    "

      ? Len( cStr )                    // result:  14
      ? Len( LTrim( cStr ) )           // result:  12
      ? Len( RTrim( cStr ) )           // result:  10
      ? Len( AllTrim( cStr ) )         // result:   8

      cStr := "xHarbour  " + CRLF

      ? Len( cStr )                    // result:  12
      ? Len( RTrim( cStr ) )           // result:  12
      ? Len( RTrim( cStr, .T. ) )      // result:   8

   RETURN
```

SaveCursor()

Saves the current cursor shape and position.

Syntax

```
SaveCursor() --> nSaved
```

Return

The function returns a numeric value. It can be passed to RestCursor() for restoring the current cursor settings.

Info

See also: RestCursor(), SetCursor(), SetPos()
Category: CT:Video, Screen functions
Source: ct\screen3.prg
LIB: xhb.lib
DLL: xhbdll.dll

SaveGets()

Saves the current *Getlist* array for nested READs.

Syntax

```
SaveGets() --> aSavedGetlist
```

Return

The function returns the current *Getlist* array and assigns a new, empty array to the PUBLIC *Getlist* variable. This allows for defining a new *Getlist* via @...GET while a READ command is executed.

Info

See also:	@...GET, READ, RestGets()
Category:	CT:GetSys, Get system
Source:	ct\ctmisc.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

SaveScreen()

Saves a rectangular screen region for later display.

Syntax

```
SaveScreen( [<nTop>], [<nLeft>], [<nBottom>], [<nRight>] ) --> cScreen
```

Arguments

<nTop>

A numeric value indicating the screen coordinate for the top row of the rectangular screen region to save. The default value is 0.

<nLeft>

A numeric values indicating the screen coordinate for the left column of the rectangular screen region to save. The default value is 0.

<nBottom>

A numeric value indicating the screen coordinate for the bottom row of the rectangular screen region to save. The default value is MaxRow().

<nRight>

A numeric values indicating the screen coordinate for the right column of the rectangular screen region to save. The default value is MaxCol().

Return

The function returns a character string holding textual and color information of the saved screen rectangle.

Description

SaveScreen() is used to save the current display in a console window. The function combines textual and color information in the returned string. This string can later be displayed using the RestScreen() function.

Note: textual information on screen is placed at odd positions in the return string, while color attributes for each character are stored at even positions in the SaveScreen() string.

Info

See also: RestScreen()
Category: Screen functions
Source: rtl\saverest.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example saves a screen region, changes the display of that
// region, and restores it to its original state.

   PROCEDURE Main
      LOCAL nTop:=0, nLeft:=0, nBottom:=10, nRight:=40, cScreen

      cScreen := SaveScreen( nTop, nLeft, nBottom, nRight )
      SET COLOR TO "W+/B"
      DispBox( nTop, nLeft, nBottom, nRight )
```

```
   WAIT "Press a key..."

   RestScreen( nTop, nLeft, nBottom, nRight, cScreen )

   WAIT "Screen is restored Press a key..."
RETURN
```

SaveSetKey()

Saves SetKey() settings and associated code blocks.

Syntax

```
SaveSetKey() --> aSavedSetKeys
```

Return

The function returns an array holding the current SetKey() settings. These settings can be restored later using function RestSetKey().

Info

See also:	RestSetKey(), SaveGets(), SetKey()
Category:	CT:GetSys, Get system
Source:	ct\setkeys.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

SaveToken()

Saves the global environment of the incremental tokenizer.

Syntax

```
SaveToken() --> cGlobalTokenEnv
```

Return

The function returns the global environment of the tokenizer as a character string.

Description

SaveToken() is required when nested calls to TokenInit() appear in a program. Before the incremental tokenizer is initialized with a new global environment, the current environment of the tokenizer is saved with SaveToken(). Then, a call to TokenInit() prepares the tokenizer with a new global environment. When the new string is processed, the previous global tokenizer environment is restored via RestToken() and the tokenizer can resume processing the previous string.

Info

See also:	RestToken(), TokenInit(), TokenNext()
Category:	CT:String manipulation, Character functions, Token functions
Source:	ct\token2.c
LIB:	xhb.lib
DLL:	xhbdll.dll

SayDown()

Outputs a string vertically to the bottom of the screen.

Syntax

```
SayDown( <cString>, ;
        [<nDelay>], ;
        [<nRow>]  , ;
        [<nCol>]    ) --> cNull
```

Arguments

<cString>

A character string being displayed vertically.

<nDelay>

This is a numeric value specifying the number of milliseconds the function waits for the next incremental display. It defaults to 4 milliseconds.

<nRow>

A numeric value indicating the screen row for display. It defaults to Row().

<nCol>

A numeric value indicating the screen column for display. It defaults to Col().

Return

The return value is always a null string ("").

Note: the function leaves the cursor position unchanged.

Info

See also:	SayMoveIn(), SaySpread()
Category:	CT:Video, Screen functions
Source:	ct\screen3.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates the output effect of SayDown().

   PROCEDURE Main
      CLS

      SayDown( "xHarbour compiler", 10, 2, 40 )
   RETURN
```

SayMoveIn()

Outputs a string on the screen using a "move in" effect.

Syntax

```
SayMoveIn( <cString>
          [<nDelay>], ;
          [<nRow>]  , ;
          [<nCol>]  , ;
          [<lRight>]  ) --> cNull
```

Arguments

`<cString>`

A character string being displayed incrementally in the specified row.

`<nDelay>`

This is a numeric value specifying the number of milliseconds the function waits for the next incremental display. It defaults to 4 milliseconds.

`<nRow>`

A numeric value indicating the screen row for display. It defaults to Row().

`<nCol>`

A numeric value indicating the screen column for display. It defaults to Col().

`<lRight>`

If .T. (true) is passed, the string is displayed incrementally from the right. The default is .F. (false), which displays *<cString>* from the left.

Return

The return value is always a null string ("").

Note: the function leaves the cursor position unchanged.

Info

See also:	SaveCursor(), SayDown(), SaySpread()
Category:	CT:Video, Screen functions
Source:	ct\screen3.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates the two output effects of SayMoveIn().

   PROCEDURE Main
      CLS

      SayMoveIn( "xHarbour compiler", 10, 2, 20, .F. )

      SayMoveIn( "xHarbour compiler", 10, 4, 20, .T. )
   RETURN
```

SayScreen()

Displays a string on screen keeping existing color attributes.

Syntax

```
SayScreen( <cString>, ;
          [<nRow>]   , ;
          [<nCol>]     ) --> cNull
```

Arguments

<cString>

A character string being displayed at the specified position.

<nRow>

A numeric value indicating the screen row for display. It defaults to Row().

<nCol>

A numeric value indicating the screen column for display. It defaults to Col().

Return

The return value is always a null string ("").

Note: the function leaves the cursor position unchanged.

Info

See also: ColorWin(), SaveScreen()
Category: CT:Video, Screen functions
Source: ct\screen2.prg
LIB: xhb.lib
DLL: xhbdll.dll

SaySpread()

Outputs a string on the screen using a "spread" effect.

Syntax

```
SaySpread( <cString> ,[<nMilliSecondsDelay>],[<nRow>],[<nCol>]) --> cNull
```

Arguments

`<cString>`

A character string being displayed using a "spread" effect.

`<nDelay>`

This is a numeric value specifying the number of milliseconds the function waits for the next incremental display. It defaults to 4 milliseconds.

`<nRow>`

A numeric value indicating the screen row for display. It defaults to Row().

`<nCol>`

A numeric value indicating the screen column for display. It defaults to Col().

Return

The return value is always a null string ("").

Note: the function leaves the cursor position unchanged.

Info

See also:	SayDown(), SayMoveIn()
Category:	CT:Video, Screen functions
Source:	ct\screen3.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates the output effect of SaySpread().

   PROCEDURE Main
      CLS

      SaySpread( "xHarbour compiler", 50, 2, 30 )
   RETURN
```

ScreenAttr()

Returns the numeric color attribute for a specified coordinate on the screen.

Syntax

```
ScreenAttr( [<nRow>], [<nCol>] ) --> nAttribute
```

Arguments

<nRow>

A numeric value indicating the screen row to query. It defaults to Row().

<nCol>

A numeric value indicating the screen column to query. It defaults to Col().

Return

The function returns the color attribute of the specified screen coordinate as a numeric value.

Info

See also:	InvertAttr(), NtoColor()
Category:	CT:Video, Screen functions
Source:	ct\screen1.c
LIB:	xhb.lib
DLL:	xhbdll.dll

ScreenFile()

Writes the contents of the current screen to a file.

Syntax

```
ScreenFile( <cFileName>, ;
           [<lAppend>] , ;
           [<nOffSet>] , ;
           [<lTruncate>] ) --> nBytesWritten
```

Arguments

<cFileName>

This is a character string holding the name of the file to save a screen to. It must include path and file extension. If the path is omitted from *<cFileName>*, the file is searched in the current directory. If the file does not exist, it is created.

<lAppend>

If .T. (true) is passed, the current screen is appended to *<cFileName>*. The default value is .F. (false), i.e. the file *<cFileName>* is overwritten.

<nOffSet>

If *<lAppend>* is set to .T. (true), this numeric parameter defines the file offset, or starting posiion, where the current screen contents are stored. The default is the end-of-file.

<lTruncate>

When *<lTruncate>* is set to .T. (true) and the file pointer is not at the end-of-file after writing, the file size is reduced to the current file pointer position. The default value is .F. (false), which leaves the file size unchanged. The parameter is only required when multiple screens are stored in one file.

Return

The function writes the contents of the current screen to *<cFileName>* and returns the number of bytes written as a numeric value.

Info

See also:	FileScreen(), SaveScreen(), RestScreen()
Category:	CT:Video, Screen functions
Source:	ct\strfile.c
LIB:	xhb.lib
DLL:	xhbdll.dll

ScreenMark()

Searches strings on the screen and changes their color.

Syntax

```
ScreenMark( <cSearch>, ;
            <xColor> , ;
            <lCase>  , ;
           [<lAll>]  , ;
           [<cPre>]  , ;
           [<cPost>]   ) --> lFound
```

Arguments

<cSearch>

This is a character string that is searched for on the screen.

<xColor>

This is either a single color value (see SetColor()), or its numeric color attribute (see ColorToN()). It defines the color for marking the search string.

<lCase>

A logical value must be passed. When *<lCase>* is .T. (true), the function performs a case sensitive search. With .F. (false), the search is case insensitive.

<lAll>

This parameter defaults to .F. (false) so that only the first occurance of *<cSearch>* is color marked. Passing .T. (true) marks all occurrances.

<cPre> and <cPost>

Optionally, delimiting characters for whole words can be specified. For example, the character sequence "is" appears two times in "This is". Using a prefix and postfix delimiter assures that only the word "is" is marked.

Return

The function returns .T. (true) when at lease one occurrance of the search string is color marked. Otherwise, .F. (false) is returned.

Note: the function leaves the cursor position unchanged.

Info

See also: ClearEol(), ColorWin()
Category: CT:Video, Screen functions
Source: ct\scrmark.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays a string and marks words in it

   PROCEDURE Main
      LOCAL cDelim := Chr( GetClearB() )
      CLS

      @ 10,10 SAY "This is the xHarbour compiler"
```

```
   ScreenMark( "is", "W+/R", .F., .T., cDelim, cDelim )

   ScreenMark( "xHarbour", "W+/B", .F. )
RETURN
```

ScreenMix()

Mixes a character string with color attributes.

Syntax

```
ScreenMix( <cString>    , ;
           <cColorAttr>, ;
          [<nRow>]     , ;
          [<nCol>]       ) --> cNull
```

Arguments

<cString>

This is a character string to display on the sccreen.

<cColorAttr>

A character string holding color attributes for each individual character of *<cString>*.

<nRow>

A numeric value indicating the screen row for display. It defaults to Row().

<nCol>

A numeric value indicating the screen column for display. It defaults to Col().

Return

The return value is always a null string ("").

Note: the function leaves the cursor position unchanged.

Info

See also:	ColorToN(), SayScreen()
Category:	CT:Video, Screen functions
Source:	ct\screen2.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays a word using a different color for
// each letter.

   PROCEDURE Main
      LOCAL cWord := "xHarbour"
      LOCAL cAttr := ""
      LOCAL i

      CLS

      FOR i:=1 TO Len( cWord )
         cAttr += Chr(i)
      NEXT

      ScreenMix( cWord, cAttr, 10, 5 )
   RETURN
```

ScreenStr()

Returns the screen contents beginning at a specified position.

Syntax

```
ScreenStr( [<nRow>], [<nCol>], [<nCount>] ) --> cScreenString
```

Arguments

<nRow>

A numeric value indicating the screen row to read from. It defaults to Row().

<nCol>

A numeric value indicating the screen column to read from. It defaults to Col().

<nCount>

Optionally, the number of characters to read from the screen can be specified as a numeric value. If omitted, the function reads all characters beginning at *<nRow>* and *<nCol>* to the end of the screen.

Return

The function returns a character string holding the screen contents and color attributes. It can be redisplayed using StrScreen().

Info

See also:	CharEven(), CharOdd(), SaveScreen(), StrScreen()
Category:	CT:Video, Screen functions
Source:	ct\screen3.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Scroll()

Scrolls a screen region horizontally and/or vertically.

Syntax

```
Scroll( [<nTop>]   , ;
        [<nLeft>]  , ;
        [<nBottom>], ;
        [<nRight>] , ;
        [<nRows>]  , ;
        [<nColumns>] ) --> NIL
```

Arguments

<nTop>

A numeric value indicating the screen coordinate for the top row of the rectangular screen region to scroll. The default value is 0.

<nLeft>

A numeric values indicating the screen coordinate for the left column of the rectangular screen region to scroll. The default value is 0.

<nBottom>

A numeric value indicating the screen coordinate for the bottom row of the rectangular screen region to scroll. The default value is MaxRow().

<nRight>

A numeric values indicating the screen coordinate for the right column of the rectangular screen region to scroll. The default value is MaxCol().

<nRows>

This is a numeric value indicating the number of rows to scroll the screen region in vertical direction. A positive value scrolls left, while negative values scroll right. The default value is zero, which does not scroll vertically.

<nColumns>

This is a numeric value indicating the number of columns to scroll the screen region in horizontal direction. A positive value scrolls up, while negative values scroll down. The default value is zero, which does not scroll horizontally.

Return

The return value is NIL.

Description

Scroll() manipulates the screen display in a console window by moving a rectangular area in horizontal and/or vertical direction. This is accomplished by deleting a row and/or column in scroll direction, shifting the screen area and displaying a blank row and/or column at the end using the standard color. This is repeated until the number of rows and/or columns specified is reached.

Note: if NIL is passed for both, *<nRows>* and *<nColumns>*, the entire rectangle defined with *<nTop>*, *<nLeft>*, *<nBottom>* and *<nRight>* is blanked.

Info

See also: @...BOX, @...CLEAR, @...TO, CLEAR SCREEN, RestScreen(), ScrollFixed()
Category: Screen functions
Source: rtl\scroll.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays numbers 0 to 50 in a screen area that has only
// ten rows. When the screen cursor hits the bottom of the area, the
// display is scrolled up.

   PROCEDURE Main
      LOCAL nTop:= 10, nLeft:=10, nBottom:=20, nRight:=20, cScreen
      LOCAL nRow, nCol, n

      CLS
      SET COLOR TO W+/B
      DispBox( nTop-1, nLeft-1, nBottom+1, nRight+1 )

      SET COLOR TO W+/R

      nRow := Row()
      nCol := Col()

      FOR n := 0 TO 50
         @ nRow, nCol SAY n
         Inkey(0.1)
         nRow ++

         IF nRow > nBottom
            Scroll( nTop, nLeft, nBottom, nRight, 1 )
            nRow := nBottom
         ENDIF
      NEXT

   RETURN
```

Scrollfixed()

Scrolls a screen region horizontally and/or vertically.

Syntax

```
ScrollFixed( [<nTop>]    , ;
             [<nLeft>]   , ;
             [<nBottom>], ;
             [<nRight>] , ;
             [<nRows>]   , ;
             [<nColumns>] ) --> NIL
```

Arguments

<nTop>

A numeric value indicating the screen coordinate for the top row of the rectangular screen region to scroll. The default value is 0.

<nLeft>

A numeric values indicating the screen coordinate for the left column of the rectangular screen region to scroll. The default value is 0.

<nBottom>

A numeric value indicating the screen coordinate for the bottom row of the rectangular screen region to scroll. The default value is 0.

<nRight>

A numeric values indicating the screen coordinate for the right column of the rectangular screen region to scroll. The default value is 0.

<nRows>

This is a numeric value indicating the number of rows to scroll the screen region in vertical direction. A positive value scrolls left, while negative values scroll right. The default value is zero, which does not scroll vertically.

<nColumns>

This is a numeric value indicating the number of columns to scroll the screen region in horizontal direction. A positive value scrolls up, while negative values scroll down. The default value is zero, which does not scroll horizontally.

Return

The return value is NIL.

Description

ScrollFixed() works exactly like the Scroll() function. The only difference is that all parameters default to zero, so that the entire screen remains unchanged when the function is called without parameters.

Info

See also: @...BOX, @...CLEAR, @...TO, CLEAR SCREEN, RestScreen(), SaveScreen(), Scroll()
Category: Screen functions, xHarbour extensions
Source: rtl\scroll.c
LIB: xhb.lib
DLL: xhbdll.dll

Seconds()

Returns the number of seconds elapsed since midnight

Syntax

```
Seconds() --> nSeconds
```

Return

The function returns a numeric value indicating the seconds elapsed since midnight with a granularity of 1/100th of a second.

Description

Seconds() is used to calculate time spans in seconds that have elapsed since a start time. Note that the return value is reset to zero when midnight passes.

Info

See also:	ElapTime(), Days(), Time(), SecondsCpu(), Secs()
Category:	Date and time
Source:	rtl\seconds.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows a typical usage scenario for Seconds()

   PROCEDURE Main
      LOCAL nStart := Seconds()

      Inkey(1.5)     // some lengthy operation

      ? "Elapsed:", Seconds() - nStart
   RETURN
```

SecondsCpu()

Returns the CPU time used by the current process.

Syntax

```
SecondsCpu( [<nWhichTime>] ) --> nSeconds
```

Arguments

<nWhichTime>

This is a numeric value used to determine the CPU usage time of the current process. The following values are supported:

Values for *<nWhichTime>*

Value	Description
1	User CPU time of the current process
2	System CPU time on behalf of the current process
3	Sum of 1 and 2 (default)
11	Sum of the user CPU time of the current + child process
12	Sum of the system CPU time of the current + child process
13	Sum of 11 and 12

Return

The function returns the requested usage time in seconds as a numeric value.

Info

See also:	HB_Clocks2Secs(), ElapTime(), Seconds()
Category:	Date and time, xHarbour extensions
Source:	rtl\seconds.c
LIB:	xhb.lib
DLL:	xhbdll.dll

SecondsSleep()

Suspends thread execution for a number of seconds.

Syntax

```
SecondsSleep( <nSeconds> ) --> NIL
```

Arguments

<nSeconds>

This is a numeric value indicating the number of seconds to pause thread execution. The granularity is three places after the decimal point (1/1000th of a second).

Return

The function returns always NIL.

Description

Function SecondsSleep() suspends program execution in the current thread for the period of *<nSeconds>* seconds. During this period of time, no CPU resources are consumed by the current thread. The function accepts a numeric value with up to 3 decimal places (1/1000th of a second).

Note: this function is also available in single threaded applications.

Info

See also:	Inkey(), StartThread(), ThreadSleep(), WAIT
Category:	Multi-threading functions, xHarbour extensions
Source:	vm\thread.c
LIB:	xhbmt.lib
DLL:	xhbmtdll.dll

Secs()

Calculates the number of seconds from a time string.

Syntax

```
Secs( <cTime>|<dDateTime> ) --> nSeconds
```

Arguments

<cTime>

This is a character string formatted as hh:mm:ss. It holds a time string based on a 24h clock.

<dDateTime>

Instead of a time string, the function accepts a DateTime() value.

Return

When a time string is passed, the function returns a numeric value representing the number of seconds corresponding to the input string.

When a DateTime value is passed, the function extracts the number of seconds of the DateTime value (:ss.ccc).

Description

Secs() is a time conversion function that accepts a Time() compliant character string and calculates from it the corresponding number of seconds. The reverse function of Secs() is TSTring().

Alternatively, the Seconds part of a DatTime value is returned

Info

See also:	DateTime(), ElapTime(), Hour(), Minute(), Seconds(), Time(), TString()
Category:	Date and time
Source:	rtl\samples.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates return values of Secs()

   PROCEDURE Main
      LOCAL dDateTime := {^ 2007/04/26 18:31:59.789 }

      ? Secs( "23:59:59" )     // result:    86399
      ? Secs( "00:00:00" )     // result:        0
      ? Secs( "12:00:00" )     // result:    43200
      ? Time()                 // result: 16:07:34
      ? Secs( Time() )         // result:    58054

      ? Secs( dDateTime )      // result: 59.789
   RETURN
```

SecToTime()

Converts numeric seconds into a time formatted character string.

Syntax

```
SecToTime( [<nSeconds>], [<lHundredth>] ) --> cTime
```

Arguments

<cTime>

This is a numeric value specifying the number of seconds elapsed since midnight. It defaults to Seconds().

<lHundredth>

If set to .T. (true), the returned character string includes 1/100ths of seconds. The default value is .F. (false).

Return

The function returns a character string containing the number of seconds formatted as HH:MM:SS[:ss].

Info

See also: Seconds(), SecToTime(), Time()
Category: CT:DateTime, Date and time
Source: ct\cttime.prg
LIB: xhb.lib
DLL: xhbdll.dll

Select()

Retrieves the work area number by alias name.

Syntax

```
Select( [<cAlias>] ) --> nWorkArea
```

Arguments

<cAlias>

A character string holding the alias name of the work area to select as current.

Return

The function returns an integer numeric value representing the work area number of the work area having the alias name *<cAlias>*. If this alias does not exist, the return value is zero. When *<cAlias>* is omitted, the function returns the work area number of the current work area.

Description

Select() retrieves the work area number by its alias name. This is the opposite of function Alias(), which retrieves the alias name of a work area by its number.

Work areas are numbered from 1 to 65535. They hold open database and index files and make them accesible to database functions and commands.

Info

See also: Alias(), DbSelectArea(), FieldGet(), SELECT, USE, Used()
Category: Database functions
Source: rdd\dbcmd.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example opens two databases in two different work areas
// shows their work area numbers.

   PROCEDURE Main
      USE Customer ALIAS Cust

      DbSelectArea(10)
      USE Invoice ALIAS Inv

      ? Select()                     // result: 10
      ? Select( "Cust" )             // result:  1
      ? Select( "Inv" )              // result: 10

      CLOSE ALL
   RETURN
```

Set()

Retrieves or changes a system setting.

Syntax

```
Set( <nDefine>, [<xNewSetting>], [<xOption>] ) --> xOldSetting
```

Arguments

`<nDefine>`

This is a numeric parameter for which #define constants exist in the file Set.ch. They identify the numerous system settings that can be queried or changed.

`<xNewSetting>`

<xNewSetting> is an optional argument specifying a new value for the system setting identified by *<nDefine>*. The data type for *<xNewSetting>* depends on the system setting to change (see below).

`<xOption>`

Some system settings require a second parameter. This is what parameter *<xOption>* is used for.

Return

The function returns the value of the specified system setting which is set before the function is called.

Description

Set() is a universal function managing numerous system settings available in xHarbour. Many of these settings can be changed via SET commands, which is the recommended way. To query the current value of a system setting, however, must be done by calling the Set() function.

This function accepts as first parameter a #define constant, which identifies the system setting to query or change. The available constants are listed in alphabetical order below:

_SET_ALTERNATE	*<lOnOff>* | on | OFF **See also:** command SET ALTERNATE. If the setting is enabled, functions QOut() | QQout() write to the screen and to a file, provided that a file is opened or created with _SET_ALTFILE. If disabled, which is the default, Qout() and QQout() only write to the screen (and/or to the PRINTFILE).
_SET_ALTFILE	*<cFileName>* , *<lAdditive>* **See also:** command SET ALTERNATE. When set, creates or opens file to write Qout() and QQout() output to. If .T. (true) is passed for *<lAdditive>* (=third parameter of Set()) and the file already exists, the file is opened and positioned at the end of file, so that new output is appended to the existing file. Otherwise, the file is created. If a file is already opened, it is closed before the new file is opened or created (even if it is the same file). The default file extension is ".txt". There is no default file name. Call with an empty string to close the file.
_SET_AUTOPEN	*<lOnOff>* | ON | off **See also:** command SET AUTOPEN. When set, opens a structural index file automatically with the USE command, when the index file has the same filename (without extension) as the database file. Default is enabled. Note: this setting must be supported by the replaceble database driver. DBFCDX supports this setting, for example. It opens the structural index file. To automatically select an index of that file, _SET_AUTORDER must be set.

_SET_AUTORDER	<*nOrdPos*> **See also:** command SET AUTORDER. When set, and _SET_AUTOPEN is set to .T. (true), the RDD selects automatically the index at position <*nOrdPos*> as current index. The default is 0, i.e. no index is selected as current, although the index file may be open. Note: this setting must be supported by the replaceble database driver. DBFCDX supports this setting, for example.
_SET_AUTOSHARE	<*nShareMode*> **See also:** command SET AUTOSHARE. The default value is zero, causing an application not to use automatic SHARE mode detection when databases are opened. The value 1 means, the application detects if it runs in a network. If it does not, all database access is done EXCLUSIVE. If <*nShareMode*> is set to 2, databases are always opened in EXCLUSIVE mode.
_SET_BACKGROUNDTASKS	<*lOnOff*> | on | OFF **See also:** command SET BACKGROUND TASKS. When enabled, background tasks are active. When disabled, which is the default, tasks are inactive.
_SET_BACKGROUNDTICK	<*nInterval*> **See also:** command SET BACKGROUNDTICK. This setting is the number of xHarbour pCodes to execute before background tasks are checked to run. The default value for <*nInterval*> is 1000. When this value is enlarged, the system saves time for task checking. Reducing the value improves the response time for background tasks.
_SET_BELL	<*lOnOff*> | on | OFF **See also:** command SET BELL. When enabled, the bell sounds when the last position of a GET entry field is reached and/or when a GET validation fails. Disabled by default.
_SET_CANCEL	<*lOnOff*> **See also:** function SetCancel(). When enabled, which is the default, pressing Alt+C or Ctrl+Break terminates the program. When disabled, both keystrokes can be read by Inkey(). Note: _SET_KEY has precedence over _SET_CANCEL.
_SET_COLOR	<*cColorString*> **See also:** function SetColor(). This setting exists for compatibility reasons. It is superseeded by function SetColor().
_SET_CONFIRM	<*lOnOff*> | on | OFF **See also:** command SET CONFIRM. If enabled, an exit key must be pressed to leave a GET. If disabled, which is the default, typing past the end will leave a GET.
_SET_CONSOLE	<*lOnOff*> | ON | off **See also:** command SET CONSOLE. If enabled, which is the default, all screen output goes to the console window. When disabled, screen output is suppressed.
_SET_CURSOR	<*nCursorType*> **See also:** function SetCursor(). <*nCursorType*> is a #define constant from the SetCurs.ch file. It defines the shape of the cursor in a console application. The default is SC_NORMAL.
_SET_DATEFORMAT	<*cDateFormat*> | AMERICAN | ansi | British | French | German | Italian | Japan | USA **See also:** command SET DATE. <*cDateFormat*> is a character string specifying the date format for displaying Date values.
_SET_DBFLOCKSCHEME	<*nLockScheme*> Refer to command SET DBFLOCKSCHEME for a comprehensive description of this setting.
_SET_DEBUG	<*lOnOff*> When set to .T., pressing Alt+D activates the debugger. When set to .F., which is the default, Alt+D can be read by Inkey(). This setting is also affected by AltD(1) and AltD(0).
_SET_DECIMALS	<*nDecimals*> **See also:** command SET DECIMALS. Sets the number of decimal digits to use for displaying or printing numeric values when

	SET FIXED is ON. Defaults to 2. If SET FIXED is OFF, then SET DECIMALS is only used to determine the number of decimal digits to use after using Exp(), Log(), SQrt(), or division. Other math operations may adjust the number of decimal digits that the result will display. Note: This never affects the precision of a number. Only the display format is affected.
_SET_DEFAULT	<*cPath*> **See also:** command SET DEFAULT. Sets the default directory in which to open, create and check for database and index files. Defaults to the current directory (empty string).
_SET_DELETED	<*lOnOff*> | on | OFF **See also:** command SET DELETED. If enabled, records marked for deletion are not visible. If disabled, which is the default, deleted records are visible.
_SET_DELIMCHARS	<*cDelimiters*> "::" **See also:** command SET DELIMITERS. Sets the GET delimiter characters. Defaults to "::".
_SET_DELIMITERS	<*lOnOff*> | on | OFF **See also:** command SET DELIMITERS. If enabled, GETs are delimited on screen. If disabled, which is the default, no GET delimiters are used.
_SET_DEVICE	SCREEN | print **See also:** command SET DEVICE. Selects the output device for DEVOUT(). When set to "PRINTER", all output is sent to the printer device or file set by _SET_PRINTFILE. When set to anything else, all output is sent to the screen. Defaults to "SCREEN".
_SET_DIRCASE	<*nCaseMode*> **See also:** command SET DIRCASE. This setting controls how the directory name will be acessed on disk. If 0 is specified, which is the default, mixed case letters are allowed. If 1 is speficied, lower case letters are used, also, converts all letters to lower case. If 2 is speficied, upper case letter are used, also, converts all letters to upper case.
_SET_DIRSEPARATOR	<*cDirSeparator*> **See also:** command SET DIRSEPARATOR. This setting specifies a single character to be used as path separator. Default is the backslash "\".
_SET_EOL	<*cEndOfLineChars*> **See also:** command SET EOL. This setting specifies the end-of-line character(s) used by an xHarbour application. It defaults to Crrariage Return plus Line Feed (Chr(13)+Chr(10)).
_SET_EPOCH	<*nYear*> (1900) **See also:** command SET EPOCH. Determines how to handle the conversion of 2-digit years to 4 digit years. When a 2-digit year is greater than or equal to the year part of the epoch, the century part of the epoch is added to the year. When a 2-digit year is less than the year part of the epoch, the century part of the epoch is incremented and added to the year. The default epoch is 1900, which converts all 2-digit years to 19xx. Example: If the epoch is set to 1950, 2-digit years in the range from 50 to 99 get converted to 19xx and 2-digit years in the range 00 to 49 get converted to 20xx.
_SET_ERRORLOG	<*cFileName*>, <*lAppend*> **See also:** command SET ERRORLOG.
_SET_ERRORLOOP	<*nRecursion*> **See also:** command SET ERRORLOOP. This setting defines the recursion depth if recursive errors occur in the xHarbour error handling routine. <*nRecursion*> is a numeric value that defines the maximume error recursion level after which an xHarbour application terminates. The default value is 8.
_SET_ESCAPE	<*lOnOff*> | ON | off **See also:** command SET ESCAPE. When enabled, which is the default, pressing Esc will exit a READ. When

disabled, pressing Esc during a READ is ignored, unless the Esc key has been assigned to a function using SetKey().

_SET_EVENTMASK

<*nEvents*> **See also:** command SET EVENTMASK. Determines which events function Inkey() responds to. #define constants must be used for <*nEvents*>. They are listed in the file INKEY.CH.

_SET_EXACT

<*lOnOff*> | on | OFF **See also:** command SET EXACT. When enabled, all string comparisons other than exactly equal (==) exclude trailing spaces when checking for equality. When disabled, which is the default, all string comparisons other than exactly equal treat two strings as equal if the right hand string is "" or if the right hand string is shorter than or the same length as the left hand string and all of the characters in the right hand string match the corresponding characters in the left hand string.

_SET_EXCLUSIVE

<*lOnOff*> | on | OFF **See also:** command SET EXCLUSIVE. When enabled, which is the default, all database files are opened in exclusive mode. When disabled, all database files are opened in shared mode. Note: The EXCLUSIVE and SHARED clauses of the USE override this setting.

_SET_EXIT

<*lOnOff*> **See also:** function ReadExit(). Toggles the use of Up-arrow and Down-arrow as READ exit keys. Specifying true (.T.) enables them as exit keys, and false (.F.) disables them. Used internally by the ReadExit() function.

_SET_EXTRAFILE

<*cFileName*> When set, creates or opens a secondary SET ALTERNATE file that receives QOut() and QQOut() output.

_SET_FILECASE

<*nCaseMatch*> **See also:** command SET FILECASE. This setting controls how the file will be acessed on disk. If 0 is specified, mixed case letters are allowed. This is the default setting. If 1 is speficied, lower case letters are used, also, convert all letters to lower case. If 2 is speficied, upper case letters are used, also, convert all letters to upper case.

_SET_FIXED

<*lOnOff*> | on | OFF **See also:** command SET FIXED. When enabled, all numeric values will be displayed and printed with the number of decimal digits set by SET DECIMALS, unless a PICTURE clause is used. When disabled, which is the default, the number of decimal digits that are displayed depends upon a variety of factors. See _SET_DECIMALS for more.

_SET_HARDCOMMIT

<*lOnOff*> | on | OFF **See also:** command SET HARDCOMMIT. When enabled, forces an immediate disk write operation for database and index files when a record is changed. When disabled,which is the default, changes are cached in internal database buffers which are flushed later to a file.

_SET_IDLEREPEAT

<*lOnOff*> When enabled, which is the default, idle tasks are repeatedly executed in the background when the application enters an idle state. When disabled, the list of idle tasks is executed only once when the application enters an idle state.

_SET_INSERT

<*lOnOff*> When enabled, characters typed in a GET or MemoEdit() are inserted. When disabled, which is the default, characters typed in a GET or MemoEdit() overwrite. Note: This setting can also be toggled between on and off by pressing the Insert key during a GET or MemoEdit().

_SET_INTENSITY	*<lOnOff>* | ON | off When enabled, which is the default, GETs and PROMPTs are displayed using the enhanced color setting. When disabled, GETs and PROMPTs are displayed using the standard color setting.
_SET_LANGUAGE	*<cLangID>* **See also:** function HB_LangSelect(). Specifies the language to be used for xHarbour messages. The default value is "EN".
_SET_MARGIN	*<nColumns>* (0) **See also:** command SET MARGIN. Sets the left margin for all printed output. The default value is 0. Note: PCol() reflects the printer's column position including the margin (e.g., SET MARGIN TO 5 followed by DevPos(5, 10) makes PCol() return 15).
_SET_MBLOCKSIZE	*<nMemoBlockSize>* **See also:** command SET MEMOBLOCK. This setting specifies the default block size for memo fields. If not set, which is the default, the default block size for memo fields is defined by the RDD.
_SET_MCENTER	*<lOnOff>* | on | OFF **See also:** command SET MESSAGE. This setting toggles the CENTER option of the SET MESSAGE command. If enabled, display PROMPTs centered on the MESSAGE row. If disabled, which is the default, display PROMPTs at column position 0 on the MESSAGE row.
_SET_MESSAGE	*<nRow>* **See also:** command SET MESSAGE. If set to 0, which is the default, PROMPTs are always suppressed. Otherwise, PROMPTs are displayed on the set row. Note: It is not possible to display prompts on the top-most screen row, because row 0 is reserved for the SCOREBOARD, if enabled.
_SET_MFILEEXT	*<cMemoFileExtension>* **See also:** command SET MFILEEXT. This setting specifies the default file extension for memo files. If not set, which is the default, the default extension is defined by the RDD.
_SET_OPTIMIZE	*<lOnOff>* | ON | off **See also:** command SET OPTIMIZE. This setting toggles the filter optimization for database navigation in the current work area.
_SET_PATH	*<cPath>* ("") **See also:** command SET PATH. Specifies a path of directories to search through to locate a file that can't be located in the DEFAULT directory. Defaults to no path (""). Directories must be separated by a semicolon (e.g., "C:\DATA;C:\MORE").
_SET_PRINTER	*<lOnOff>* | on | OFF **See also:** command SET PRINTER. If enabled, QOut() and QQOut() write to the screen and to the printer. If disabled, which is the default, QOut() and QQOut() only write to the screen (and/or to the SET ALTERNATE file).
_SET_PRINTFILE	*<cPrintFileName>* , *<lAdditive>* **See also:** command SET PRINTER. When set, creates or opens file to write QOut(), QQOut() and DevOut() output to. If *<lAdditive>* is TRUE and the file already exists, the file is opened and positioned at end of file. Otherwise, the file is created. If a file is already opened, it is closed before the new file is opened or created (even if it is the same file). The default file extension is ".prn". The default file name is "PRN", which maps to the default printer device. Call with an empty string to close the file.
_SET_SCOREBOARD	*<lOnOff>* | ON | off **See also:** command SET SCOREBOARD. When enabled, which is the default, READ and MemoEdit() display status messages on screen row 0. When disabled, READ and MemoEdit() status messages are suppressed.

_SET_SOFTSEEK	*<lOnOff>* | on | OFF **See also:** command SET SOFTSEEK. When enabled, a SEEK that fails will position the record pointer to the first key that is higher than the sought after key or to LastRec() + 1 if there is no higher key. When disabled, which is the default, a SEEK that fails will position the record pointer to LastRec()+1.
_SET_STRICTREAD	*<lOnOff>* | on | OFF **See also:** command SET STRICTREAD. When disabled, which is the default, internal memory buffers are used to load datbase fields into memory variables. When enabled, datbase fields are always read from disk.
_SET_TIMEFORMAT	*<cTimeFormat>* **See also:** command SET TIME. *<cTimeFormat>* is a character string specifying the Time part of DateTime() values. It is coded as "hh:mm:ss.ccc [pm]".
_SET_TRACE	*<lOnOff>* | ON | off **See also:** command SET TRACE. When enabled, which is the default, output of function TraceLog() is written to the trace file specified with _SET_TRACEFILE. When disabled, function TraceLog() produces no output.
_SET_TRACEFILE	*<cFileName>* The setting specifies the trace file name where function TraceLog() writes output to. The default value is "trace.log".
_SET_TRACESTACK	*<nCallStack>* The setting defines the first routine in the callstack to add to the trace file with function TraceLog(). The default value is 2.
_SET_TRIMFILENAME	*<lOnOff>* When disabled, which is the default, strings containing file names are treated according to SET FILECASE and SET DIRCASE. When enabled, trailing blank spaces in file names are ignored, in addition.
_SET_TYPEAHEAD	*<nKeyboardBufferSize>* **See also:** command SET TYPEAHEAD. Sets the size of the keyboard typeahead buffer. Defaults to 50. The minimum is 16 and the maximum is 4096.
_SET_UNIQUE	*<lOnOff>* | on | OFF **See also:** command SET UNIQUE. When enabled, indexes are not allowed to have duplicate keys. When disabled, indexes are allowed duplicate keys.
_SET_VIDEOMODE	*<nVideoMode>* **See also:** command SET VIDEOMODE. Specifies the video mode.
_SET_WRAP	*<lOnOff>* | on | OFF **See also:** command SET WRAP. When enabled, lightbar menus can be navigated from the last position to the first and from the first position to the last. When disabled, which is the default, there is a hard stop at the first and last positions.

Info

See also: SetCancel()
Category: Environment functions
Header: set.ch
Source: rtl\set.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example implements a user defined function that saves/restores all
// system settings available in xHarbour. Note the constants _SET_COUNT,
// HB_SET_COUNT and HB_SET_BASE. They are used to distinguish Clipper
// compatible settings from xHarbour additions.
```

```
#include "Set.ch"

PROCEDURE Main
   LOCAL aSet := Settings()

   AEval( aSet, {|x| QOut( ValToPrg(x) ) } )
RETURN

FUNCTION Settings( aNewSet )
   LOCAL nAll := _SET_COUNT + HB_SET_COUNT
   LOCAL aOldSet[ nAll ]
   LOCAL nSetting := 0, i, j

   DO WHILE ++ nSetting <= _SET_COUNT
      IF aNewSet == NIL
         aOldSet[nSetting] := Set( nSetting )
      ELSE
         aOldSet[nSetting] := Set( nSetting, aNewSet[nSetting] )
      ENDIF
   ENDDO

   j := nSetting
   FOR i:=1 TO HB_SET_COUNT
      nSetting := HB_SET_BASE + i - 1
      IF aNewSet == NIL
         aOldSet[j] := Set( nSetting )
      ELSE
         aOldSet[j] := Set( nSetting, aNewSet[j] )
      ENDIF
      j++
   NEXT
RETURN aOldSet
```

SetAtLike()

Sets the search mode for At***() functions

Syntax

```
SetAtLike( [<nNewMode>], [<cWildCard>] ) --> nOldMode
```

Arguments

<nNewMode>

This parameter specifies the search mode for At***() functions. Two numeric values are recognized:

Search modes for At*() functions**

Value	Description
0	Searches exact match of search string
1	Search string may contain wild card characters

<cWildCard>

Optionally, a single character can be specified as a wild card character. The default wild card character is a question mark (?). When a wild card character is included in the search string, it matches all characters in the searched string. Note that the asterisk (*) cannot be used as a wild card character.

Return

The function returns the previous SetAtAlike() mode as a numeric value.

Info

See also: AfterAtNum(), AtAdjust(), AtNum(), AtRepl(), BeforAtNum(), NumAt(), StrDiff()
Category: CT:String manipulation, Character functions
Source: ct\ctstr.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example uses function AtNum() to demonstrate wild card
// matching with SetAtLike().

   PROCEDURE Main
      LOCAL cStr := "Hello, Mister Miller"
      LOCAL nPos

      ? nPos := AtNum( "Miller", cStr )     // result: 15
      ? nPos := AtNum( "Mi??er", cStr )     // result: 0

      SetAtLike( 1 )
      ? nPos := AtNum( "Mi??er", cStr )     // result: 15
      ? SubStr( cStr, nPos )                // result: Miller

      ? nPos := AtNum( "Mi??er", cStr, 1 )  // result: 8
      ? SubStr( cStr, nPos )                // result: Mister Miller
   RETURN
```

SetBit()

Sets one or more bits of a numeric integer value to 1.

Syntax

```
SetBit( <nInteger>|<cHex>, [<nBitPos,...>]) --> nInteger
```

Arguments

<nInteger>

A numeric 32-bit integer value can be specified as first parameter.

<cHex>

Alternatively, the integer can be passed as a hex-encoded character string (see NumToHex()).

<nBitPos>

The positions of the bits to set are defined as a comma separated list of numeric integer values. The range for *<nBitPos>* is 1 to 32.

Return

The function returns a numeric integer value with the bits at the specified positions set to 1.

Info

See also: IsBit(), NumAND(), NumNOT(), NumOR(), NumXOR()
Category: CT:NumBits, Bitwise functions, Numbers and Bits
Source: ct\bit2.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays results of SetBit() along with the
// binary representation of the numbers.

   PROCEDURE Main
      ? NtoC( 0, 2, 8, "0" )          // result: 00000000

      ? n := SetBit( 0, 1, 7 )        // result:    65.00
      ? NtoC( n, 2, 8, "0" )          // result: 01000001

      ? n := SetBit( 0, 2, 4, 6 )     // result:    42.00
      ? NtoC( n, 2, 8, "0" )          // result: 00101010
   RETURN
```

SetBlink()

Determines how to treat the asterisk in a SetColor() string.

Syntax

```
SetBlink( [<lOnOff>] ) --> lOldSetting
```

Arguments

<lOnOff>

This is a logical value. If .T. (true) is passed, the asterisk (*) in the background color of a SetColor() string causes the foreground color (text) to blink. If .F. (false) is passed, the background color is set to high intensity.

Return

The function returns the setting which is active before SetBlink() is called.

Description

SetBlink() is a compatibility function which toggles interpretation of the blink attribute (*) if it is specified for the backgroud color of a SetColor() string. The blink attribute is only relevant when an application runs in full screen text mode. In addition, its availability is operating system dependent.

If the blink attribute is not supported by the operating system, the asterisk (*) is treated like the high intensity color attribute (+).

Info

See also: ColorSelect(), SetColor()
Category: Screen functions
Source: rtl\setcolor.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays text with the SetBlink() setting on and off.

   PROCEDURE Main
      CLS
      SetColor( "N/GR*" )

      @ 0, 0 CLEAR TO 2, MaxCol()
      ? "Blink attribute is:", SetBlink()

      WAIT "Press a key to toggle blink attribute"

      SetBlink( .NOT. SetBlink() )

      @ 0, 0 CLEAR TO 2, MaxCol()
      ? "Blink attribute is:", SetBlink()

      WAIT "Press a key to end"
   RETURN
```

SetCancel()

Determines if Alt+C and Ctrl+Break terminate an application

Syntax

```
SetCancel( [<lOnOff>] ) --> lOldSetting
```

Arguments

<lOnOff>

This is a logical value. If .T. (true) is passed, the keys Alt+C and Ctrl+Break terminate an application unconditionally. .F. (false) disables automatic program termination when a user presses these keys.

Return

The function returns the setting which is active before SetCancel() is called.

Description

SetCancel() is used to disable the automatic program termination routine which can be activated by pressing the keys Alt+C or Ctrl+Break. The setting is .T. (true) by default. When a user presses either key combination, an xHarbour application is terminated unconditionally.

If automatic program termination is not desired, SetCancel() should be set to .F. (false) and a program termination routine should be implemented.

Info

See also: SET ESCAPE, SET KEY, Set(), SetKey()
Category: Environment functions
Source: rtl\set.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates howto override the default program termination
// for Alt+C or Ctrl+Break with a user defined one.

   #include "Inkey.ch"


   PROCEDURE Main

      WAIT "Alt+C terminates unconditionally"

      SetCancel( .F. )
      SetKey( K_ALT_C, {|| MyExitProc() } )

      WAIT "Alt+C calls user defined exit routine"

      ? "Normal program termination"
   RETURN


   PROCEDURE MyExitProc
      LOCAL n := Alert( "Exit program?", { "Yes", "No" } )

      IF n == 1
```

```
        ? "User terminated program"
        QUIT
     ENDIF
  RETURN
```

SetClearA()

Sets the default color attribute for clearing the screen.

Syntax

```
SetClearA( [<xColor>] ) --> cNull
```

Arguments

<xColor>

This is either a single color value (see SetColor()), or its numeric color attribute (see ColorToN()). It defines the default color for clearing the screen. If no parameter is passed, the default color ("W/N") is set.

Return

The return value is always a null string ("").

Info

See also:	GetClearA()
Category:	CT:Video, Screen functions
Source:	ct\setclear.c
LIB:	xhb.lib
DLL:	xhbdll.dll

SetClearB()

Sets the default character attribute for clearing the screen.

Syntax

```
SetClearB( [<xChar>] ) --> cNull
```

Arguments

<cChar>

This is a single character or its numeric ASCII code. It defines the character for clearing the screen. If no parameter is passed, the default character (32) is set.

Return

The return value is always a null string ("").

Info

See also: GetClearB()
Category: CT:Video, Screen functions
Source: ct\setclear.c
LIB: xhb.lib
DLL: xhbdll.dll

SetClrPair()

Replaces a color value in a color string.

Syntax

```
SetClrPair( <cColorString>, <nPos>, <cColor> ) --> cNewColorString
```

Arguments

`<cColorString>`

This is a SetColor() compliant character string holding color values as a comma separated list.

`<nPos>`

This numeric value specifies the ordinal position of the color value to find in *<cColorString>*. The first color value in the color string has the ordinal position 1.

`<cColor>`

A character string holding a color value consisting of foreground and background color.

Return

The function replaces the color at the ordinal position *<nPos>* with *<cColor>* and returns the modified color string. If *<cColorString>* has less than *<nPos>* colors, the return value is a null string ("").

Info

See also:	GetClrPair(), SetColor()
Category:	Screen functions
Source:	rtl\color53.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates how a color value can be
// replaced in a color string.

   PROCEDURE Main
      LOCAL cColor := "W/R,W/G,W/B,W+/R,W+/G,W+/B,W/BG,W+/BG"

      ? GetClrPair( cColor, 6 )              // result: W+/B

      cColor := SetClrPair( cColor, 6, "BG+/R" )

      ? GetClrPair( cColor, 6 )              // result: BG+/R

   RETURN
```

SetColor()

Retrieves and/or changes the current color setting for text mode.

Syntax

```
SetColor( [<cNewColorString>] ) --> cOldColorString
```

Arguments

<cNewColorString>

The parameter is a character string holding color settings. The first five colors are used for different pre-defined scopes. A color setting consists of a color pair that defines the foreground and background color in text mode applications. Single colors are specified with a letter (see description), foreground and background colors must be separated with a slash, and color pairs for different scopes must be comma separated.

Color settings in a color string

Setting	Position	Scope
Standard	1	All screen output commands and functions
Enhanced	2	GETs and selection highlights
Border	3	Border around screen, not supported on most monitors
Background	4	Not supported
Unselected	5	Unselected GETs

Return

The function returns a color string holding the previous color settings.

Description

SetColor() is used to query or change the current color settings for screen output in text mode applications. A single color value consists of two letters separated by a backslash. They define the foreground and background color.

A color value can be modified with a color attribute. The plus sign (+) raises the intensity, or brightness, of a color, while the asterisk can be interpreted either a as intensity or as blink attribute (see function SetBlink())

The letters listed in the following table are recognized as color values:

Letters for colors in text mode

Letter	Color monitor	Monochrome
B	Blue	Underline
B+	Bright Blue	Bright Underline
BG	Cyan	White
BG+	Bright Cyan	Bright White
G	Green	White
G+	Bright Green	Bright White
GR	Brown	White
GR+	Yellow	Bright White
I	Inverse Video	Inverse Video
N+	Gray	Black
N, Space	Black	Black
R	Red	White
R+	Bright Red	Bright White
RB	Magenta	White

RB+	Bright Magenta	Bright White
U	Black	Underline
W	White	White
W+	Bright White	Bright White
X	Blank	Blank

Info

See also: ColorSelect(), IsDefColor(), SET COLOR, SET INTENSITY, SetBlink()
Category: Screen functions
Source: rtl\setcolor.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// This example demonstrates how to save, change and restore
// color settings.

   PROCEDURE Main
      LOCAL cOldColor := SetColor()
      LOCAL cNewColor := PadR( "W+/N,W+/B", 40 )

      CLS
      ? "Current color  :", cOldColor
      ?
      SetColor( cNewColor )
      @ Row(), Col() SAY "Enter new color:" GET cNewColor
      READ

      cNewColor := Trim( cNewColor )
      SetColor( cNewColor )
      ? "New color is   :", cNewColor

      SetColor( cOldColor )
      ? "Back to original"
   RETURN
```

SetCursor()

Queries or changes the shape of the cursor on the screen.

Syntax

```
SetCursor( [<nNewCursorShape>] ) --> nOldCursorShape
```

Arguments

<nNewCursorShape>

This is a numeric value for wich #define constants listed in SETCURS.CH are used. They specify a particular cursor shape.

Constants for cursor shapes

Constant	Value	Description
SC_NONE	0	No cursor
SC_NORMAL	1	Underline
SC_INSERT	2	Lower half block
SC_SPECIAL1	3	Full block
SC_SPECIAL2	4	Upper half block

Return

The function returns a numeric value representing the cursor shape.

Description

SetCursor() specifies the shape of the screen cursor for text mode applications. Passing the value zero hides the cursor. A value greater than zero displays the cursor in the corresponding shape. If no parameter is passed, the function returns the current setting.

Info

See also: SET CONSOLE, SET CURSOR, SET(), SETCOLOR(), SetPos()
Category: Screen functions
Header: Setcurs.ch
Source: rtl\setcurs.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays all possible cursor shapes by iterating
// an array.

   #include "Inkey.ch"
   #include "Setcurs.ch"

   PROCEDURE Main
      LOCAL nCurrent    := 0
      LOCAL nOldCursor := SetCursor()
      LOCAL aNewCursor := { ;
          { "SC_NONE     ", SC_NONE     }, ;
          { "SC_NORMAL   ", SC_NORMAL   }, ;
          { "SC_INSERT   ", SC_INSERT   }, ;
          { "SC_SPECIAL1", SC_SPECIAL1 }, ;
          { "SC_SPECIAL2", SC_SPECIAL2 }  }
```

```
   DO WHILE Lastkey() <> K_ESC
      @ 0, 0
      IF ++ nCurrent > 5
         nCurrent := 1
      ENDIF
      SetCursor( aNewCursor[nCurrent,2] )
      ? "Cursor: ", aNewCursor[nCurrent,1]
   ENDDO

RETURN
```

SetDate()

Changes the system date from a Date value.

Syntax

```
SetDate( <dDate> ) --> lSuccess
```

Arguments

<dDate>

This is a Date value the sytsme date should be set to.

Return

The function returns .T. (true) when the system date is successfully changed, otherwise .F. (false) is returned.

Info

See also:	Date(), SetNewDate(), SetTime()
Category:	CT:DateTime, Environment functions, Date and time
Source:	ct\dattime3.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

SetErrorMode()

Queries or changes the behavior with operating system errors.

Syntax

```
SetErrorMode( [<nNewErrorMode>] ) --> nOldErrorMode
```

Arguments

<nNewErrorMode>

This parameter is a numeric value of 0 or 1. Other values are ignored.

Return

The function returns a numeric value indicating the previous error mode.

Description

Function SetErrorMode() instructs the operating system how an xHarbour application handles errors that occur on the operating system level. For example, when function DiskChange() is called and the specified disk drive has no disk inserted, the operating system would display a message box prompting the user for inserting a disk. To suppress message boxes displayed by the operating system, SetErrorMode(1) must be called.

SetErrorMode(0) enables the display of message boxes created by the operating system.

Info

See also:	DiskChange(), ErrorBlock(), IsDisk()
Category:	Error functions, xHarbour extensions
Source:	rtl\errorsys.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

SetFAttr()

Sets file attributes.

Syntax

```
SetFAttr( <cFileName>, [<nAttributes>] ) --> nErrorCode
```

Arguments

<cFileName>

This is a character string holding the name of the file to set attributes for. It must include path and file extension. If the path is omitted from *<cFileName>*, the file is searched in the current directory only.

<nAttributes>

This is a numeric value specifying the file attributes to set. Values of the following list are used for file attributes. To specifiy multiple attributes, pass the sum of the corresponding values:

Values for file attributes

Value	Attribute
0	Normal
1	Read only
2	Hidden
4	System
8	Volume
32	Archived

Return

The function returns zero on success or a numeric error code on failure.

Info

See also:	BitToC(), CtoBit(), FileAttr(), FileSeek()
Category:	CT:DiskUtil, File functions, Low level file functions
Source:	ct\files.c
LIB:	xhb.lib
DLL:	xhbdll.dll

SetFCreate()

Sets the default file attribute(s) for creating files.

Syntax

```
SetFcreate( [<nNewAttribute>] ) --> nOldAttribute
```

Arguments

`<nNewAttribute>`

This is a numeric value specifying the file attributes to set. Values of the following list are used for file attributes. To specifiy multiple attributes, pass the sum of the corresponding values:

Values for file attributes

Value	Attribute
0	Normal
1	Read only
2	Hidden
4	System
8	Volume
32	Archived

Return

The function returns the default file attribute(s) current before the function is called.

Description

This function exists for compatibility reasons only. A default file attribute can be queried and/or set with SetFCreate(). However, no file function uses this file attribute as a default value.

Info

See also:	FileAppend(), FileCCont(), FileCopy(), ScreenFile(), StrFile()
Category:	CT:DiskUtil, File functions, Low level file functions
Source:	ct\strfile.c
LIB:	xhb.lib
DLL:	xhbdll.dll

SetFDaTi()

Sets the last change date and time of a file.

Syntax

```
SetFDaTi( <cFileName> , ;
         [<dFileDate>], ;
         [<cFileTime>]  ) --> lSuccess
```

Arguments

<cFileName>

This is a character string holding the name of the file to set date and time for. It must include path and file extension. If the path is omitted from *<cFileName>*, the file is searched in the current directory only.

<dFileDate>

This is the date value to be set. It defaults to the return value of Date().

<cFileTime>

This is a time formatted string that defines the new file time. It defaults to Time().

Return

The function returns .T. (true) if the file date and time is set, otherwise .F. (false) is returned.

Info

See also:	FileDate(), FileSeek(), FileTime()
Category:	CT:DiskUtil, File functions, Low level file functions
Source:	ct\files.c
LIB:	xhb.lib
DLL:	xhbdll.dll

SetKey()

Associates a code block with a key.

Syntax

```
SetKey( <nInkeyCode>    , ;
       [<bNewCodeblock>], ;
       [<bCondition>]     ) --> bOldCodeblock
```

Arguments

<nInkeyCode>

This is a numeric value representing the key code of the key to associate *<bNewCodeblock>* with. The key code is returned by the Inkey() function. To specify values for this parameter use #define constants from the file INKEY.CH.

<bNewCodeblock>

This parameter optionally specifies a code block to be executed when the key *<nInkeyCode>* is pressed. Passing NIL explicitly for *<bNewCodeblock>* releases a previously assigned code block.

<bCondition>

Optionally, a code block returning a logical value can be passed. If specified, the code block *<bNewCodeblock>* is only evaluated in a wait state, when *<bCondition>* returns .T. (true).

Note: this code block can only be queried with function HB_SetKeyGet().

Return

The function returns the code block associated with *<nInkeyCode>* before SetKey() is called. If no code block is associated with the key, the return value is NIL.

Description

SetKey() queries or changes associations between key codes and code blocks. When a code block is associated with a key and the user presses this key, the code block is automatically executed during a wait state. An application enters a wait state when user input is requested. This includes functions like AChoice(), Browse(), DbEdit(), MemeoEdit() and commands like ACCEPT, INPUT, READ and WAIT, but it does not include the Inkey() function. Inkey() does not monitor key/code block associations.

The associated code block receives three parameters: the return values of functions ProcName(), ProcLine() and ReadVar().

When an application starts, the F1 key is automatically associated with a code block calling the user-defined function or procedure named Help, if it exists.

Info

See also: Eval(), HB_SetKeyArray(), HB_SetKeyCheck(), HB_SetKeyGet(), HB_SetKeySave(), Inkey(), SET KEY
Category: Environment functions, Keyboard functions
Header: inkey.ch
Source: rtl\setkey.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example implements function SetKeys() which accepts a
// two column array holding data for key/code block associations.
// Passing the array for the first time, activates the
// associations. Passing it a second time, releases them.

   #include "Inkey.ch"

   PROCEDURE Main
      LOCAL cString := "Testing SetKey()"
      LOCAL aKey := { ;
          { K_F2, {|c1,n,c2| F2_Key(c1,n,c2) } }, ;
          { K_F3, {|c1,n,c2| F3_Key(c1,n,c2) } }  ;
        }

      CLS

      // associate keys with code blocks
      SetKeys( aKey )

      @ 10,10 SAY "Press F1, F2 or F3" GET cString
      READ

      // release code block associations
      SetKeys( aKey )
   RETURN


   PROCEDURE Help( cProcName, nProcLine, cReadVar )
      LOCAL cScreen := SaveScreen()
      LOCAL bBlock  := SetKey( K_F1, NIL ) // no recursive calls
      CLS
      ? "Help routine"
      ? "Called from:", cProcName, nProcLine, cReadVar
      WAIT
      SetKey( K_F1, bBlock )
      Restscreen(,,,, cScreen )
   RETURN


   PROCEDURE F2_Key( cProcName, nProcLine, cReadVar )
      LOCAL cScreen := SaveScreen()
      LOCAL bBlock  := SetKey( K_F2, NIL ) // no recursive calls
      CLS
      ? "F2 key pressed"
      ? "Called from:", cProcName, nProcLine, cReadVar
      WAIT
      SetKey( K_F2, bBlock )
      Restscreen(,,,, cScreen )
   RETURN
```

```
PROCEDURE F3_Key( cProcName, nProcLine, cReadVar )
   LOCAL cScreen := SaveScreen()
   LOCAL bBlock  := SetKey( K_F3, NIL ) // no recursive calls
   CLS
   ? "F3 key pressed"
   ? "Called from:", cProcName, nProcLine, cReadVar
   WAIT
   SetKey( K_F3, bBlock )
   Restscreen(,,,, cScreen )
RETURN


PROCEDURE SetKeys( aKeys )
   LOCAL i, imax := Len( aKeys )

   FOR i:=1 TO imax
      aKeys[i,2] := SetKey( aKeys[i,1], aKeys[i,2] )
   NEXT
RETURN
```

SetLastError()

Sets a numeric value as last error code.

Syntax

```
SetLastError( <nNewErrorCode> ) --> nOldErrorCode
```

Arguments

<nNewErrorCode>

This is the numeric value specifying the new return value of function GetLastError().

Return

The function returns the current error code of GetLastError().

Description

SetLastError() is used to void any previous error code returned by GetLastError() before a DLL function is executed. Normally, the function is called with the parameter 0, which means "no error". If GetLastError() returns a value <> 0 when the DLL function has returned, a programmer can be sure that the error occured within this DLL function.

Info

See also:	CallDll(), DllCall(), GetLastError()
Category:	DLL functions, xHarbour extensions
Source:	rtl\dllcall.c
LIB:	xhb.lib
DLL:	xhbdll.dll

SetLastKey()

Changes the return value of function LastKey()

Syntax

```
SetLastKey( <nKey> ) --> cNullString
```

Arguments

<nKey>

This is the numeric Inkey() code of the key to set function LastKey() to. Refer to the file Inkey.ch for #define constants that can be used for *<nKey>*.

Return

The function changes the return value of LastKey() to *<nKey>* and returns a null string ("")

Info

See also:	Inkey(), LastKey(), NextKey()
Category:	CT:Settings, Keyboard functions
Header:	Inkey.ch
Source:	ct\setlast.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example changes LastKey() to K_ESC as soon as an
// alphabetic key is pressed

   #include "Inkey.ch"

   PROCEDURE Main
      LOCAL nKey := 0

      CLS

      DO WHILE LastKey() <> K_ESC
         nKey := Inkey(0.1)

         @ 0,0 SAY "Inkey code: "
         ?? nKey

         IF IsAlpha( Chr(nKey) )
            SetLastKey( K_ESC )
         ENDIF

      ENDDO
   RETURN
```

SetMode()

Changes the size of a console window.

Syntax

```
SetMode( <nRowCount>, <nColCount> ) --> lSuccess
```

Arguments

<nRowCount>

This is a numeric value specifying the height of a console window. It is the number of rows available for display.

<nColCount>

This is a numeric value specifying the width of a console window. It is the number of columns available for display.

Return

The function returns .T. (true) if the size of a console window is changed, otherwise .F. (false).

Description

SetMode() changes the size of a console window to the specified number of rows and columns. The changed size is reflected by functions MaxRow() and MaxCol(). If the values for <*nRowCount*> or <*nColCount*> are too large to fit on the screen, they are adjusted.

Note: if a console application runs in full screen text mode, there are only a limited number of row/column combinations that can be displayed. This is hardware dependent. Common combinations for the number of rows and columns are: 25,80 | 43,80 | 50,80 | 60,80 | 25,132 | 43,132 | 50,132 | 60,132.

Info

See also: MaxCol(), MaxRow()
Category: Screen functions
Source: rtl\gx.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example allows for interactively changing the size of a
// console window.

   #include "Inkey.ch"

   PROCEDURE Main
      LOCAL nMaxRow := MaxRow()
      LOCAL nMaxCol := MaxCol()

      CLS

      DO WHILE LastKey() <> K_ESC
         @ 1,1 SAY "Enter MaxRow:" GET nMaxRow PICTURE "999"
         @ 2,1 SAY "Enter MaxCol:" GET nMaxCol PICTURE "999"
         READ

         CLS
```

```
      IF SetMode( nMaxRow+1, nMaxCol+1 )
         ? "New MaxRow()", MaxRow()
         ? "New MaxCol()", MaxCol()
      ELSE
         ? "Unable to set new screen mode"
      ENDIF

      WAIT
      CLS
   ENDDO

RETURN
```

SetMouse()

Determines the visibility of the mouse cursor.

Syntax

```
SetMouse( [<lOnOff>], [<nRow>], [<nCol>] ) --> lIsMouseVisible
```

Arguments

`<lOnOff>`

A logical value can be passed. .T. (true) makes the mouse cursor visible and .F. (false) hides it.

`<nRow>`

A numeric value between 0 and MaxRow() specifying the new row position of the mouse cursor.

`<nCol>`

A numeric value between 0 and MaxCol() specifying the new columnn position of the mouse cursor.

Return

The function returns .T. (true) when the mouse cursor is visible, otherwise .F. (false)

Description

SetMouse() is used in full screen or console window applications to show or hide the mouse cursor and optionally move it to a new position on the screen. It combines the functions MSetPos(), MShow() and MHide().

Info

See also:	NumButtons(), MSetCursor(), MSetPos()
Category:	Mouse functions
Source:	rtl\mousex.c
LIB:	xhb.lib
DLL:	xhbdll.dll

SetNetDelay()

Queries or changes the timeout period for Net*() functions.

Syntax

```
SetNetDelay( [<nNewTimeout>] ) --> nOldTimeout
```

Arguments

<nNewTimeout>

This optional numeric value specifies the timout period in seconds that must elapse before a Net*() function returns unsuccessfully. The default value is 30 seconds.

Return

The function returns the previous timeout period in seconds as a numeric value.

Description

SetNetDelay() queries and/or changes the current timeout period monitored by Net*() functions. When the requested database operation fails in multi-user access, it is automatically retried until the timeout period has elapsed.

Database operations that can fail in a network with multi-user access are opening a database file (see NetDbUse()) locking a database file (see NetFileLock()), locking a record (see NetRecLock()) or the creation of a new record in a database (see NetAppend()).

All Net*() functions monitoring a timeout period return immediately when the requested operation is successful. If the requested operation fails, it is retried once per second until either the timout period has elapsed or the operation is successfully retried.

The default timeout period is 30 seconds, which is by far sufficient for a database operation to complete in a network. For example, if a database record cannot be locked for write access with NetRecLock() because it is currently locked by another process, the function NetRecLock() tries 30 times to lock the current record until the default timeout period expires. If a record lock cannot be obtained within this period of time, the program logic concurrent database acces in a multi-user environment should be carefully reviewed.

Info

See also:	NetAppend(), NetDbUse(), NetDelete(), NetFileLock(), NetRecall(), NetRecLock(), SetNetMsgColor()
Category:	Database functions, Network functions, xHarbour extensions
Source:	rtl\ttable.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

SetNetMsgColor()

Queries or changes the color for displaying failure messages of Net*() functions.

Syntax

```
SetNetMsgColor( [<cNewColor>] ) --> cOldColor
```

Arguments

<cNewColor>

This is a SetColor() compliant color string used to display messages when a Net*() function fails.

Return

The function returns the previous color as a character string.

Description

SetNetMsgColor() queries and/or changes the color of messages displayed by Net*() functions on the screen, when a requested database operation fails in a multi-user environment. The messages are displayed in the last screen row (at the bottom of the screen).

Info

See also:	NetAppend(), NetDbUse(), NetDelete(), NetFileLock(), NetRecall(), NetRecLock(), SetNetDelay()
Category:	Network functions, Screen functions, xHarbour extensions
Source:	rtl\ttable.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

SetNewDate()

Changes the system date from Numeric values.

Syntax

```
SetNewDate( <nYear>, <nMonth>, <nDay> ) --> lSuccess
```

Arguments

<nYear>

A numeric value specifying the year of the new system date.

<nMonth>

A numeric value specifying the month of the new system date.

<nDay>

A numeric value specifying the day of the new system date.

Return

The function returns .T. (true) when the system date is successfully changed, otherwise .F. (false) is returned.

Info

See also:	Date(), SetDate(), SetTime()
Category:	CT:DateTime, Date and time, xHarbour extensions
Source:	ct\settime.c
LIB:	xhb.lib
DLL:	xhbdll.dll

SetNewTime()

Changes the system time from Numeric values.

Syntax

```
Setnewtime( <nHour>, <nMinute>, <nSecond> ) --> lSuccess
```

Arguments

<nHour>

A numeric value specifying the hour of the new system time.

<nMinute>

A numeric value specifying the minute of the new system time.

<nSecond>

A numeric value specifying the second of the new system time.

Return

The function returns .T. (true) when the system time is successfully changed, otherwise .F. (false) is returned.

Info

See also:	SetDate(), SetNewDate(), SetTime(), Time()
Category:	CT:DateTime, Date and time, xHarbour extensions
Source:	ct\settime.c
LIB:	xhb.lib
DLL:	xhbdll.dll

SetPos()

Changes the position of the screen cursor in text mode.

Syntax

```
SetPos( <nRow>, <nCol> ) --> NIL
```

Arguments

<nRow>

This is a numeric value in the range of 0 to MaxRow(). It specifies the new row position to move the screen cursor to.

<nCol>

This is a numeric value in the range of 0 to MaxCol(). It specifies the new column position to move the screen cursor to.

Return

The return value is always NIL.

Description

The function SetPos() moves the screen cursor in text mode applications to the specified row and column coordinates. The new position is reflected by the functions Row() and Col(). Use function SetCursor() to change visibility and shape of the cursor.

The current screen cursor position is used by console commands and functions which do not recognize screen coordinates as starting point for display.

Info

See also:	Col(), Row(), SET CURSOR, SetCursor(), SetPosBs()
Category:	Screen functions
Source:	rtl\setpos.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The displays the current positionof the screen cursor, changes it
// and displays a string at the new position.

   PROCEDURE Main
      CLS
      ? Row(), Col()

      SetPos( 13, 30 )
      ?? "New output goes here"

      ? Row(), Col()
   RETURN
```

SetPosBS()

Moves the screen cursor (text mode) one column to the right.

Syntax

```
SetPosBS() --> NIL
```

Return

The function move the screen cursor in text mode one column to the right and returns NIL.

Info

See also: Col(), Row(), SET CURSOR, SetCursor(), SetPos()
Category: Screen functions
Source: rtl\setposbs.c
LIB: xhb.lib
DLL: xhbdll.dll

SetPrc()

Changes the PRow() and PCol() values.

Syntax

```
SetPrc( <nRow>, <nCol> ) --> NIL
```

Arguments

<nRow>

This is a numeric value specifying the new return value of function PRow().

<nCol>

A numeric value specifying the new return value of function PCol().

Return

The return value is always NIL.

Description

SetPrc() is used to programmatically set the return values of functions PRow() and PCol(). Both functions maintain internal counters for monitoring the print head position in console applications when output is directed to the printer, either with SET PRINTER or SET DEVICE.

The PCol() value is incremented with every character sent to the printer. If a character string contains non printable characters or printer control codes, the physical print head does not change its position on paper while the internal counter is incremented. In this situation, the internal counter gets out of sync with the actual print head position. The internal counter must then be manually adjusted using SetPrc().

When print output is accomplished using @...SAY, SetPrc() is required to reset PRow() if the new row position is smaller than PRow(). Otherwise, an automatic page eject occurs.

Info

See also:	PCol(), PRow(), SET DEVICE, Set()
Category:	Environment functions
Source:	rtl\console.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example implements a user-defined function that sends non-printable
// characters, or printer control characters, to the printer. The current
// PRow() and PCol() values are saved before characters are sent, and then
// restored using SetPrc().

   FUNCTION PrintCtrlCode( cCtrlCode )
      LOCAL lPrint := Set( _SET_PRINTER, .T. )
      LOCAL nRow   := PRow()
      LOCAL nCol   := PCol()

      ?? cCtrlCode

      Set( _SET_PRINTER, lPrint )
   RETURN SetPrc( nRow, nCol )
```

SetPrec()

Specifies the computing precision for trigonometric functions.

Syntax

```
SetPrec( <nDecimalPlaces> ) --> cNullString
```

Arguments

<nDecimalPlaces>

A numeric value in the range between 1 and 16 can be passed. The default precision is 16.

Return

The function returns always a null string ("").

Info

See also:	ACos(), ASin(), ATan(), ATn2(), Cos(), Cot(), GetPrec(), Sin(), Tan()
Category:	CT:Math, Mathematical functions, Trigonometric functions
Source:	ct\ctmath.c
LIB:	xhb.lib
DLL:	xhbdll.dll

SetRegistry()

Creates a key/value pair in the registry.

Syntax

```
SetRegistry( <nHKEY>   , ;
             <cRegPath>, ;
             <cRegKey> , ;
             <xValue>    ) --> lSuccess
```

Arguments

<nHKEY>

This numeric parameter specifies the root entry in the Windows registry to create a new registry key in. #define constants are available in the Winreg.ch file that can be used for *<nHKEY>*. They begin the the prefix HKEY_.

<cRegPath>

This is a character string holding the registry path to create *<cRegKey>* in. The path must include a backslash as delimiter, if required, but may neither begin or end with a backslash.

<cRegKey>

This is a character string holding the name of the registry key to create. It is created directly underneath *<cRegPath>*.

<xValue>

This is a value of data type Character, Date, Logical or Numeric to assign to *<cRegKey>*.

Return

The function returns .T. (true), when the specified key/value pair is created in the registry. Otherwise .F. (false) is returned.

Description

Function SetRegistry() creates a key/value pair in the Windows registry. If the key *<cRegKey>* exists already, the value *<xValue>* is assigned. The value may only be of Valtype() "C", "D", "L" or "N". Other data types are not supported.

A Date value is converted with function StoD() an stored as a REG_SZ value in the registry

A logical value will be converted to 1 or 0 and set as REG_DWORD.

Winreg.ch

Winreg.ch adds quite some overhead to an application program by adding structure definitions. If this is not required, Winreg.ch does not need to be #included. SetRegistry() recognizes the following values for *<nHKEY>* in addition to the HKEY_* #define constants:

Registry keys

Registry Key	Equivalent value
HKEY_LOCAL_MACHINE	0
HKEY_CLASSES_ROOT	1
HKEY_CURRENT_USER	2
HKEY_CURRENT_CONFIG	3
HKEY_LOCAL_MACHINE	4
HKEY_USERS	5

Note: on Windows NT, 2000, XP or later, the user may need certain security rights in order to be able to change the registry.

Info

See also: QueryRegistry(), GetRegistry()
Category: Registry functions, xHarbour extensions
Header: Winreg.ch
Source: rtl\winreg.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example creates a new registry path and fills it with
// five new key/value pairs. The new created keys are read from
// the registry and displayed.

   * #include "Winreg.ch"           // not needed for this example

   #define HKEY_CURRENT_USER  0     // use alternative #define constant

   PROCEDURE Main
      LOCAL nHKey    := HKEY_CURRENT_USER
      LOCAL cRegPath := "SOFTWARE\MyApplication"
      LOCAL hRegKey  := Hash()
      LOCAL cRegKey

      hRegKey["Version"]      := "1.0"
      hRegKey["Creationdate"] := Date()
      hRegKey["Demo"]         := .T.
      hRegKey["Trialdays"]    := 30
      hRegKey["rootpath"]     := "C:\programs\myapp\"

      HEval( hRegKey, ;
             {|cKey, xVal| SetRegistry( nHKey, cRegPath, cKey, xVal ) ;
           } )

      FOR EACH cRegKey IN hRegKey:keys
         ? cRegKey, "=", GetRegistry( nHKey, cRegPath, cRegKey )
      NEXT
   RETURN
```

SetTime()

Changes the system time from a Time string.

Syntax

```
SetTime( <cTime> ) --> lSuccess
```

Arguments

<cTime>

A Time() formatted character string specifying the new system time (hh:mm:ss).

Return

The function returns .T. (true) when the system time is successfully changed, otherwise .F. (false) is returned.

Info

See also:	SetDate(), SetNewTime(), Time()
Category:	CT:DateTime, Date and time
Source:	ct\dattime3.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

ShowTime()

Displays the system time continuously at a specified screen position.

Syntax

```
ShowTime( [<nRow>]      , ;
          [<nCol>]      , ;
          [<lHideSecs>], ;
          [<cColor>]    , ;
          [<l12h>]      , ;
          [<lAmPm>]       ) --> cNull
```

Arguments

<nRow>

This is a numeric value in the range of 0 to MaxRow(). It specifies the row position for the time display. It defaults to Row().

<nCol>

This is a numeric value in the range of 0 to MaxCol(). It specifies the column position for the time display. It defaults to Col().

<lHideSecs>

If this parameter is set to .T. (true), the time is displayed without seconds. The default value is .F. (false).

<cColor>

An optional SetColor() compliant color string can be specified to display the time. It defaults to the standard color of SetColor().

<l12h>

If this parameter is set to .T. (true), the time is displayed using a 12h clock. The default value is .F. (false), i.e. a 24h clock is used.

<lAmPm>

If *<l12h>* is .T. (true), an AM/PM indicator can be added to the time display by by setting *<lAmPm>* to .T. (true). The default is .F. (false).

Return

The function returns a null string ("").

Description

The ShowTime() function must be called at least with one parameter to install the time display at the specified position. Calling ShowTime() without a parameter again stops the time display.

Info

See also: Time()
Category: CT:DateTime, Date and time
Source: ct\dattime3.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays the time using a 12h clock
// while the Browse() function is active.

   PROCEDURE Main
      CLS
      ShowTime( 0, MaxCol()-8,,"N/BG" ,.T. , .T. )

      USE Customer

      Browse()

   RETURN
```

Sign()

Converts the sign of a number to a numeric value.

Syntax

```
Sign( <nNumber> ) --> nSign
```

Arguments

<nNumber>

Any numeric value can be passed.

Return

The function returns -1 for negative numbers, 0 for zero, and +1 for positive numbers.

Info

See also:	Abs()
Category:	CT:Math, Mathematical functions
Source:	ct\ctmath2.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Sin()

Calculates the sine.

Syntax

```
Sin( <nAngle> ) --> nSine
```

Arguments

<nAngle>

A numeric value specifies the angle as a fraction (or multiple) of Pi().

Return

The function returns the sine of an angle in the range between -1 and +1.

Info

See also:	Cos(), Cot(), DtoR(), RtoD(), SinH(), Tan()
Category:	CT:Math, Mathematical functions, Trigonometric functions
Source:	ct\trig.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays results of function Sin().

   PROCEDURE Main
      ? Str( Sin( -Pi()*1.0 ), 18, 15)  // result:  0.000000000000000
      ? Str( Sin( -Pi()*0.5 ), 18, 15)  // result: -1.000000000000000
      ? Str( Sin( -Pi()*0.3 ), 18, 15)  // result: -0.809016994374948
      ? Str( Sin(  Pi()*0.0 ), 18, 15)  // result:  0.000000000000000
      ? Str( Sin(  Pi()*0.3 ), 18, 15)  // result:  0.809016994374948
      ? Str( Sin(  Pi()*0.5 ), 18, 15)  // result:  1.000000000000000
      ? Str( Sin(  Pi()*1.0 ), 18, 15)  // result:  0.000000000000000
   RETURN
```

SinH()

Calculates the hyperbolic sine.

Syntax

```
SinH( <nRadians> ) --> nHyperbolicSine
```

Arguments

```
<nRadians>
```

A numeric value specifying the angle value in radians.

Return

The function returns the hyperbolic sine in the range of plus and minus Infinity().

Info

See also:	Cos(), Cot(), DtoR(), RtoD(), SinH(), Tan(), TanH()
Category:	CT:Math, Mathematical functions, Trigonometric functions, xHarbour extensions
Source:	ct\trig.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example display results of SinH(). Note that there is a numeric overflow.

   PROCEDURE Main
      ? Str( SinH( 0.0 ), 18, 15)  // result:  0.000000000000000
      ? Str( SinH( 1.0 ), 18, 15)  // result:  1.175201193643801
      ? Str( SinH( 2.0 ), 18, 15)  // result:  3.626860407847019
      ? Str( SinH( 4.0 ), 18, 15)  // result: 27.289917197127750
      ? Str( SinH( 8.0 ), 18, 15)  // result: ******************
   RETURN
```

SoundEx()

Converts a character string using the Soundex algorithm.

Syntax

```
SoundEx( <cString> ) --> cSoundex
```

Arguments

<cString>

This is a character string to convert.

Return

The functions returns a soundex string. It begins with a letter, followed by three digits.

Description

SoundEx() converts a character string applying the Soundex algorithm. This algorithm compares letters according to their phonetic similarity, so that words with similar phonetics, but different spelling, may yield the same soundex string. This is used to detect misspelled words, or to index database fields by phonetic similarity.

Note that the Soundex algorithm works only with 7bit letters and is case insensitive. It is developed for the English language and may not work with other languages that use special characters in their alphabet.

Info

See also:	DbSeek(), INDEX, OrdWildSeek(), SET SOFTSEEK
Category:	Character functions, Conversion functions
Source:	rtl\soundex.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Space()

Returns a string consisting of blank spaces.

Syntax

```
Space( <nCount> ) --> cSpaces
```

Arguments

<nCount>

This is a numeric value specifying the number of blank spaces to return.

Return

The function returns a character string consisting of <*nCount*> blank spaces (Chr(32)). If <*nCount*> is smaller than 1, the return value is an empty string ("").

Description

Space() is a specialized form of Replicate() that replicates the blank space (chr(32)) <*nCount*> times and returns the result string. Space() is commonly used to initialize variables with a character string buffer that has no content but blank spaces.

Info

See also:	PadC() | PadL() | PadR(), Replicate()
Category:	Character functions
Source:	rtl\space.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Sqrt()

Calculates the square root of a positive number

Syntax

```
SQrt( <nNumber> ) --> nSquareRoot
```

Arguments

<nNumber>

This is a positive numeric value to compute the square root for.

Return

The function returns the square root of *<nNumber>* as a numeric value.

Description

The function calculates the square root of a positive number to double precision. When the result is output, the number of decimal places displayed depends on SET DECIMALS and SET FIXED. Note that the displayed decimals may be rounded for display which does not affect numeric precision in calculations.

Info

See also:	Exp(), Log(), SET DECIMALS, SET FIXED
Category:	Mathematical functions, Numeric functions
Source:	rtl\math.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays various results of SQrt()

   PROCEDURE Main
      SET DECIMALS TO 6

      ? SQrt(0.2)              // Result:  0.447214
      ? SQrt(2.0)              // Result:  1.414214
      ? SQrt(200)              // Result: 14.142136

      ? SQrt(9)                // Result:  3.000000
      ? SQrt(9) ^ 2            // Result:  9.000000
   RETURN
```

Standard()

Selects the standard color of SetColor().

Syntax

```
Standard() --> cNull
```

Return

The function selects the 1st color of the SetColor() setting for standard console output and returns a null string ("").

Info

See also:	ColorSelect(), Enhanced(), UnSelected()
Category:	CT:Video, Screen functions
Source:	ct\color.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

StartThread()

Starts a new thread.

Syntax

```
StartThread( <bBlock>|<cFuncName>|<pFuncPtr> ;
           [,<xParams,...>] ) --> pThreadHandle
or

StartThread( <oObject>, <cMethodName>|<pMethodPtr> ;
           [,<xParams,...>] ) --> pThreadHandle
```

Arguments

`<bBlock>`

This is a code block to be executed in the new thread.

`<cFuncName>`

This is a character string holding the symbolic name of a function or procedure to be executed in the new thread.

`<pFuncPtr>`

Instead o a function orprocedure name, the pointer to a function or procedure can be specified. It can be obtained using function HB_FuncPtr().

`<oObject>`

If the first parameter is an object (Valtype()=="O"), the second parameter specifies the method to execute in the new thread.

`<cMethodName>`

This is a character string holding the symbolic name of the method of *<oObject>* to be executed.

`<pMethodPrt>`

This is a pointer to the method to be executed. It can be obtained using function HB_ObjMsgPtr().

`<xParams,...>`

An optional, comma separated list of values to be passed as parameters to a code block, function, procedure or method can be specified with *<xParams,...>*.

Return

The function returns a handle to the started thread.

Description

StartThread() is the only function in xHarbour that creates a new thread, and assigns to it the program code to be executed in the new thread. An application begins execution always with one thread, the Main thread, from where new threads can be spawned. When an application uses multiple threads, the program code specified for a particular thread runs parallel with program code assigned to other threads. The simultaneous execution of program code in multiple threads is the task of the operating system which grants CPU access for different threads on its own behalf.

Function StartThread() assigns the program code to execute in a new thread to the operating system. This can be a function, procedure, code block or the method of an object. These possibilities are covered by the parameters accepted by StartThread(). In addition, all values specified with *<xParams,...>* are passed on to the routine defined for the new thread when the operating system starts with thread execution.

When StartThread() returns, the new thread may or may not have started to execute the assigned routine. StartThread() merely guarantees that all memory resources are set up for the operating system to run the new thread. Unless an error occurs, the function returns a handle to the new thread. This Thread handle must be preserved for use with other multi-threading functions. The Thread handle is valid as long as the operating systeme executes the thread, i.e. as long as the thread is running. The Thread handle becomes invalid when the operating system terminates the thread. Function IsValidThread() is available to test if a thread is still running.

Notes on Multi-threaded programming

A detailed description of "Threads" and "Multi-threaded programming" goes way beyond the scope of this documentation. There are many books available on this topic and many free online resources inform about them (see www.wikipedia.org for a start)

A programmer has absolutely no control about WHEN the operating system starts or ends a thread, or WHEN it grants a thread access to the CPU. This fact of parallel program execution rises concurrency issues when two threads access the same memory resources, like a STATIC variable, for example. The value of a STATIC variable is not guaranteed when two threads change its value simultaneously. The thread that had last access to a STATIC variable, determines its value.

Concurrency issues for memory resources that may arise in multi-threaded programming can be resolved be means of Mutexes. A Mutex can be locked and freed by one thread. A thread that holds a lock on a Mutex blocks execution of other threads, thus preventing them from accessing the memory recources protected by the Mutex while it is locked. See function HB_MutexCreate() for more information on Mutexes.

Info

See also:	GetCurrentThread(), GetThreadID(), HB_MutexCreate(), HB_OpenProcess(), IsValidThread(), JoinThread(), StopThread(), ThreadSleep()
Category:	Multi-threading functions, xHarbour extensions
Source:	vm\thread.c
LIB:	xhbmt.lib
DLL:	xhbmtdll.dll

Example

```
// The example uses a thread to display the time in the
// upper right corner of the screen while function Browse()
// is active. The thread is suspended for one second with
// ThreadSleep().

   PROCEDURE Main
      LOCAL pThread

      CLS
      USE Customer
      pThread := StartThread( "ShowTime", 0, MaxCol()-7 )

      Browse()

      StopThread( pThread )

      WaitForThreads()
   RETURN


   PROCEDURE ShowTime( nRow, nCol )
      DO WHILE .T.
         DispOutAt( nRow, nCol, Time() )
         ThreadSleep( 1000 )
```

```
    ENDDO
RETURN
```

StoD()

Converts a "yyyymmdd" formatted string to a Date value

Syntax

```
StoD( [<cDate>] ) --> dDate
```

Arguments

<cDate>

This is an optional character string holding 8 digits to convert to a Date value. The digits are interpreted as year, month and day ("yyyymmdd").

Return

The function returns the converted character string as a Date value. If no parameter is passed, or if *<cDate>* is an empty string (""), an empty Date value is returned.

Description

StoD() converts a DtoS() string back to a Date value. This string contains date information which is independent from country specific Date format settings.

Info

See also:	CtoD(), Date(), DtoC(), DtoS(), StoT()
Category:	Conversion functions, Date and time
Source:	rtl\dateshb.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays various results of StoD()

   PROCEDURE Main
      SET CENTURY ON

      ? StoD( "20060501" )  // result: 05/01/2006
      ? StoD( "19060521" )  // result: 05/21/1906
      ? StoD()              // result:   /  /
      ? StoD("")            // result:   /  /
      ? StoD( "ABCDEFG" )   // result:   /  /
   RETURN
```

StopThread()

Stops a thread from outside.

Syntax

```
StopThread( <pThreadHandle>, [<pMutexHandle>] ) --> NIL
```

Arguments

`<pThreadHandle>`

This is the handle of the thread to stop running. A thread handle is returned from function StartThread().

`<pMutexHandle>`

Optionally, the handle of a Mutex owned by *<pThreadHandle>* can be passed. This causes NotifyAll() being called when the thread is stopped. All other threads blocked by *<pMutexHandle>* resume operation when the thread is terminated by the operating system.

Return

The return value is always NIL.

Description

Function StopThread() sends a thread termination request for *<pThreadHandle>* to the operating system, and halts the current thread until the requested second thread has finally terminated. If the second thread owns the Mutex *<pMutexHandle>*, all threads blocked by *<pMutexHandle>* are notified and resume operation when the thread represented by *<pThreadHandle>* has ended.

Be **very cautiuos** with StopThread(). This function instructs the operating system to terminate a running **second** thread defined with *<pThreadHandle>*. That means, the current thread executing StopThread() can stop a second thread from outside. The result of the second thread is undetermined, since it may be interrupted by the operating system at **any** programming instruction (the second thread could be terminated by the operating system within a FOR or DO WHILE loop, before a RETURN statement is executed).

It is highly recommended to use Mutexes to control the begin and end of a thread when the current thread requires the result of another thread.

Important: StopThread() is different from KillThread(). When a thread calls StopThread(), it is suspended until the requested (second) thread has actually ended. KillThread(), in contrast, does not halt the current thread. KillThread() works asynchronously while Stopthread() works synchronously.

Info

See also: HB_MutexCreate(), HB_MutexLock(), HB_MutexUnlock(), KillThread(), Notify(), StartThread(), StopThread()
Category: Multi-threading functions, xHarbour extensions
Source: vm\thread.c
LIB: xhbmt.lib
DLL: xhbmtdll.dll

StoT()

Converts a "YYYYMMDDhhmmss.ccc" formatted string to a DateTime value

Syntax

```
StoT( [<cDateTime>] ) --> dDateTime
```

Arguments

<cDateTime>

This is an optional character string holding up to 18 digits (and period) to convert to a DateTime value. The digits are interpreted as year, month and day ("YYYYMMDD") and Hour, Minute Seconds and Milliseconds ("hhmmss.ccc").

Return

The function returns the converted character string as a DateTime value. If no parameter is passed, or if *<cDateTime>* is not a valud DateTime string (""), an empty DateTime value is returned.

Description

StoT() converts a TtoS() string back to a DateTime value. This string contains date and time information which is independent from country specific Date format settings.

Info

See also:	Date(), DateTime(), StoD(), TtoC(), TtoS()
Category:	Conversion functions, Date and time, xHarbour extensions
Source:	rtl\dateshb.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays various results of StoT()

   PROCEDURE Main
      SET CENTURY ON

      ? StoT( "20070426" )            // result: 04/26/2007
      ? StoT( "20070426133045" )      // result: 04/26/2007 13:30:45.00
      ? StoT( "20070426133045.67" )   // result: 04/26/2007 13:30:45.67
      ? StoD()                        // result:   /  /
      ? StoD( "ABCDEFG" )             // result:   /  /
   RETURN
```

Str()

Converts a numeric value to a character string.

Syntax

```
Str( <nNumber>    , ;
    [<nLength>]   , ;
    [<nDecimals>], ;
    [<lTrim>]      ) --> cString
```

Arguments

<nNumber>

A numeric value to convert to a character string.

<nLength>

An optional numeric value specifying the length of the return string, including sign and decimal places.

<nDecimals>

This is an optional numeric value indicating the number of decimal places to return.

<lTrim>

This parameter defaults to .F. (false). When .T. (true) is passed, the returned string has no leading spaces.

Return

The function returns *<nNumber>* formatted as a character string.

Description

Str() converts a numeric value to a character string. This is required when numeric values must be concatenated with character strings for formatted display, or when index keys containing Numeric and Character fields must be created.

If the length of the return string is specified too small, so that Int(*<nNumber>*) does not fit into the result, Str() returns a string filled with asterisks (*).

If only decimal places do not fit entirely, Str() rounds *<nNumber>* to the requested decimal places.

If both, *<nLength>* and *<nDecimals>*, are not specified, Str() obtains default values for both optional parameters as follows:

Formatting rules of Str()

Numeric value	Length of return string
Field variable	Field length plus decimals
Expressions/constants	Minimum of 10 digits plus decimals
Val()	Minimum of 3 digits
Day() and Month()	3 digits
Year()	5 digits
Recno()	7 digits

Note: the number string is formatted without leading spaces when .T. (true) is specified for the fourth parameter.

Info

See also: CStr(), NumToHex(), StrZero(), Transform(), Val()
Category: Character functions, Conversion functions
Source: rtl\str.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates formatting rules applied by Str()

   PROCEDURE Main
      LOCAL  nValue := 123.456

      ? Str( nValue )              // result:        123.456

      ? Str( nValue,  1 )          // result: *
      ? Str( nValue,  2 )          // result: **

      ? Str( nValue,  3 )          // result: 123
      ? Str( nValue,  4 )          // result:  123
      ? Str( nValue,  5 )          // result:   123

      ? Str( nValue,  5, 1 )       // result: 123.5
      ? Str( nValue,  6, 2 )       // result: 123.46
      ? Str( nValue,  7, 3 )       // result: 123.456
      ? Str( nValue,  8, 4 )       // result: 123.4560

      ? Str( nValue, 10, 0 )       // result:        123
      ? Str( nValue, 10, 1 )       // result:      123.5
      ? Str( nValue, 10, 2 )       // result:     123.46
      ? Str( nValue, 10, 3 )       // result:    123.456
      ? Str( nValue, 10, 4 )       // result:   123.4560
   RETURN
```

Strdel()

Deletes characters from a string based on a mask string.

Syntax

```
StrDel( <cString>, <Mask> ) --> cNewString
```

Arguments

`<cString>`

This is acharacter string where characters are deleted.

`<cMask>`

This is the mask charater string. Characters different from a blank space space indicate that the character at the same position in *<cString>* are removed.

Return

The function removes the characters from *<cString>* according to *<cMask>* and returns the resulting string.

Info

See also:	StrTran(), SubStr()
Category:	Character functions, xHarbour extensions
Source:	rtl\strdel.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates return values of StrDel().

   PROCEDURE Main
      LOCAL cString := "Hello World"

      ? StrDel( cString, "xx  xx " )      // result: llWorld
      ? StrDel( cString, "  xx  xx " )    // result: Heo rld

      ? StrDel( cString, "x x x x x x" )  // result: el ol
      ? StrDel( cString, " x x x x x x" ) // result: HloWrd
   RETURN
```

StrDiff()

Calculates the similarity of two strings.

Syntax

```
StrDiff( <cString1> , ;
         <cString2> , ;
        [<nReplace>], ;
        [<nDelete>] , ;
        [<nInsert>]   ) --> nSimilarity
```

Arguments

<cString1> and <cString2>

Thes are two character strings being compared for their similarity.

<nReplace>

The weighing factor for Replace operations defaults to 3. It can be changed to a numeric value between 0 and 255.

<nDelete>

The weighing factor for Delete operations defaults to 6. It can be changed to a numeric value between 0 and 255. Delete operations are considered the most expensive ones.

<nInsert>

The weighing factor for Insert operations defaults to 1. It can be changed to a numeric value between 0 and 255.

Return

The function returns a numeric value indicating the similarity of two character strings.

Description

The function calculates the Levenshtein distance which indicates the similarity of two character strings. The algorithm weighs Delete, Insert and Replace operations required to transform *<cString1>* into *<cString2>*. The weighing factors of each operation influence the result. It is assumed that two strings are more similar the smaller the result is.

Info

See also:	SetAtLike(), SoundEx()
Category:	CT:String manipulation, Character functions
Source:	ct\strdiff.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows results of StrDiff() for two
// similar and to very different strings.

   PROCEDURE Main

      ? StrDiff( "Mister", "Miller" )      // result: 6

      ? StrDiff( "Clipper", "xHarbour" )   // result: 19

   RETURN
```

StrFile()

Writes a string to a file starting at a specified position.

Syntax

```
StrFile( <cString>      , ;
         <cFileName>    , ;
        [<lUseExisting>], ;
        [<nOffset>]     , ;
        [<lTruncate>]     ) --> nBytesWritten
```

Arguments

`<cString>`

This is a character string to be written to the file *<cFileName>*.

`<cFileName>`

This is a character string holding the name of the file to write to. It must include path and file extension. If the path is omitted from *<cFileName>*, the file is searched in the current directory only.

`<lUseExisting>`

This parameter defaults to .F. (false) so that a new file is created. To write *<cString>* to an existing file, *<lUseExisting>* must be set to .T. (true).

`<nOffset>`

This optional numeric parameter specifies the starting position for writing. It defaults to the end-of-file.

`<lTruncate>`

This parameter defaults to .F. (false). When set to .T. (true), the file is truncated after *<cString>* is written, i.e. the file ends with *<cString>*.

Return

The function returns the number of bytes written to the file as a numeric value.

Info

See also:	CSetSafety(), FileStr(), FWrite()
Category:	CT:DiskUtil, File functions, Low level file functions
Source:	ct\strfile.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example writes the names of all PRG files of the current directory
// to a file.

   PROCEDURE Main
      LOCAL cFileName := "Prg_files.txt"
      LOCAL aFiles    := Directory( "*.prg" )
      LOCAL aFile

      FOR EACH aFile In aFiles
         StrFile( aFile[1] + HB_OsNewLine(), ;
                  cFileName, .T. )
```

```
      NEXT
   RETURN
```

StringToLiteral()

Creates a literal character string from a string.

Syntax

```
StringToLiteral( <cString> ) --> cLiteralString
```

Arguments

`<cString>`

This is a character string to create a literal string from.

Return

The function converts *<cString>* to a literal string and returns the result.

Description

Literal character strings are strings that can be processed by the macro compiler and produce the original string. The function StringToLiteral() searches *<cString>* for quotes and non-printable characters, and replaces them with adequate characters which produce the original when processed by the macro compiler.

Info

See also:	&, CStr(), HB_Serialize(), Valtype(), ValToPrg(), ValToPrgExp()
Category:	Conversion functions, xHarbour extensions
Source:	rtl\cstr.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

StrPeek()

Determines the ASCII code of a specified character in a string.

Syntax

```
StrPeek( <cString>, <nPos> ) --> nASCII
```

Arguments

<cString>

This is a character string to query for an ASCII code.

<nPos>

The ordinal position of the character to query must be specified as a numeric value.

Return

The function returns the ASCII code of the character at the specified position as a numeric value.

Note: the function exists for compatibility reasons only.

Info

See also:	[ ] (string), Asc(), AscPos(), StrPoke()
Category:	Miscellaneous functions
Source:	rtl\strpeek.c
LIB:	xhb.lib
DLL:	xhbdll.dll

StrPoke()

Changes the ASCII code of a specified character in a string.

Syntax

```
StrPoke( <cString>, <nPos>, <nASCII> ) --> cString
```

Arguments

<cString>

This is a character string to change a single character in.

<nPos>

The ordinal position of the character to change must be specified as a numeric value.

<nASCII>

This is the numeric ASCII code to change the specified character to.

Return

The function changes the ASCII code of the character at the specified position and returns the modified string.

Note: the function exists for compatibility reasons only.

Info

See also:	[ ] (string), Chr(), StrPeek()
Category:	Miscellaneous functions
Source:	rtl\strpeek.c
LIB:	xhb.lib
DLL:	xhbdll.dll

StrScreen()

Displays a screen string at the specified position.

Syntax

```
StrScreen( <cScreen>, [<nRow>], [<nCol>] ) --> cNull
```

Arguments

<cScreen>

This is a character string returned from ScreenStr().

<nRow>

A numeric value indicating the screen row for display. It defaults to Row().

<nCol>

A numeric value indicating the screen column for display. It defaults to Col().

Return

The return value is always a null string ("").

Note: the function leaves the cursor position unchanged.

Info

See also:	RestScreen(), ScreenStr()
Category:	CT:Video, Screen functions
Source:	ct\screen3.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

StrSwap()

Exchanges characters between two strings.

Syntax

```
StrSwap( [@]<cString1> , [@]<cString2> ) --> cNull
```

Arguments

`<cString1> and <cString2>`

These are two character strings whose characters are exchanged from left to right. One or both parameters must be passed by reference.

Return

The function exchanges characters between *<cString1>* and *<cString2>* beginning with the first character up to the length of the shorter string. To obtain a result, at least one parameter must be passed by reference. The return value is always a null string ("").

Info

Category:	CT:String manipulation, Character functions
Source:	ct\strswap.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows results of StrDiff() passing either
// parameter or both by reference

   PROCEDURE Main
      LOCAL cStr1, cStr2

      cStr1 := "xHarbour compiler"
      cStr2 := "01234567"

      StrSwap( cStr1, @cStr2 )
      ? cStr1                  // result: xHarbour compiler
      ? cStr2                  // result: xHarbour

      cStr1 := "xHarbour compiler"
      cStr2 := "01234567"

      StrSwap( @cStr1, cStr2 )
      ? cStr1                  // result: 01234567 compiler
      ? cStr2                  // result: 01234567

      cStr1 := "xHarbour compiler"
      cStr2 := "01234567"

      StrSwap( @cStr1, @cStr2 )
      ? cStr1                  // result: 01234567 compiler
      ? cStr2                  // result: xHarbour

   RETURN
```

StrToHex()

Converts a character string to a Hex string.

Syntax

```
StrToHex( <cString>, [<cSeparator>] ) --> cHexString
```

Arguments

<cString>

This is a character string to convert to hexadecimal notation.

<cSeparator>

Optionally, a character string can be specified that is inserted between the hexadecimal ASCII codes of each character in *<cString>*.

Return

The function returns a character string holding the passed value in hexadecimal notation.

Description

StrToHex() converts a character string to a Hex formatted character string by converting the ASCII codes of each charater in *<cString>* and collecting them in the return string. The reverse function is HexToStr() which accepts a Hex encoded character string.

Note: *<cSeparator>* is provided for formatting purposes of the returned string and to enhance readability. If it is used, the returned string cannot be decoded with HexToStr().

Info

See also:	CStr(), HexToNum(), HexToStr(), NumToHex(), Transform()
Category:	Character functions, Conversion functions, xHarbour extensions
Source:	rtl\hbhex2n.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates return values of StrToHex() without
// and with separating character

   PROCEDURE Main
      LOCAL cString := "xHarbour"

      ? StrToHex( cString )        // result: 78486172626F7572

      ? StrToHex( cString, "|" )   // result: 78|48|61|72|62|6F|75|72
   RETURN
```

StrTran()

Searches and replaces characters within a character string or memo field.

Syntax

```
StrTran( <cString>   , ;
         <cSubString>, ;
        [<cReplace>] , ;
        [<nStart>]   , ;
        [<nCount>]     ) --> cNewString
```

Arguments

<cString>

This parameter is the input string or memo field to search *<cSubString>* in.

<cSubString>

This is the character string to search for in *<cString>*.

<cReplace>

A character string *<cSubString>* is replaced with in *<cString>*. It defaults to an empty string (""), i.e. if *<cReplace>* is not specified, *<cSubString>* is removed from *<cString>*.

<nStart>

This is a numeric value indicating the first occurrence of *<cSubString>* to replace. The default value is 1.

<nCount>

This is a numeric value indicating the number of occurrences of *<cSubString>* to replace. If not specified, all occurrences of *<cSubString>* are replaced.

Return

The function returns a copy of *<String>* where *<cSubString>* is replaced with *<cReplace>*.

Description

StrTran() is a powerful search and replace function used to modify character strings. The function searches *<cSubString>* in the input string and replaces it with the replacement string. If *<nStart>* and *<nCount>* are not specified, all occurrances of *<cSubString>* are replaced. The search and replacement strings do not need to have the same length.

Info

See also:	At(), HardCR(), HB_RegExReplace(), MemoTran(), RAt(), StrDel(), Stuff(), SubStr()
Category:	Character functions
Source:	rtl\strtran.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows various possibilities for StrTran() usage.
// It begins with simple character search and replacements. At the
// end, StrTran() is used to create a macro expression that creates
// an array from a character string.

   PROCEDURE Main
      LOCAL cString := "a,BBB,cccc"
```

```
   LOCAL aArray

   ? StrTran( cString, "B" )              // result: a,,cccc

   ? StrTran( cString, "B", "b", 2 )      // result: a,Bbb,cccc

   ? StrTran( cString, "c", "C", 2 , 2 )  // result: a,BBB,cCCc

   aArray := &( '{"' + StrTran( cString, ',', '","' ) + '"}' )

   ? aArray[1]                            // result: a
   ? aArray[2]                            // result: BBB
   ? aArray[3]                            // result: cccc
RETURN
```

StrZero()

Converts a numeric value to a character string, padded with zeros.

Syntax

```
StrZero( <nNumber>, [<nLength>], [<nDecimals>] ) --> cString
```

Arguments

`<nNumber>`

A numeric value to convert to a character string.

`<nLength>`

An optional numeric value specifying the length of the return string, including sign and decimal places.

`<nDecimals>`

This is an optional numeric value indicating the number of decimal places to return.

Return

The function returns *<nNumber>* formatted as a character string, padded with zeros.

Description

StrZero() converts a numeric value to a character string. This is required when numeric values must be concatenated with character strings for formatted display, or when index keys containing Numeric and Character fields must be created.

If the length of the return string is specified too small, so that Int(*<nNumber>*) does not fit into the result, StrZero() returns a string filled with asterisks (*).

If only decimal places do not fit entirely, StrZero() rounds *<nNumber>* to the requested decimal places.

If both, *<nLength>* and *<nDecimals>*, are not specified, StrZero() obtains default values for both optional parameters as follows:

Formatting rules of StrZero()

Numeric value	Length of return string
Field variable	Field length plus decimals
Expressions/constants	Minimum of 10 digits plus decimals
Val()	Minimum of 3 digits
Day() and Month()	3 digits
Year()	5 digits
Recno()	7 digits

Info

See also: Str(), Transform(), Val()
Category: Conversion functions, Character functions
Source: rtl\strzero.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates formatting rules applied by StrZero()

   PROCEDURE Main
      LOCAL  nValue := 123.456

      ? StrZero( nValue )              // result: 0000000123.456

      ? StrZero( nValue, 1 )           // result: *
      ? StrZero( nValue, 2 )           // result: **
      ? StrZero( nValue, 3 )           // result: 123
      ? StrZero( nValue, 4 )           // result: 0123
      ? StrZero( nValue, 5 )           // result: 00123

      ? StrZero( nValue, 5, 1 )        // result: 123.5
      ? StrZero( nValue, 6, 2 )        // result: 123.46
      ? StrZero( nValue, 7, 3 )        // result: 123.456
      ? StrZero( nValue, 8, 4 )        // result: 123.4560

      nValue := - 123.456
      ? StrZero( nValue, 10, 0 )       // result: -000000123
      ? StrZero( nValue, 10, 1 )       // result: -0000123.5
      ? StrZero( nValue, 10, 2 )       // result: -000123.46
      ? StrZero( nValue, 10, 3 )       // result: -00123.456
      ? StrZero( nValue, 10, 4 )       // result: -0123.4560
   RETURN
```

Stuff()

Deletes and/or inserts characters in a string.

Syntax

```
Stuff( <cString>, <nStart>, <nDelete>, <cInsert> ) --> cNewString
```

Arguments

<cString>

This parameter is the input string to modify.

<nStart>

This is a numeric value specifying the position of the first character in *<cString>* to begin the operation with.

<nDelete>

This is a numeric value specifying the number of characters to delete, beginning at position *<nStart>*, before *<cInsert>* is inserted into *<cString>*.

<cInsert>

This is a character string to insert into *<cString>* at position *<nStart>*. Its length can be smaller or larger than *<nDelete>*.

Return

The function returns a modified copy of *<cString>*.

Description

Stuff() modifies an input string by first deleting *<nDelete>* characters and then inserting the replacement string *<cInsert>*. Since *<nDelete>* can be in the range of 0 to Len(*<cString>*) and *<cInsert>* can be any character string, including an empty string (""), the function can perform the operations Delete, Insert, Replace or a combination of them.

Unlike function StrTran(), which searches a substring, Stuff() uses a numeric start position to perform the character string operation.

Info

See also:	At(), HB_RegExReplace(), Left(), RAt(), Right(), StrTran(), SubStr()
Category:	Character functions
Source:	rtl\stuff.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates the different character string operations
// that can be performed with Stuff()

   PROCEDURE Main
      LOCAL cString := "1234567"

      // Delete
      ? Stuff( cString, 3, 2, "" )        // result: 12567

      // Insert
      ? Stuff( cString, 3, 0, "abc" )     // result: 12abc34567
```

```
   // Replace
   ? Stuff( cString, 3, 2, "ab" )    // result: 12ab567

   // Replace and delete
   ? Stuff( cString, 3, 4, "ab" )    // result: 12ab7

   // Replace and insert
   ? Stuff( cString, 3, 2, "abcde" ) // result: 12abcde567

   // Replace and delete rest
   ? Stuff( cString, 3, 6, "abc" )   // result: 12abc

RETURN
```

Subscribe()

Subscribes for notifications on a Mutex.

Syntax

```
Subscribe( <pMutexHandle>   , ;
           <nWaitMilliSecs>, ;
           [@<lNotified>]    ) --> xReturn
```

Arguments

<pMutexHandle>

This is the handle of the Mutex to subscribe, or listen, to. A mutex handle is returned by HB_MutexCreate().

<nWaitMilliSecs>

A numeric value specifying the maximum period of time to wait before the function returns when there is no notification on the Mutex. The unit for this value is 1/1000th of a second. If *<nWaitMilliSecs>* is omitted, the current thread waits infinitely for a notification, i.e. the function does not return until a notification is received.

@<lNotified>

When this parameter is passed by reference, it is assigned .T. (true) or .F. (false). True is assigned, when the thread executing Subscribe() is notified within the timeout period of *<nWaitMilliSecs>* from the thread that has locked the Mutex *<pMutexHandle>*.

Return

The function returns NIL, unless the thread which calls Notify() on the Mutex passes a second parameter to Notify(). This second parameter is returned by Subscribe().

Description

Function Subscribe() is the counterpart of Notify(), or NotifyAll(), respectively. The functions are used to control concurrent program execution between two ore more threads. The functions Notify() and Subscribe() can only be called in two different threads and require a common Mutex to operate on.

When a thread "A" calls SubScribe() with the Mutex *<pMutexHandle>*, this thread "A" is suspended for a time period of *<nWaitMilliSecs>*. Specifying nothing for the timeout period suspends thread "A" until another thread "B" calls Notify().

When thread "B" calls Notify() or NotifyAll() during the time period of *<nWaitMilliSecs>* on the same Mutex *<pMutexHandle>*, thread "A" resumes program execution.

As a result, function Subscribe() puts the **current** thread on hold until a second thread notifies the Mutex *<pMutexHandle>*, or the timeout period has expired. The timeout never expires when *<nWaitMillisecs>* is omitted, i.e. the current thread resumes only if the Mutex is notified by a second thread.

The parameter *<lNotified>* must be passed by reference to detect if the current thread resumes as a result of a notification by a second thread (*<lNotified>*==.T.) or due to the expiration of the timeout period of *<nWaitMilliSec>* (*<lNotified>*==.F.).

When thread "A" calls Subscribe() and thread "B" calls Notify(), the second parameter passed to Notify() in thread "B" defines the return value of Subscribe() in thread "A". That is, thread "B" can pass a value to thread "A" via the Subscribe()/Notify() protocol. All that is required is a Mutex which is subscribed to in thread "A", and notified in thread "B".

Info

See also: GetCurrentThread(), GetThreadID(), HB_MutexCreate(), HB_MutexLock(), Notify(), NotifyAll(), StartThread(), SubscribeNow()
Category: Multi-threading functions, Mutex functions, xHarbour extensions
Source: vm\thread.c
LIB: xhbmt.lib
DLL: xhbmtdll.dll

SubscribeNow()

Subscribes for notifications on a Mutex and discards pending notifications.

Syntax

```
SubscribeNow( <pMutexHandle>  , ;
              <nWaitMilliSecs>, ;
              [@<lNotified>]    ) --> xReturn
```

Arguments

<pMutexHandle>

This is the handle of the Mutex to subscribe, or listen, to. A mutex handle is returned by HB_MutexCreate().

<nWaitMilliSecs>

A numeric value specifying the maximum period of time to wait before the function returns when there is no notification on the Mutex. The unit for this value is 1/1000th of a second. If *<nWaitMilliSecs>* is omitted, the current thread waits infinitely for a notification, i.e. the function does not return until a notification is received.

@<lNotified>

When this parameter is passed by reference, it is assigned .T. (true) or .F. (false). True is assigned, when the thread executing SubscribeNow() is notified within the timeout period of *<nWaitMilliSecs>* from the thread that has locked the Mutex *<pMutexHandle>*.

Return

The function returns NIL, unless the thread which calls Notify() on the Mutex passes a second parameter to Notify(). This second parameter of Notify() is returned by SubscribeNow().

Description

Function SubscribeNow() is used in the same way as Subscribe(). The only difference is that SubscribeNow() voids all possibly pending notifications that may have been issued on *<pMutexHandle>* while SubscribeNow() is being executed. This guarantees that the current thread is put on hold until another thread notifies the Mutex **after** SubscribeNow() has returned. Refer to Subscribe() for detailed information on the SubScribe()/Notify() protocol.

Info

See also:	GetCurrentThread(), GetThreadID(), HB_MutexCreate(), HB_MutexLock(), Notify(), StartThread(), Subscribe()
Category:	Multi-threading functions, Mutex functions, xHarbour extensions
Source:	vm\thread.c
LIB:	xhbmt.lib
DLL:	xhbmtdll.dll

SubStr()

Extracts a substring from a character string.

Syntax

```
SubStr( <cString>, <nStart>, [<nCount>] ) --> cSubstring
```

Arguments

<cString>

This parameter is the input string to extract a substring from.

<nStart>

This is a numeric value specifying the first character of *<cString>* to include in the extracted string. If *<nStart>* is a positive number, the function extracts from the beginning, or the left side, of *<cString>*. If *<nStart>* is a negative number, the function extracts from the end, or the right side, of *<cString>*.

<nCount>

This is a numeric value specifying the number of characters to extract, beginning at position *<nStart>*. If omitted, all characters from *<nStart>* to the end of *<cString>* are extracted.

Return

The function returns the extracted character string.

Description

SubStr() is a versatile function for extracting parts of a character string beginning at the position *<nStart>*. It is similar to functions Left() and Right(), but can extract a substring from the middle of the input string. SubStr() is often used in conjunction with At() and RAt() to find the starting position of the substring to extract.

Info

See also:	At(), HB_ATokens(), HB_RegEx(), Left(), RAt(), Right(), Str(), StrTran(), Stuff()
Category:	Character functions
Source:	rtl\substr.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates extraction of substrings and how
// SubStr() relates to Left() and Right().

   PROCEDURE Main
      LOCAL cString := "www.xHarbour.com"

      ? SubStr( cString, 5 )           // result: xHarbour.com
      ? SubStr( cString, 5, 8 )        // result: xHarbour

      ? SubStr( cString, 1, 3 )        // result: www
      ? Left( cString, 3 )             // result: www

      ? SubStr( cString, -3, 3 )       // result: com
      ? Right( cString, 3 )            // result: com
   RETURN
```

SX_Decrypt()

Decrypts an encrypted character string.

Syntax

```
Sx_Decrypt( <cEncryptedString>, <cPassword> ) --> cString
```

Arguments

<cEncryptedString>

This is a previously encrypted character string to decrypt.

<cPassword>

This is a character string holding the password used for encryption.

Return

The function returns the decrypted character string.

Description

Function SX_Decrypt() decrypts a character string previously encrypted with function SX_Encrypt(). A successful decryption requires the same password *<cPassword>* as used for encrypting the unencrypted string.

Info

See also:	HB_Crypt(), HB_Decrypt(), SX_Encrypt()
Category:	Character functions, Conversion functions, Six driver
Source:	rdd\hbsix\sxcrypt.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates encryption and decryption of a
// character string.

   PROCEDURE Main
      LOCAL cString   := "xHarbour compiler"
      LOCAL cPassWord := "AB0187"
      LOCAL cCipher

      ? cCipher := SX_Encrypt( cString, cPassWord )

      ? cString := SX_Decrypt( cCipher, cPassWord )

   RETURN
```

SX_DtoP()

Converts a Date value into a 3-byte character string.

Syntax

```
SS_DtoP( <dDate> ) --> cPackedDate
```

Arguments

<dDate>

This is an expression returning a value of data type Date.

Return

The function returns the packed Date value as a binary character string.

Description

Function SX_DtoP() is used to compress Date values from 8 bytes to a binary character string of 3-bytes, thus reducing storage space in index or database files for Dates. Function SX_PtoD() converts a packed Date value back to its original data type.

Info

See also:	SX_PtoD()
Category:	Conversion functions, Date and time, Six driver
Source:	rdd\hbsix\sxdate.c
LIB:	xhb.lib
DLL:	xhbdll.dll

SX_Encrypt()

Encrypts a character string.

Syntax

```
SX_Encrypt( <cString>, <Password> ) --> cEncryptedString
```

Arguments

`<cString>`

This is a character string to encrypt.

`<cPassword>`

This is a character string holding the password used for encryption.

Return

The function returns the encrypted character string.

Description

Function SX_Encrypt() encrypts a character string using the password <*cPassword*>. The returned encrypted string can be decrypted by passing it to function SX_Decrypt() along with the same password.

Info

See also:	HB_Crypt(), HB_Decrypt(), SX_Decrypt()
Category:	Character functions, Conversion functions, Six driver
Source:	rdd\hbsix\sxcrypt.c
LIB:	xhb.lib
DLL:	xhbdll.dll

SX_FCompress()

Compresses a file.

Syntax

```
SX_FCompress( <cSourceFile>, <cTargetFile> ) --> lSuccess
```

Arguments

<cSourceFile>

This is the name of the source file to compress. It must include path and file extension.

<cTargetFile>

This is the name of the target file to create. It must include path and file extension.

Return

The function returns .T. (true) when the compressed target file is successfully created, otherwise .F. (false) is returned.

Description

Function SX_FCompress() reads the entire file *<cSourceFile>* and writes its compressed contents into the new file *<cTargetFile>*. If *<cTargetFile>* exists already, it is overwritten without warning.

Info

See also:	HB_Compress(), SX_FDecompress()
Category:	File functions, Six driver
Source:	rdd\hbsix\sxcompr.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example compresses a MAP file to demonstrate the size of
// a compressed file.

   PROCEDURE Main
      LOCAL cSource := "sx_Fcompress.map"
      LOCAL cTarget := "sx_Fcompress.cmp"

      ? FileSize( cSource )                // result: 246331

      ? SX_FCompress( cSource, cTarget )   // result: .T.

      ? FileSize( cTarget )                // result:  63286

   RETURN
```

SX_FDecompress()

Decompresses a compressed file.

Syntax

```
Sx_FDecompress( <cCompressed>, <cUncompressed ) --> lSuccess
```

Arguments

<cCompressed>

This is the name of file previously compressed with SX_FCompress(). It must include path and file extension.

<cUncompressed>

This is the name of the file to create. It must include path and file extension.

Return

The function returns .T. (true) when the uncompressed file is successfully created, otherwise .F. (false) is returned.

Description

Function SX_FDeompress() reads the entire file <*cCompressed*> and writes its uncompressed contents into the new file <*cUncompressed*>. If this file exists already, it is overwritten without warning.

Info

See also:	HB_Compress(), SX_FCompress()
Category:	File functions, Six driver
Source:	rdd\hbsix\sxcompr.c
LIB:	xhb.lib
DLL:	xhbdll.dll

SX_PtoD()

Unpacks a packed 3-byte date value.

Syntax

```
Sx_PtoD( <cPackedDate> ) --> dDate
```

Arguments

<cPackedDate>

This is a packed Date value as returned from SX_DtoP().

Return

The function converts the packed Date string and returns the original Date value.

Description

Function SX_PtoD() is used to uncompress a packed Date string of three bytes as returned from SX_DtoP() and convert it to the original Date value.

Info

See also:	SX_DtoP()
Category:	Conversion functions, Date and time, Six driver
Source:	rdd\hbsix\sxdate.c
LIB:	xhb.lib
DLL:	xhbdll.dll

TabExpand()

Replaces a tab with a specified number of another character.

Syntax

```
TabExpand( <cString>   , ;
          [<nTabWidth>], ;
          [<xFillChar>]  ) --> cResultString
```

Arguments

<cString>

This is a character string to expand Tab characters in (Chr(9)).

<nTabWidth>

This parameter defaults to 8 so that Tabs are replaced by a character string of eight *<xFillChar>*.

<xFillChar> [Chr(32)]

Either a single character or its numeric ASCII code can be specified as replacement for Tab characters. The default is a space character (chr(32)).

Return

The function replaces each Tab character found in *<cString>* with a string consisting of *<nTabWith>* times *<xFillChar>* and returns the modified string.

Info

See also:	TabPack()
Category:	CT:String manipulation, Character functions
Source:	ct\tab.c
LIB:	xhb.lib
DLL:	xhbdll.dll

TabPack()

Inserts a Tab (Chr(9)) for multiple occurrances of a character.

Syntax

```
TabPack( <cString>   , ;
        [<nTabWidth>], ;
        [<xFillChar>]  ) --> cResult
```

Arguments

`<cString>`

This is the character string to process.

`<nTabWidth>`

This numeric parameter defaults to 8, the default width of a Tab character (chr(9)).

`<xFillChar>`

Either a single character or its numeric ASCII code can be specified to search for. The default is a space character (chr(32)).

Return

The function seraches multiple occurrances of *<xFillChar>* and replaces them with a Tab (Chr(9)) according to the specified tab width. The packed string is returned.

Info

See also:	TabExpand()
Category:	CT:String manipulation, Character functions
Source:	ct\tab.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Tan()

Calculates the tangent.

Syntax

```
Tan( <nAngle> ) --> nTangent
```

Arguments

<nAngle>

A numeric value specifies the angle as a fraction (or multiple) of Pi().

Return

The function returns the tangent in the range of plus and minus Infinity().

Info

See also: Cos(), Cot(), DtoR(), RtoD(), Sin(), TanH()
Category: CT:Math, Mathematical functions, Trigonometric functions
Source: ct\trig.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example display results of Tan(). Note that there is a numeric overflow.

   PROCEDURE Main
      ? Str( Tan( -Pi()*1.0 ), 18, 15)  // result:  0.000000000000000
      ? Str( Tan( -Pi()*0.5 ), 18, 15)  // result: ******************
      ? Str( Tan( -Pi()*0.3 ), 18, 15)  // result: -1.376381920471174
      ? Str( Tan(  Pi()*0.0 ), 18, 15)  // result:  0.000000000000000
      ? Str( Tan(  Pi()*0.3 ), 18, 15)  // result:  1.376381920471174
      ? Str( Tan(  Pi()*0.5 ), 18, 15)  // result: ******************
      ? Str( Tan(  Pi()*1.0 ), 18, 15)  // result:  0.000000000000000
   RETURN
```

TanH()

Calculates the hyperbolic tangent.

Syntax

```
TanH( <nRadians> ) --> nHyperbolicTangent
```

Arguments

<nRadians>

A numeric value specifying the angle value in radians.

Return

The function returns the hyperbolic tangent in the range between -1 and +1.

Info

See also: Cos(), Cot(), DtoR(), RtoD(), Sin(), Tan()
Category: CT:Math, Mathematical functions, Trigonometric functions, xHarbour extensions
Source: ct\trig.c
LIB: xhb.lib
DLL: xhbdll.dll

TBMouse()

Moves the browse cursor to the mouse pointer.

Syntax

```
TBMouse( <oTBrowse>, <nMouseRow>, <nMouseCol> ) --> nHandled
```

Arguments

`<oTBrowse>`

This parameter must be a TBrowse() object.

`<nMouseRow>`

A numeric value between 0 and MaxRow() specifying the row position of the mouse cursor. It can be queried using MRow().

`<nMouseCol>`

A numeric value between 0 and MaxCol() specifying the columnn position of the mouse cursor. It can be queried using MCol().

Return

The function returns 0 when the browse cursor was successfully moved to the screen coordinates passed for the mouse pointer. Otherwise, the return value is 1.

Description

TBMouse() is a utiliy function for implementing "mouse awareness" for TBrowse objects. When the mouse pointer is located within the data area of a TBrowse object, the function calls navigation methods of the object until the browse cursor is located underneath the screen coordinates specified with *<nMouseRow>* and *<nMouseCol>*.

A call to TBMouse() is standard behavior for TBrowse():applyKey().

Info

See also:	MCol(), MRow(), TBrowse()
Category:	Mouse functions, xHarbour extensions
Source:	rtl\tbrowse.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

TBrowseDB()

Creates a new TBrowse object to be used with a database.

Syntax

```
TBrowseDB( [<nTop>]    , ;
           [<nLeft>]   , ;
           [<nBottom>], ;
           [<nRight>]    ) --> oTBrowse
```

Arguments

`<nTop> and <nLeft>`

Numeric values indicating the screen coordinates for the upper left corner of the TBrowse() window. The default value for both parameters is zero.

`<nBottom>`

Numeric value indicating the bottom row screen coordinate. It defaults to MaxRow().

`<nRight>`

Numeric value indicating the right column screen coordinate. It defaults to MaxCol().

Return

TBrowseDB() returns new TBrowse object with the specified coordinate and a default :skipBlock, :goTopBlock and :goBottomBlock to browse a database.

Description

TBrowseDB() is a quick way to create a TBrowse object along with the minimal support needed to browse a database. Note that the returned TBrowse object contains no TBColumn objects. They must be added for each field to display in the browse window.

Info

See also:	Browse(), DbEdit(), TBColumn(), TBrowse(), TBrowseNew()
Category:	Object functions
Source:	rtl\tbrowse.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example builds a simple TBrowse object for a database

   #include "TBrowse.ch"

   PROCEDURE Main
      LOCAL oTBrowse, aFields, cField, nKey

      USE Customer

      aFields := Array( FCount() )
      AEval( aFields, {|x,i| aFields[i] := FieldName(i) } )

      oTBrowse := TBrowseDB()

      WITH OBJECT oTBrowse
         FOR EACH cField IN aFields
```

```
            :addColumn( TBColumnNew( cField, FieldBlock( cField ) ) )
         NEXT
      END

      DO WHILE .T.
         oTBrowse:forceStable()
         nKey := Inkey(0)

         IF oTBrowse:applyKey( nKey ) == TBR_EXIT
            EXIT
         ENDIF
      ENDDO

      CLOSE Customer
   RETURN
```

TBrowseNew()

Creates a new TBrowse object.

Syntax

```
TBrowseNew( [<nTop>]    , ;
            [<nLeft>]   , ;
            [<nBottom>], ;
            [<nRight>]    ) --> oTBrowse
```

Arguments

<nTop>

This is the numeric screen coordinate for the top row of the browse display. It defaults to zero and is assigned to the instance variable :nTop.

<nLeft>

This is the numeric screen coordinate for the left column of the browse display. It defaults to zero and is assigned to the instance variable :nLeft.

<nBottom>

This is the numeric screen coordinate for the bottom row of the browse display. It defaults to MaxRow() and is assigned to the instance variable :nBottom.

<nRight>

This is the numeric screen coordinate for the right column of the browse display. It defaults to MaxCol() and is assigned to the instance variable :nRight.

Return

Function TBrowseNew() returns a new, initialized TBrowse object.

Description

Function TBrowseNew() is the functional equivalent of TBrowse():new(). Refer to the description of the Tbrowse object.

Info

See also:	TBrowse()
Category:	Object functions
Source:	rtl\tbrowse.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

ThreadSleep()

Puts a thread to sleep.

Syntax

```
ThreadSleep( <nMilliSeconds>, [<lNoCpu>] ) --> NIL
```

Arguments

`<nMilliSeconds>`

This is a numeric value indicating the period of time to suspend the current thread. The unit is 1/1000th of a second.

`<lNoCpu>`

The parameter defaults to .F. (false) so that the thread resumes after <*nMilliSeconds*> have elapsed. When set to .T. (true), the thread resumes when a new event is placed into the thread's message queue on the operating system level. During this stage, the thread consumes no CPU resources.

Return

The function returns always NIL.

Description

Function ThreadSleep() is used to suspend the current thread for a defined period of time. This is useful, for example, when a thread should check some status variables periodically and process data only if the status has changed.

If the second parameter is set to .T. (true), the thread ignores <*nMilliSeconds*> an resumes only after a message is put into the thread's message queue by the operating system.

Note: this function is also available in single threaded applications.

Info

See also:	Inkey(), SecondsSleep(), StartThread(), WAIT
Category:	Multi-threading functions, xHarbour extensions
Source:	vm\thread.c
LIB:	xhbmt.lib
DLL:	xhbmtdll.dll

Throw()

Throws an exception.

Syntax

```
Throw( <oError> ) --> NIL
```

Arguments

<oError>

This parameter must be an Error() object.

Return

The return value is NIL.

Description

Function Throw() is part of xHarbour's error handling system and is used in conjunction with a TRY...CATCH statement. The error object passed to Throw() is forwarded to the CATCH option, if present. There, the information stored in the error object can be evaluated for error handling.

Info

See also: BEGIN SEQUENCE, Error(), ErrorBlock(), ErrorNew(), TRY...CATCH
Category: Error functions, Debug functions, xHarbour extensions
Source: vm\throw.c
LIB: xhb.lib
DLL: xhbdll.dll

THtmlCleanup()

Releases memory tables required for HTML classes.

Syntax

```
THtmlCleanup() --> lSuccess
```

Return

The function returns always .T. (true).

Description

Function THtmlCleanup() releases all memory lookup tables required for the THtmlDocument() class. The function should be called, when no more THtmlDocument and THtmlNode objects exist.

Info

See also:	THtmlDocument(), THtmlInit(), THtmlNode()
Category:	HTML functions, xHarbour extensions
Source:	tip\thtml.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

THtmlInit()

Initializes memory tables required for HTML classes.

Syntax

```
THtmlInit() --> lSuccess
```

Return

The function returns always .T. (true).

Description

Function THtmlInit() initializes all memory lookup tables required for the THtmlDocument() class. It is automatically called when a THtmlDocument object is created. Call THtmlCleanup() to release the HTML memory resources when no more THtmlDocument and THtmlNode objects exist.

Info

See also:	THtmlCleanup(), THtmlDocument(), THtmlNode()
Category:	HTML functions, xHarbour extensions
Source:	tip\thtml.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

THtmlIsValid()

Validates a HTML tag name and attribute.

Syntax

```
THtmlIsValid( <cTagName>, [<cAttrName>] ) --> lIsValid
```

Arguments

`<cTagName>`

This is a character string holding the tag name of an HTML tag.

`<cAttrName>`

This is an optional character string holding the attribute name of an HTML tag

Return

The function returns .T. (true) when *<cTagName>* is a valid HTML tag name. When *<cAttrName>* is passed, the function checks if the HTML tag has this attribute and returns .T. (true) if the attribute is valid. The return value is .F. (false), when *<cTagName>* or *<cAttrName>* are invalid.

Info

See also: THtmlCleanup(), THtmlDocument(), THtmlInit(), THtmlNode()
Category: HTML functions, xHarbour extensions
Source: tip\thtml.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays results of THtmlIsValid()

   PROCEDURE Main
      THtmlInit()

      ? THtmlIsValid( "html" )              // result: .T.

      ? THtmlIsValid( "xml" )               // result: .F.

      ? THtmlIsValid( "meta", "content" )   // result: .T.

      ? THtmlIsValid( "table", "content" ) // result: .F.

      ? THtmlIsValid( "table", "border" )   // result: .T.

      THtmlCleanup()
   RETURN
```

Time()

Retrieves the system time as a formatted character string.

Syntax

```
Time() --> cHHMMSS
```

Return

The function returns a character string containing the system time formatted as hh:mm:ss.

Description

Time() is used to obtain a time formatted character string containing hours, minutes and seconds of a 24h clock.

Info

See also: AmPm(), Date(), Seconds(), SubStr()
Category: Date and time
Source: rtl\dateshb.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example shows the results of Time() and how to extract
// hour, minutes, and seconds

   PROCEDURE Main
      LOCAL cTime := Time()
      LOCAL cHour, cMin, cSec

      ? cTime                          // Result: 13:27:47
      cSec  := Right ( cTime, 2 )
      cMin  := SubStr( cTime, 3, 2 )
      cHour := Left  ( cTime, 2 )

      ? Val( cHour ) * 3600 + Val( cMin ) * 60 + Val( cSec )
                                       // result: 48467.00
   RETURN
```

TimeToSec()

Calculates the number of seconds since midnight.

Syntax

```
TimeToSec( [<cTime>] ) --> nSeconds
```

Arguments

<cTime>

This is a character string formatted as HH:MM:SS or HH:MM:SS:ss. It holds a time string based on a 24h clock.

Return

The function returns a numeric value representing the number of seconds corresponding to *<cTime>*. If no parameter is passed, the return value is determined by Seconds().

Info

See also:	Seconds(), SecToTime(), Time()
Category:	CT:DateTime, Date and time
Source:	ct\cttime.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

TimeValid()

Checks if a character string is a valid time string.

Syntax

```
TimeValid( <cTime> ) --> lIsTimeString
```

Arguments

<cTime>

This is a time formatted character string as returned by Time().

Return

The function returns .T. (true) if a character string in the format hh:mm:ss is passed. Otherwise .F. (false) is returned.

Info

See also:	SetTime(), Time()
Category:	CT:DateTime, Date and time
Source:	ct\dattime3.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Token()

Retrieves the n-th token from a string.

Syntax

```
Token( <cString>         , ;
       [<cDelimiter>]    , ;
       [<nTokenPos>]     , ;
       [<nSkipWidth>]    , ;
       [@<cPreTokenSep>] , ;
       [@<cPostTokenSep>]  ) --> cToken
```

Arguments

`<cString>`

This is a character string to find a token in.

`<cDelimiter>`

This character string holds a list of characters recognized as delimiters between tokens. The default list of delimiters consist of non-printable characters having the ASCII codes 0, 9, 10, 13, 26, 32, 138 and 141, and the following punctuation characters: ,.;:!?/\<>()!HSH&%+-*

`<nTokenPos>`

A numeric value indicating the ordinal position of the token to retrieve. It defaults to the last token.

`<nSkipWidth>`

This optional numeric value defaults to zero. This causes the function to find empty tokens. To suppress this behavior, set *<nSkipWidth>* to 1.

`@<cPreTokenSep> and @<cPostTokenSep>`

If passed by reference, these parameters receive the pre- and post delimiters of the found token, making an extra call to TokenSep() obsolete.

Return

The function returns the token found at *<nTokenPos>* in a character string. When no more tokens can be found, the return value is a null string ("").

Note: Token() does not use the tokenizer environment. Therefore, TokenInit() does not need to be called. However, when large strings are tokenized, it is recommended to take advantage of the tokenizer environment for achieving best performance.

Info

See also:	AtToken(), HB_ATokens(), NumToken(), TokenInit(), TokenLower(), TokenSep(), TokenUpper()
Category:	CT:String manipulation, Character functions, Token functions
Source:	ct\token1.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example extracts all directories from the PATH
// environment variable.

   PROCEDURE Main
```

```
    LOCAL cPath := GetEnv( "PATH" )
    LOCAL cDirectory, nCount := 0

    // Last token
    ? Token( cPath, ";" )

    // All tokens
    DO WHILE .T.
       cDirectory := Token( cPath, ";", ++nCount )

       IF Empty( cDirectory )
          EXIT
       ENDIF

       ? nCount, cDirectory
    ENDDO

 RETURN
```

TokenAt()

Returns the start and end position of a token.

Syntax

```
TokenAt( [<lNextDelimiter>], ;
         [<nCount>]         , ;
         [@<cTokenEnv>]      ) --> nPosition
```

Arguments

<lNextDelimiter>

This parameter defaults to .F. (false) which causes the function to return the start position of the token found with TokenNext(). When set to .T. (true) the function returns the position of the next delimiter following the found token.

<nTokenPos>

Optionally, a numeric value can be passed indicating the ordinal position of the token to retrieve. By default, the position of the current token found with TokenNext() is returned.

@<cTokenEnv>

Optionally, a character string holding a local tokenizer environment can be passed. It is obtained via the fourth parameter of TokenInit(). *<cTokenEnv>* must be passed by reference. If *<cTokenEnv>* is omitted, the function uses the global tokenizer environment.

Return

The function returns either the position of the token found with TokenNext() (default) or the position of the following delimiter as a numeric value.

Info

See also:	TokenInit(), TokenNext()
Category:	CT:String manipulation, Character functions, Token functions
Source:	ct\token2.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example determines the start and end position of the second
// token in a string and extracts it with function SubStr().

    PROCEDURE Main
       LOCAL cIPAddr := "189.88.3.256"
       LOCAL cTokenEnv, nStart, nEnd

       TokenInit( cIPAddr, ".", @cTokenEnv )

       ? nStart := TokenAt( .F., 2, @cTokenEnv )  // result: 5
       ? nEnd   := TokenAt( .T., 2, @cTokenEnv )  // result: 7

       ? SubStr( cIPAddr, nStart, nEnd-nStart )   // result: 88
    RETURN
```

TokenEnd()

Tests if tokens can still be found with TokenNext().

Syntax

```
TokenEnd( [@<cTokenEnv>] ) --> lTokenEnd
```

Arguments

@<cTokenEnv>

Optionally, a character string holding a local tokenizer environment can be passed. It is obtained via the fourth parameter of TokenInit(). *<cTokenEnv>* must be passed by reference. If *<cTokenEnv>* is omitted, the function uses the global tokenizer environment.

Return

The function returns .T. (true) when a new call to TokenNext() returns a token, otherwise .F. (false) is returned.

Info

See also: TokenInit(), TokenNext()
Category: CT:String manipulation, Character functions, Token functions
Source: ct\token2.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example is a typical usage scenario for TokenEnd(). It delivers
// the termination condition for extracting all tokens from a character
// string within a DO WHILE loop.

   PROCEDURE Main
      LOCAL cPath  := GetEnv( "PATH" )
      LOCAL nCount := 0

      TokenInit( @cPath, ";" )

      DO WHILE .NOT. TokenEnd()
         ? ++nCount, TokenNext( @cPath )
      ENDDO

      TokenExit()
   RETURN
```

TokenExit()

Releases memory resources of the global tokenizer environment.

Syntax

```
TokenExit() --> lSuccess
```

Return

TokenExit() must be executed when TokenInit() is called without a fourth parameter. The function returns .T. (true) when the memory resources of the global tokenizer environment are successfully released, otherwise .F. (false) is returned.

Info

See also:	TokenInit(), TokenNext()
Category:	CT:String manipulation, Character functions, Token functions, xHarbour extensions
Source:	ct\token2.c
LIB:	xhb.lib
DLL:	xhbdll.dll

TokenInit()

Initializes the environment for the incremental tokenizer.

Syntax

```
TokenInit( [@<cString>]  , ;
           [<cDelimiter>], ;
           [<nSkipWidth>], ;
           [@<cTokenEnv>]  ) --> lSuccess
```

Arguments

@<cString>

This is a character string to be tokenized. Tokens are retrieved from the string with TokenNext(). Note that *<cString>* must be passed by reference. If *<cString>* is omitted, the global tokenizer environment is reset.

<cDelimiter>

This character string holds a list of characters recognized as delimiters between tokens. The default list of delimiters consist of non-printable characters having the ASCII codes 0, 9, 10, 13, 26, 32, 138 and 141, and the following punctuation characters: ,.;:!?/\<>()!HSH&%+-*

<nSkipWidth>

This optional numeric value defaults to zero. This causes the incremental tokenizer to find empty tokens. To suppress this behavior, set *<nSkipWidth>* to the length of the largest delimiter.

@<cTokenEnv>

If this parameter is passed by reference, it receives a character string holding a local environment for the incremental tokenizer. This character string must then be passed to other functions of the tokenizer, like TokenNext().

Return

The function returns .T. (true) when the environment for the incremental tokenizer is successfully initialized, otherwise .F. (false) is returned.

Description

TokenInit() initializes the environment of the incremental tokenizer of xHarbour. In contrast to the Clipper CA-Tools, xHarbour maintains one global tokenizer environment and any number of local tokenizer environments. The latter are created by passing a fourth parameter by reference to TokenInit(). *<cTokenEnv>* receives the local tokenizer environment. As a result, the functions SaveToken() and RestToken() become obsolete via the local tokenizer environment.

Note: when TokenInit() is called with *<cString>* only, and no local tokenizer environment is created by passing the fourth parameter by reference, the global tokenizer environment is initialized. The memory resources for the global tokenizer environment must be released afterwards with TokenExit().

Info

See also: HB_ATokens(), RestToken(), SaveToken(), TokenEnd(), TokenNext(), TokenExit()
Category: CT:String manipulation, Character functions, Token functions
Source: ct\token2.c
LIB: xhb.lib
DLL: xhbdll.dll

Examples

Using the global tokenizer environment

```
// The example calculates line and word count of a text file
// using the global tokenizer environment. To determine the word
// count, each line is tokenized in function WordCount(). To
// accomplish this, the global tokenizer environment is saved
// and restored.

   #define CRLF   Chr(13)+Chr(10)

   PROCEDURE Main
      LOCAL cText  := MemoRead( "Textfile.txt" )
      LOCAL cToken
      LOCAL aLines := {}
      LOCAL nLines := 0
      LOCAL nWords := 0

      // initialize global tokenizer environment
      TokenInit( @cText, CRLF, 2 )

      DO WHILE .NOT. TokenEnd()
         cToken := TokenNext( @cText )

         IF cToken == ""
            // one blank space is an empty line for AChoice()
            cToken := " "
         ENDIF

         nLines ++
         nWords += WordCount( @cToken )

         AAdd( aLines, cToken )
      ENDDO

      // release global tokenizer environment
      TokenExit()

      // display the text file
      AChoice( ,,,, aLines )

      CLS
      ? "Line count:", nLines
      ? "Word count:", nWords
   RETURN


   FUNCTION WordCount( cText )
      LOCAL cSave  := SaveToken()
      LOCAL nWords := 0

      TokenInit( @cText, " ,.!?" )
```

```
      DO WHILE .NOT. TokenEnd()
         TokenNext( @cText )
         nWords ++
      ENDDO

      RestToken( cSave )

   RETURN nWords
```

Using a local tokenizer environment

```
// This example does the same, but takes advantage of local tokenizer
// environments. The performance is about 20% better compared to the
// global tokenizer environment, since SaveToken() and RestToken() are
// not needed.

   #define CRLF   Chr(13)+Chr(10)

   PROCEDURE Main
      LOCAL cText  := MemoRead( "Textfile.txt" )
      LOCAL cToken, cTokenEnv
      LOCAL aLines := {}
      LOCAL nLines := 0
      LOCAL nWords := 0

      // initialize local tokenizer environment
      TokenInit( @cText, CRLF, 2, @cTokenEnv )

      DO WHILE .NOT. TokenEnd( @cTokenEnv )
         cToken := TokenNext( @cText, , @cTokenEnv )

         IF cToken == ""
            // one blank space is an empty line for AChoice()
            cToken := " "
         ENDIF

         nLines ++
         nWords += WordCount( @cToken )

         AAdd( aLines, cToken )
      ENDDO

      // display the text file
      AChoice( ,,,, aLines )

      CLS
      ? "Line count:", nLines
      ? "Word count:", nWords
   RETURN


   FUNCTION WordCount( cText )
      LOCAL cTokenEnv
      LOCAL nWords := 0

      TokenInit( @cText, " ,.!?", @cTokenEnv )

      DO WHILE .NOT. TokenEnd( @cTokenEnv )
         TokenNext( @cText, , @cTokenEnv )
         nWords ++
      ENDDO
```

```
RETURN nWords
```

TokenLower()

Changes the first character of tokens to lower case.

Syntax

```
TokenLower( <cString>    , ;
           [<cDelimiter>], ;
           [<nTokenPos>] , ;
           [<nSkipWidth>]  ) --> cResult
```

Arguments

<cString>

This is a character string to process.

<cDelimiter>

This character string holds a list of characters recognized as delimiters between tokens. The default list of delimiters consist of non-printable characters having the ASCII codes 0, 9, 10, 13, 26, 32, 138 and 141, and the following punctuation characters: ,.;:!?/\<>()!HSH&%+-*

<nTokenPos>

A numeric value indicating the ordinal position of the token to change. If not specified, all tokens are changed.

<nSkipWidth>

This optional numeric value defaults to zero. This causes the function to find empty tokens. To suppress this behavior, set *<nSkipWidth>* to 1.

Return

The function changes the first character of the specified token to lower case and returns the modified string.

Info

See also: AtToken(), CSetRef(), NumToken(), Token(), TokenUpper()
Category: CT:String manipulation, Character functions, Token functions
Source: ct\token1.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays results of TokenLower()

   PROCEDURE Main
      LOCAL cString := "This Is The XHarbour Compiler"

      ? TokenLower( cString )        // this is the xHarbour compiler

      ? TokenLower( cString, "X" )   // This Is The Xharbour Compiler

      ? TokenLower( cString, , 2 )   // This is The XHarbour Compiler

   RETURN
```

TokenNext()

Retrieves the next token from a string.

Syntax

```
TokenNext([@]<cString>    , ;
            [<nTokenPos>], ;
           [@<cTokenEnv>]  ) --> cToken
```

Arguments

<cString>

This is a character string to extract a token from. It is recommended to pass it by reference to achieve maximum performance.

<nTokenPos>

A numeric value indicating the ordinal position of the token to retrieve. It defaults to the current token. If omitted, the token counter of the tokenizer environment is incremented so that a subsequent call to TokenNext() retrieves the next token.

@<cTokenEnv>

Optionally, a character string holding a local tokenizer environment can be passed. It is obtained via the fourth parameter of TokenInit(). *<cTokenEnv>* must be passed by reference. If *<cTokenEnv>* is omitted, the function uses the global tokenizer environment.

Return

The function retrieves the next token from a character string prepared for the incremental tokenizer with TokenInit().

Info

See also:	RestToken(), SaveToken(), TokenAt(), TokenInit()
Category:	CT:String manipulation, Character functions, Token functions
Source:	ct\token2.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example lists all directories from the PATH environment variable

   PROCEDURE Main
      LOCAL cPath  := GetEnv( "PATH" )
      LOCAL nCount := 0

      TokenInit( @cPath, ";" )

      DO WHILE .NOT. TokenEnd()
         ? ++nCount, TokenNext( @cPath )
      ENDDO

      TokenExit()
   RETURN
```

TokenNum()

Returns the number of tokens in a tokenizer environment.

Syntax

```
Tokennum( [<@cTokenEnv>] ) --> nTokenCount
```

Arguments

`@<cTokenEnv>`

Optionally, a character string holding a local tokenizer environment can be passed. It is obtained via the fourth parameter of TokenInit(). *<cTokenEnv>* must be passed by reference. If *<cTokenEnv>* is omitted, the function uses the global tokenizer environment.

Return

The function returns the total number of tokens of the specified tokenizer environment as a numeric value. The tokenizer environment must be initialized with TokenInit().

Info

See also:	TokenEnd(), TokenInit(), TokenNext()
Category:	CT:String manipulation, Character functions, Token functions, xHarbour extensions
Source:	ct\token2.c
LIB:	xhb.lib
DLL:	xhbdll.dll

TokenSep()

Retrieves the separating characters of a token.

Syntax

```
TokenSep( [<lRightSeparator>] ) --> cSeparator
```

Arguments

<lRightSeparator>

This parameter defaults to .F. (false) causing the function to return the separator on the left side of the current token. When .T. (true) is passed, the function returns the right separaor.

Return

The function returns the left or right delimiting character of a token retrieved with function function Token(). It cannot be used with a tokenizer environment initialized with TokenInit().

Note: the function exists for compatibility reasons. It is superseeded by the 5th and 6th reference parameters of function Token().

Info

See also:	Token()
Category:	CT:String manipulation, Character functions, Token functions
Source:	ct\token1.c
LIB:	xhb.lib
DLL:	xhbdll.dll

TokenUpper()

Changes the first character of tokens to upper case.

Syntax

```
TokenLower( <cString>    , ;
           [<cDelimiter>], ;
           [<nTokenPos>] , ;
           [<nSkipWidth>]  ) --> cResult
```

Arguments

<cString>

This is a character string to process.

<cDelimiter>

This character string holds a list of characters recognized as delimiters between tokens. The default list of delimiters consist of non-printable characters having the ASCII codes 0, 9, 10, 13, 26, 32, 138 and 141, and the following punctuation characters: ,.;:!?/\<>()!HSH&%+-*

<nTokenPos>

A numeric value indicating the ordinal position of the token to change. If not specified, all tokens are changed.

<nSkipWidth>

This optional numeric value defaults to zero. This causes the function to find empty tokens. To suppress this behavior, set *<nSkipWidth>* to 1.

Return

The function changes the first character of the specified token to upper case and returns the modified string.

Info

See also: AtToken(), CSetRef(), NumToken(), Token(), TokenLower()
Category: CT:String manipulation, Character functions, Token functions
Source: ct\token1.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays results of TokenUpper()

   PROCEDURE Main1
      LOCAL cString := "this is the xharbour compiler"

      ? TokenUpper( cString )        // This Is The XHarbour Compiler

      ? TokenUpper( cString, "x" )   // this is the xHarbour compiler

      ? TokenUpper( cString, , 2 )   // this Is the xharbour compiler

   RETURN
```

Tone()

Sounds a speaker tone with specific tone frequency and duration.

Syntax

```
Tone( <nFrequency>, [<nDuration>] ) --> NIL
```

Arguments

<nFrequency>

This is a positive numeric value specifying the frequency of the tone.

<nDuration>

This is a positive numeric value specifying the duration of the tone in units of 1/18th of a second.

Return

The function returns always NIL.

Description

Tone() is a compatibility function used to produce a speaker tone of defined frequency and duration.

Info

See also: Chr(), SET BELL
Category: Environment functions
Source: rtl\tone.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example shows a user-defined function that alerts a user
// when an error condition occurs.

   FUNCTION ErrorBeep()
      Tone( 1000, 1 )
      Tone( 1000, 2 )
   RETURN NIL
```

TraceLog()

Traces and logs the contents of one or more variables.

Syntax

```
TraceLog( <xVariables,...> ) --> lTrue
```

Arguments

<xVariables>

This is a comma separated list of variables to trace during program execution.

Return

The function returns .T. (true).

Description

The TraceLog() function traces and logs the contents of one or more variables in the trace log file. Each time TraceLog() is called, a new entry is added to the log file. To suppress output of the function SET TRACE must be set to OFF.

Info

See also: AltD(), HB_BldLogMsg(), INIT LOG, SET ERRORLOG, SET TRACE
Category: Debug functions, Log functions, xHarbour extensions
Source: rtl\traceprg.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// This example logs the contents of three variables in the trace.log file. The
// log contains type and value of the variables as well as the line number,
// procedure and source file where the trace was executed.

   PROCEDURE Main()
      LOCAL xValue1 := 1
      LOCAL xValue2 := 1
      LOCAL xValue3 := 1

      TraceLog( xValue1, xValue2, xValue3 )

      xValue1 := 2
      xValue2 := 33
      xValue3 := xValue1 + xValue2
      TraceLog( xValue1, xValue2, xValue3 )

      xValue1 := Str(++xValue1)
      xValue2 := 33
      xValue3 := xValue2++

      TraceLog( xValue1, xValue2, xValue3 )

      MemoEdit( Memoread( "Trace.log" ) )
   RETURN
```

Transform()

Converts values to a PICTURE formatted character string.

Syntax

```
Transform( <xValue>, <cPicture> ) --> cFormattedString
```

Arguments

<xValue>

This is a value of data type Character, Date, Logic, Memo or Numeric to be formatted.

<cPicture>

This is a PICTURE formatting string defining the formatting rules (see below).

Return

The function returns a character string holding the formatted value of *<xValue>*.

Description

Transform() is used to convert values of simple data types (C,D,L,M,N) to formatted character strings. Formatting rules are defined with the second parameter *<cPicture>*. This picture string may consist of characters defining a picture function, or characters defining a picture mask, or both. If picture function and mask are present in the picture string, the picture function must be first and the mask characters must be separated from the picture function by a single blank space.

Picture function

A picture function specifies formatting rules for the entire output string. It must begin with the @ sign followed by one or more letters listed in the table below:

Picture function characters

Function	Formatting rule
B	Formats numbers left-justified
C	Adds CR (credit) after positive numbers
D	Formats dates in SET DATE format
E	Formats dates and numbers in British format
L	Pads numbers with zeros instead of blank spaces
R	Nontemplate characters are inserted
X	Adds DB (debit) after negative numbers
Z	Formats zeros as blanks
(	Encloses negative numbers in parentheses
!	Converts alphabetic characters to uppercase

Picture mask

The picture mask must be separated by a single space from the picture function. When no picture function is used, the picture string is identical with the picture mask. The mask defines formatting rules for individual characters in the output string. Characters from the following table can be used. The position of a character of a picture mask specifies formatting for the character of the output string at the same position. An exception is the @R function which causes non-mask characters being inserted into the output string.

Picture mask characters

Charactere	Formatting rule
A,N,X,9,#	Formats digits for any data type
L	Formats logicals as "T" or "F"
Y	Formats logicals as "Y" or "N"
!	Converts alphabetic characters to uppercase
$	Adds a dollar sign in place of a leading space in a number
*	Adds an asterisk in place of a leading space in a number
.	Specifies a decimal point position
,	Specifies a comma position

Info

See also: @...SAY, DtoC(), Lower(), PadC() \| PadL() \| PadR(), Str(), Upper(), Val()
Category: Conversion functions
Source: rtl\transfrm.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates different formatting results of Transform()

   PROCEDURE Main
      LOCAL nGain  := 8596.58
      LOCAL nLoss  := -256.50
      LOCAL cPhone := "5558978532"
      LOCAL cName  := "Jon Doe"

      ? Transform( 8596.58, "@E 9,999.99" )   // result: 8.596,58

      ? Transform( 8596.58, "999,999.99" )     // result:   8,596.58
      ? Transform( 8596.58, "@L 999,999.99" ) // result: 008,596.58

      ? Transform( -256.50, "@)" )            // Result: (256.50)

      ? Transform( "5558978532", "@R (999)999-9999" )
                                              // Result: (555)897-8532

      ? Transform( "xharbour", "@!" )         // Result: XHARBOUR
      ? Transform( "xharbour", "A!AAAAAA" )   // Result: xHarbour
   RETURN
```

Trim()

Removes trailing blank spaces from a character string.

Syntax

```
Trim( <cString> [,<lAllWhiteSpace>] ) --> cTrimmedString
RTrim( <cString> [,<lAllWhiteSpace>] ) --> cTrimmedString
```

Arguments

`<cString>`

A character string which is copied without trailing blank space characters.

`<lAllWhiteSpace>`

This parameter defaults to .F. (false). If .T. (true) is passed, function Trim() treats the white-space characters TAB (Chr(9)) and CRLF (Chr(13)+Chr(10)) like blank spaces and removes them as well.

Return

The function returns a copy of *<cString>* without blank spaces at the end of the string.

Description

Trim() is used for formatting character strings whose ending characters consist of blank spaces (Chr(32)). The function creates a copy of *<cString>* but ignores blank spaces at the end of the input string. It is frequently used to format character strings stored in database fields. Such strings are padded with blank paces up to the field length.

Note() function RTrim() is a synonym for Trim().

Info

See also: Alltrim(), LTrim(), PadC() | PadL() | PadR(), SubStr()
Category: Character functions
Source: rtl\trim.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example shows various possibilities for trimming a string.

   #define CRLF     Chr(13)+Chr(10)

   PROCEDURE Main
      LOCAL cStr  := "  xHarbour    "

      ? Len( cStr )                  // result:  14
      ? Len( LTrim( cStr ) )         // result:  12
      ? Len( Trim ( cStr ) )         // result:  10
      ? Len( AllTrim( cStr ) )       // result:   8

      cStr := "xHarbour  " + CRLF

      ? Len( cStr )                  // result:  12
      ? Len( Trim( cStr ) )          // result:  12
      ? Len( Trim( cStr, .T. ) )     // result:   8

   RETURN
```

TrueName()

Completes a relative path to include the root directory.

Syntax

```
TrueName( <cPath> ) --> cRoot
```

Arguments

`<cPath>`

A character string holding a valid path specification like "." or "..\..\".

Return

The function returns a character string containing a full path name beginning from the root directory. If *<cPath>* is invalid, the return value is a null string ("").

Info

See also: AfterAtNum(), Token()
Category: CT:DiskUtil, Directory functions
Source: ct\disk.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example shows the result of TrueName()

   PROCEDURE Main

      ? TrueName( "..\lib" )   // result: C:\xhb\lib

      ? TrueName( "..\..\dll\xhbdll.dll" )
                               // result: C:\xhb\dll\xhbdll.dll
   RETURN
```

TString()

Converts numeric seconds into a time formatted character string.

Syntax

```
TString( <nSeconds> ) --> cTimeString
```

Arguments

<nSeconds>

This is a numeric value specifying the number of seconds elapsed since midnight. If *<nSeconds>* is larger than 86400, it is set back to 0.

Return

The function returns a character string containing the number of seconds formatted as hh:mm:ss.

Description

TString() is used to convert numeric seconds into a time formatted character string containing hours, minutes and seconds of a 24h clock. It is the reverse function of Secs().

Info

See also: Days(), ElapTime(), Time(), Seconds(), Secs()
Category: Date and time
Source: rtl\samples.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example shows various possibilities of converting seconds
// and time strings.

   PROCEDURE Main
      LOCAL nSeconds := Seconds()
      LOCAL cTime    := Time()

      ? nSeconds                     // result:     57816.61
      ? TString( nSeconds )          // result: 16:03:36
      ? Secs( cTime )                // result:     57816

      ? TString( 0 )                 // result: 00:00:00
      ? TString( 60 )                // result: 00:01:00
      ? TString( 3600 )              // result: 01:00:00

      ? TString( 86399 )             // result: 23:59:59
      ? TString( 86400 )             // result: 00:00:00
      ? TString( 86460 )             // result: 00:01:00
   RETURN
```

TtoC()

Converts a DateTime value to a character string in SET DATE and SET TIME format.

Syntax

```
TtoC( <dDateTime> ) --> cDateTimeString
```

Arguments

<dDateTime>

The parameter must be a value of data type Date or DateTime.

Return

The return value is a character string formatted in the current SET DATE and SET TIME format.

Description

The function converts a DateTime value to a character string. The string is formatted according to the current SET DATE and SET TIME format.

Important: use TtoC() and its counterpart CtoT() with extreme care. The result of both functions depends on the current SET DATE, SET EPOCH and SET TIME settings.

Info

See also:	DateTime(), DtoC(), SET CENTURY, SET DATE, SET EPOCH, SET TIME, StoT(), TtoC(), TtoS()
Category:	Conversion functions, Date and time, xHarbour extensions
Source:	rtl\dateshb.c
LIB:	xhb.lib
DLL:	xhbdll.dll

TtoS()

Converts a Date value to a character string in YYYYMMDDhhmmss.ccc format.

Syntax

```
TtoS( <dDateTime>)  --> cYYYYMMDDhhmmss.ccc
```

Arguments

<dDateTime>

The parameter must be a value of data type Date or DateTime.

Return

The return value is a character string of 18 bytes in lenth formatted as "YYYYMMDDhhmmss.ccc".

Description

This function converts a Date value to a character string of 18 bytes length that is independent from the settings SET CENTURY, SET DATE, SET EPOCH, and SET TIME. Therefore, TtoS() and its counterpart StoT() are the recommended DateTime to Character string conversion functions, unless a DateTime value must be output in a country specific format.

When DateTime fields are combined with Character fields in an index expression, use TtoS() to concatenate the DateTime field with the Character field. This ensures that the Date field is sorted in chronological order.

Info

See also:	DateTime(), DtoS(), SET CENTURY, SET DATE, SET EPOCH, StoD(), StoT(), TtoC()
Category:	Conversion functions, Date and time, xHarbour extensions
Source:	rtl\dateshb.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates TtoS() return values and outlines
the effect of SET DATE and SET TIME for DateTime output.

   PROCEDURE Main
      LOCAL dDateTime := {^ 2007/04/26 16:31:24.789 }
      LOCAL cDateTime := TtoS( dDateTime )

      ? dDateTime                      // result: 04/26/07 16:31:24.78
      ? cDateTime                      // result: 20070426163124.789

      SET DATE TO ITALIAN
      SET TIME FORMAT TO "hh:mm pm"

      ? dDateTime                      // result: 26-04-07 04:31 PM
      ? cDateTime                      // result: 20070426163124.789

      ? cDateTime := TtoS( {^ 0/0/9 } )  // result:         000000.000
      ? Len( cDateTime )               // result:         18
   RETURN
```

Type()

Determines the data type of a macro expression.

Syntax

```
Type( <cMacroExpr> ) --> cDataType
```

Arguments

`<cMacroExpr>`

This is a character string holding the name of a dynamic memory variable (PRIVATE or PUBLIC), field variable or an expression.

Return

The function returns a character string identifying the data type of *<cMacroExpr>* when it is evaluated using the macro operator (&) (see description).

Description

Type() is used to determine the data type of a macro expression by evaluating or executing it with the macro operator. The expression is most often the name of a dynamic memory variable, or a field variable.

Type() can also determine the data type of a macro expression containing function calls. This is restricted, however, to basic functions built into xHarbour which operate on simple data types. If the macro expression contains calls to complex functions or user-defined functions, Type() does not evaluate the macro expression and, thus, cannot determine the data type.

The function returns one of the following characters or character strings

Return values of Type()

Return value	Data type
A	Array
B	Code block
C	Character string
D	Date value
H	Hash
L	Logical value
M	Memo field
N	Numeric value
O	Object
P	Pointer to function or method
U	Undefined symbol or NIL
UE	Syntax error in macro expression
UI	Data type is indeterminable (macro expression too complex)

Note Type() cannot determine the data type of lexically scoped variables (GLOBAL, LOCAL and STATIC). Use Valtype() for such variables.

Info

See also: Valtype()
Category: Debug functions
Source: rtl\type.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// This example shows all possible return values of Type()

   PROCEDURE Main
      PRIVATE aArray  := { 1, 2, 3 }
      PRIVATE bBlock  := {|x| 1+x }
      PRIVATE cChar   := "xHarbour"
      PRIVATE dDate   := Date()
      PRIVATE hHash   := Hash()
      PRIVATE lLogic  := .F.
      PRIVATE nNumber := 123.45
      PRIVATE oObject := GetNew()
      PRIVATE pPtr    := ( @Test() )
      PRIVATE uNIL    := NIL

      USE Customer ALIAS Cust

      ? Type( "aArray"  )            // result: A
      ? Type( "bBlock"  )            // result: B
      ? Type( "cChar"   )            // result: C
      ? Type( "dDate"   )            // result: D
      ? Type( "hHash"   )            // result: H
      ? Type( "lLogic"  )            // result: L
      ? Type( "Cust->notes" )        // result: M
      ? Type( "nNumber" )            // result: N
      ? Type( "oObject" )            // result: O
      ? Type( "pPtr"    )            // result: P
      ? Type( "uNIL"    )            // result: U

      ? Type( "Val(" )               // result: UE
      ? Type( "Test()" )             // result: UI
      ? Type( "TestFunc()" )         // result: U
      ? Type( "Date()" )             // result: D
      ? Type( "oObject:hasFocus()" ) // result: L

      Test( pPtr, hHash )            // result: P H

      ? Type( "'A'" )                // result: C
      ? Type( "A" )                  // result: U
      ? Type( "{1}" )                // result: A
      ? Type( "{=>}" )               // result: H
      ? Type( "9" )                  // result: N
      ? Type( ".F." )                // result: L
      ? Type( "IIf(.T.,'A',1)" )     // result: C
   RETURN


   PROCEDURE Test
      PARAMETERS p1, p2

      ? Type( "p1" ), Type( "p2" )
   RETURN
```

U2bin()

Converts a numeric value to an unsigned long binary integer (4 bytes).

Syntax

```
U2bin( <nNumber> ) --> cInteger
```

Arguments

<nNumber>

A numeric value in the range of 0 to +(2^32)-1.

Return

The function returns a four-byte character string representing a 32-bit unsigned long binary integer.

Description

U2bin() is a binary conversion function that converts a numeric value (Valtype()=="N") to a four-byte binary number (Valtype()=="C").

The range for the numeric return value is determined by an unsigned long integer. If *<nNumber>* is outside this range, a runtime error is raised.

U2bin() is the inverse function of Bin2U().

Info

See also: Asc(), Bin2i(), Bin2l(), Bin2u(), Bin2w(), Chr(), I2bin(), L2bin(), W2bin(), Word()
Category: Binary functions, Conversion functions
Source: rtl\binnumx.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates return values of U2bin().

   PROCEDURE Main
      LOCAL c

      c := U2bin(0)
      ? Asc(c[1]), Asc(c[2]), Asc(c[3]), Asc(c[4])
      //      0          0          0          0

      c := U2bin(1)
      ? Asc(c[1]), Asc(c[2]), Asc(c[3]), Asc(c[4])
      //      1          0          0          0

      c := U2bin(256)
      ? Asc(c[1]), Asc(c[2]), Asc(c[3]), Asc(c[4])
      //      0          1          0          0

      c := U2bin(65536)
      ? Asc(c[1]), Asc(c[2]), Asc(c[3]), Asc(c[4])
      //      0          1          0          0

      c := U2bin(2147483648)
      ? Asc(c[1]), Asc(c[2]), Asc(c[3]), Asc(c[4])
      //      0          0          0        128
```

```
    c := U2bin(4294967295)
    ? Asc(c[1]), Asc(c[2]), Asc(c[3]), Asc(c[4])
    //    255        255        255        255
RETURN
```

Unselected()

Selects the unselected color of SetColor().

Syntax

```
Unselected() --> cNull
```

Return

The function selects the 5th color of the SetColor() setting for standard console output and returns a null string ("").

Info

See also:	ColorSelect(), Enhanced(), Standard()
Category:	CT:Video, Screen functions
Source:	ct\color.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

UntextWin()

Removes all text characters from the screen.

Syntax

```
UntextWin( <nTop>          , ;
           <nLeft>         , ;
           <nBottom>       , ;
           <nRight>        , ;
           <xReplace>      , ;
           [<xExceptFrom>], ;
           [<cExceptTo>]    ) --> cNull
```

Arguments

<nTop>

A numeric value indicating the screen coordinate for the top row of the rectangular screen region to clear.

<nLeft>

A numeric values indicating the screen coordinate for the left column of the rectangular screen region to clear.

<nBottom>

A numeric value indicating the screen coordinate for the bottom row of the rectangular screen region to clear.

<nRight>

A numeric values indicating the screen coordinate for the right column of the rectangular screen region to save.

<xReplace>

This is a single character or its numeric ASCII code. It replaces all textual characters on the screen.

<xExceptFrom> and <xEceptTo>

These are two single characters or their numeric ASCII codes that mark the lower and upper boundary of characters to leave unchanged. The default is the range from Chr(176) to Chr(223). This excludes all graphical characters used for drawing boxes from being replaced.

Return

The function replaces all textual characters on the screen and returns a null string ("").

Note: the function leaves the cursor position unchanged.

Info

See also:	ClWin(), ClearWin()
Category:	CT:Video, Screen functions
Source:	ct\untext.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Updated()

Queries the updated flag of the READ command.

Syntax

```
Updated() --> lIsUpdated
```

Return

The function returns the Updated flag of the READ command as a logical value. This flag is set to .F. (false) when READ begins, and to .T. (true) whenever a single Get enry field is edited and changed by the user.

Info

See also:	@...GET, Get(), READ, ReadUpdated()
Category:	Get system
Source:	rtl\getsys.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Upper()

Converts a character string to uppercase.

Syntax

```
Upper( <cString> ) --> cUpperCaseString
```

Arguments

<cString>

A character string to convert to uppercase letters.

Return

The return value is a character string containing only uppercase letters.

Description

The character function Upper() copies the passed string, replaces all lowercase letters with uppercase letters and returns the result. It is related to function Lower() wich converts a string to lowercase letters. Both functions are used for case-insensitive string routines.

Info

See also:	IsLower(), IsUpper(), Lower()
Category:	Character functions
Source:	rtl\strcase.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates various results of Upper()

   PROCEDURE Main

      ? Upper( "xHARbOUR" )             // result: XHARBOUR
      ? Upper( "123 ABC - def" )        // result: 123 ABC - DEF

   RETURN
```

Used()

Determines if a database file is open in a work area.

Syntax

```
Used() --> lIsUsed
```

Return

The function returns .T. (true) if a database file is open in a specified work area, otherwise .F. (false) is returned.

Description

Used() determines if a database file is open in a work area, i.e. if a work area is "used". If the function is called without the alias operator, it operates in the current work area. The current work area is selected with function DbSelectArea().

Info

See also:	Alias(), DbSelectArea(), DbUseArea(), SELECT, Select(), USE
Category:	Database functions, Environment functions
Source:	rdd\dbcmd.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The shows how to query the Used status of work areas

   PROCEDURE Main
      ? Select()                     // result:  1
      ? Used()                       // result: .F.

      USE Customer ALIAS Cust NEW
      ? Select()                     // result:  1
      ? Used()                       // result: .T.

      DbSelectArea( 2 )
      ? Select()                     // result:  2
      ? Used()                       // result: .F.

      ? (1)->(Used())                // result: .T.
      ? (2)->(Used())                // result: .F.
      ? Cust->(Used())               // result: .T.

      CLOSE Cust
   RETURN
```

UsrRdd_AreaData()

Queries or attaches user-defined data to a work area of a user-defined RDD.

Syntax

```
UsrRdd_AreaData( <nWA>|<pWA>, [<xNewData>] ) --> xOldData
```

Arguments

<nWA>

This is the numeric work area identifier of a work area maintained by a user-defined RDD.

<pWA>

Alternatively, the work area pointer as passed from internal functions of the RDD system can be specified.

<xNewData>

This is the data to attach to a work area maintained by a user-defined RDD.

Return

The function returns the previous data attached to the work area.

Description

Function UsrRdd_AreaData() is used to attach user-defined data to the work area maintained by a user-defined RDD. Such data is usually an array or an object holding state variables of a work area. The function is normally called in response to the USE command, when a table is opened in a work area. User-defined data is discarded automatically when the work area is closed.

Refer to the PRG files in \source\rdd\usrrdd\rdds for examples of attaching user-defined data to a work area.

Info

See also:	UsrRdd_RddData(), UsrRdd_AreaResult(), Select()
Category:	Database drivers, User-defined RDD, xHarbour extensions
Header:	usrrdd.ch
Source:	rdd\usrrdd\usrrdd.c
LIB:	lib\xhb.lib
DLL:	dll\xhbdll.dll

UsrRdd_AreaResult()

Queries the result of the last operation of a user-defined RDD.

Syntax

```
UsrRdd_AreaResult( <nWA>|<pWA> ) --> xResult
```

Arguments

<nWA>

This is the numeric work area identifier of a work area maintained by a user-defined RDD.

<pWA>

Alternatively, the work area pointer as passed from internal functions of the RDD system can be specified.

Return

The function returns the result of the last operation of a user-defined RDD.

Description

UsrRdd_AreaResult() can be used to query the result of the last database operation performed by a user-defined RDD. The data type of the return value is equivalent with the corresponding Db*() function that performs the same operation with a DBF database.

Info

See also:	UsrRdd_AreaData(), UsrRdd_RddData(), Select()
Category:	Database drivers, User-defined RDD, xHarbour extensions
Header:	usrrdd.ch
Source:	rdd\usrrdd\usrrdd.c
LIB:	lib\xhb.lib
DLL:	dll\xhbdll.dll

UsrRdd_ID()

Queries the ID of a user-defined RDD.

Syntax

```
UsrRdd_ID( <nWA>|<pWA> ) --> RddID
```

Arguments

`<nWA>`

This is the numeric work area identifier of a work area maintained by a user-defined RDD.

`<pWA>`

Alternatively, the work area pointer as passed from internal functions of the RDD system can be specified.

Return

The function returns the ID of the user-defined RDD maintaining a work area. When the RDD of the current work-area is not a user-defined RDD, the return value is NIL.

Description

UsrRdd_ID() determines the ID of a user-defined RDD from a work area ID. The return value is required for function UsrRdd_RddData() which queries data attached the user-defined RDD.

Refer to the PRG files in \source\rdd\usrrdd\rdds for examples of querying the ID of a user-defined RDD.

Info

See also:	RddRegister(), UsrRdd_AreaResult(), UsrRdd_RddData(), Select()
Category:	Database drivers, User-defined RDD, xHarbour extensions
Header:	usrrdd.ch
Source:	rdd\usrrdd\usrrdd.c
LIB:	lib\xhb.lib
DLL:	dll\xhbdll.dll

UsrRdd_RddData()

Queries or attaches user-defined data a user-defined RDD.

Syntax

```
UsrRdd_RddData( <nRddID>, [<xNewData>] ) --> xOldData
```

Arguments

<nRddID>

This is the numeric identifier of a user-defined RDD. It is returned from function UsrRdd_ID().

<xNewData>

This optional parameter holds arbitrary data of any data type to attach to a user-defined RDD.

Return

The function returns the previous data attached to the user-defined RDD.

Description

Function UsrRdd_RddData() is used to attach user-defined data to the user-defined RDD. Such data is usually an array or an object. The function is normally called in the initialization routine of the user-defined RDD.

Refer to the PRG files in \source\rdd\usrrdd\rdds for examples of attaching user-defined data to a user-defined RDD.

Info

See also:	UsrRdd_AreaData(), UsrRdd_AreaResult(), Select()
Category:	Database drivers, User-defined RDD, xHarbour extensions
Header:	usrrdd.ch
Source:	rdd\usrrdd\usrrdd.c
LIB:	lib\xhb.lib
DLL:	dll\xhbdll.dll

UsrRdd_SetBof()

Sets the BoF() flag in a work area of a user-defined RDD.

Syntax

```
UsrRdd_SetBof( <nWA>|<pWA>, <lBof> ) --> NIL
```

Arguments

<nWA>

This is the numeric work area identifier of a work area maintained by a user-defined RDD.

<pWA>

Alternatively, the work area pointer as passed from internal functions of the RDD system can be specified.

<lBof>

This is a logical value specifying the BoF() flag for the work area.

Return

The return value is always NIL

Description

UsrRdd_SetBof() is used to set the Begin-of-file flag to .T. (true) or .F (false). A user-defined RDD is not required to re-implement the Begin-of-file flag since it is incorporeted in the user-defined RDD system.

Info

See also:	BoF(), UsrRdd_SetBottom(), UsrRdd_SetEof(), UsrRdd_SetFound(), UsrRdd_SetTop()
Category:	Database drivers, User-defined RDD, xHarbour extensions
Header:	usrrdd.ch
Source:	rdd\usrrdd\usrrdd.c
LIB:	lib\xhb.lib
DLL:	dll\xhbdll.dll

UsrRdd_SetBottom()

Defines the bottom scope value for a work area of a user-defined RDD.

Syntax

```
UsrRdd_SetBottom( <nWA>|<pWA>, <xBottom> ) --> NIL
```

Arguments

<nWA>

This is the numeric work area identifier of a work area maintained by a user-defined RDD.

<pWA>

Alternatively, the work area pointer as passed from internal functions of the RDD system can be specified.

<xBottom>

This ia the bottom scope value.

Description

UsrRdd_SetBottom() is used to set the Bottom scope value of a work area maintained by a user-defined RDD. A user-defined RDD is not required to re-implement the Bottom scope value since it is incorporeted in the user-defined RDD system.

Refer to SET SCOPEBOTTOM for more information on the bottom scope.

Info

See also:	SET SCOPEBOTTOM, UsrRdd_SetBof(), UsrRdd_SetEof(), UsrRdd_SetFound(), UsrRdd_SetTop()
Category:	Database drivers, User-defined RDD, xHarbour extensions
Header:	usrrdd.ch
Source:	rdd\usrrdd\usrrdd.c
LIB:	lib\xhb.lib
DLL:	dll\xhbdll.dll

UsrRdd_SetEof()

Sets the Eof() flag in a work area of a user-defined RDD.

Syntax

```
UsrRdd_SetEof( <nWA>|<pWA>, <lEof> ) --> NIL
```

Arguments

<nWA>

This is the numeric work area identifier of a work area maintained by a user-defined RDD.

<pWA>

Alternatively, the work area pointer as passed from internal functions of the RDD system can be specified.

<lEof>

This is a logical value specifying the EoF() flag for the work area.

Return

The return value is always NIL

Description

UsrRdd_SetEof() is used to set the End-of-file flag to .T. (true) or .F (false). A user-defined RDD is not required to re-implement the End-of-file flag since it is incorporeted in the user-defined RDD system.

Info

See also:	UsrRdd_SetBof(), UsrRdd_SetBottom(), UsrRdd_SetFound(), UsrRdd_SetTop()
Category:	Database drivers, User-defined RDD, xHarbour extensions
Header:	usrrdd.ch
Source:	rdd\usrrdd\usrrdd.c
LIB:	lib\xhb.lib
DLL:	dll\xhbdll.dll

UsrRdd_SetFound()

Sets the Found() flag in a work area of a user-defined RDD.

Syntax

```
UsrRdd_SetFound( <nWA>|<pWA>, <lFound> ) --> NIL
```

Arguments

<nWA>

This is the numeric work area identifier of a work area maintained by a user-defined RDD.

<pWA>

Alternatively, the work area pointer as passed from internal functions of the RDD system can be specified.

<lFound>

This is a logical value specifying the EoF() flag for the work area.

Return

The return value is always NIL

Description

UsrRdd_SetFound() is used to set the Found flag to .T. (true) or .F (false). A user-defined RDD is not required to re-implement the Found flag since it is incorporeted in the user-defined RDD system.

Info

See also: UsrRdd_SetBof(), UsrRdd_SetBottom(), UsrRdd_SetEof(), UsrRdd_SetTop()
Category: Database drivers, User-defined RDD, xHarbour extensions
Header: usrrdd.ch
Source: rdd\usrrdd\usrrdd.c
LIB: lib\xhb.lib
DLL: dll\xhbdll.dll

UsrRdd_SetTop()

Defines the top value for a work area of a user-defined RDD.

Syntax

```
UsrRdd_SetTop( <nWA>|<pWA>, <xTop> ) --> NIL
```

Arguments

<nWA>

This is the numeric work area identifier of a work area maintained by a user-defined RDD.

<pWA>

Alternatively, the work area pointer as passed from internal functions of the RDD system can be specified.

<xTop>

This ia the top scope value.

Description

UsrRdd_SetTop() is used to set the Top scope value of a work area maintained by a user-defined RDD. A user-defined RDD is not required to re-implement the Top scope value since it is incorporeted in the user-defined RDD system.

Refer to SET SCOPETOP for more information on the Top scope.

Info

See also:	SET SCOPETOP, UsrRdd_SetBof(), UsrRdd_SetBottom(), UsrRdd_SetEof(), UsrRdd_SetFound()
Category:	Database drivers, User-defined RDD, xHarbour extensions
Header:	usrrdd.ch
Source:	rdd\usrrdd\usrrdd.c
LIB:	lib\xhb.lib
DLL:	dll\xhbdll.dll

Val()

Convert a character string containing digits to numeric data type

Syntax

```
Val( <cNumber> ) --> nNumber
```

Arguments

<cNumber>

This is a character string containing digits. In addition, it may contain a decimal point and/or sign.

Return

The function returns a numeric value as the result of the conversion operation. If *<cNumber>* does not represent a number, the return value is zero.

Description

Val() analyses the parameter *<cNumber>* which may contain only "+-.0123456789". If *<cNumber>* contains any other character, a letter, for example, or if a second decimal point is detected, the conversion is terminated and the digits analysed so far are returned as numeric value. Leading blank space characters, however, are ignored.

The function recognizes the SET FIXED and SET DECIMALS settings and rounds the number if *<cNumber>* contains more decimal places than set with SET DECIMAL.

Info

See also:	Round(), SET DECIMALS, SET FIXED, Str(), Transform()
Category:	Character functions, Conversion functions
Source:	rtl\val.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows return values of Val() depending on SET FIXED
// and SET DECIMALS

   PROCEDURE Main

      SET DECIMALS TO 2
      SET FIXED ON

      ? Val( "123.456" )                // result: 123.46
      ? Val("  123.456")                // result:   123.46
      ? Val( "123*456" )                // result:     123.00
      ? Val("x123*456" )                // result:        0.00

      ? Val( "-123.456" )               // result: -123.46
      ? Val( "  -123.456" )             // result:    -123.46
      ? Val( "-   123.456" )            // result:        0.00

      SET DECIMALS TO 3
      SET FIXED OFF

      ? Val( "123.456" )                // result: 123.456
      ? Val("  123.456")                // result:   123.456
```

```
   ? Val( "123*456" )              // result:     123
   ? Val("x123*456" )              // result:        0

   ? Val( "-123.456" )             // result: -123.456
   ? Val( "  -123.456" )           // result:    -123.456
   ? Val( "-   123.456" )          // result:        0.000

RETURN
```

ValPos()

Returns the numeric value of a digit at a specified position in a string.

Syntax

```
ValPos( <cString>, [<nPos>] ) --> nValue
```

Arguments

<cString>

This is a character string containing digits.

<nPos>

Optionally, the ordinal position of the character to query can be specified as a numeric value. It defaults to Len(*<cString>*), the last character.

Return

The function returns the numeric value of the character (digit) at the specified position. If the character is not a digit or *<nPos>* is larger than Len(*<cString>*), the return value is zero.

Note: the function exists for compatibility reasons. The [ ] operator can be applied to character strings in xHarbour and is more efficient for extracting a single character.

Info

See also:	[ ] (string), AscPos(), Val()
Category:	CT:String manipulation, Character functions
Source:	ct\ascpos.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows two different ways of obtaining the
// numeric value of a single digit in a string.

   PROCEDURE Main
      LOCAL cString := "abc8def6"

      // compatibility
      ? ValPos( cString )         // result: 6
      ? ValPos( cString, 4 )      // result: 8

      // xHarbour [] operator
      ? Val( cString[-1] )        // result: 6
      ? Val( cString[4] )         // result: 8
   RETURN
```

ValToPrg()

Converts a value to PRG code.

Syntax

```
ValToPrg( <xValue> ) --> cPRGcode
```

Arguments

<xValue>

This is a value of any data type.

Return

The function returns a character string that can be compiled as PRG code.

Description

Function ValToPrg() accepts a value of any data type and returns a character string holding PRG code. The PRG code can be compiled and produces the passed value. This is guaranteed for all data types except Object and Code block, since certain restrictions exist for these data types when they are serialized into a character string. Refer to function HB_Serialize() for more information on these restrictions.

Note: if a pointer is passed, the produced PRG code is a Hex number which does not compile to a pointer but to a Numeric value. Pointers are only determined at runtime of an application and cannot be hard coded.

Info

See also:	CStr(), HB_Serialize(), HB_ValToStr(), ValToPrgExp(), Valtype()
Category:	Conversion functions, xHarbour extensions
Source:	rtl\cstr.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates return values of function ValToPrg()
// with data types where the function is guaranteed to work.

   PROCEDURE Main
      LOCAL aArray   := { 1, 2, 3 }
      LOCAL cString  := "xHarbour"
      LOCAL dDate    := Date()
      LOCAL hHash    := { "A" => 1, "B" => 2 }
      LOCAL lLogic   := .T.
      LOCAL nNumber  := 1.2345
      LOCAL pPointer := ( @Main() )
      LOCAL undef    := NIL

      ? ValToPrg( aArray   )    // result: {  1,  2,  3 }
      ? ValToPrg( cString  )    // result: "xHarbour"
      ? ValToPrg( dDate    )    // result: sToD( '20061011' )
      ? ValToPrg( hHash    )    // result: { "A" =>  1, "B" =>  2 }
      ? ValToPrg( lLogic   )    // result: .T.
      ? ValToPrg( nNumber  )    // result:          1.2345
      ? ValToPrg( pPointer )    // result: 0x4C5000
      ? ValToPrg( undef    )    // result: NIL
   RETURN
```

ValToPrgExp()

Converts a value to a character string holding a macro-expression.

Syntax

```
ValToPrgExp( <xValue> ) --> cMacroExpression
```

Arguments

<xValue>

This is a value of any data type.

Return

The function returns a character string that can be compiled with the macro operator.

Description

Function ValToPrgExp() accepts a value of any data type and returns a character string holding a macro-expression. This expression produces the value when it is compiled with the Macro-operator. This is guaranteed for all data types except Object and Code block, since certain restrictions exist for these data types when they are serialized into a character string. Refer to function HB_Serialize() for more information on these restrictions.

Note: if a pointer is passed, the produced PRG code is a Hex number which does not macro-compile to a pointer but to a Numeric value.

Info

See also:	&, CStr(), HB_Serialize(), PrgExpToVal(), Valtype(), ValToPrg()
Category:	Conversion functions, xHarbour extensions
Source:	rtl\cstr.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates return values of function ValToPrgExp()
// with data types where the function is guaranteed to work.

   PROCEDURE Main
      LOCAL aArray   := { 1, 2, 3 }
      LOCAL cString  := "xHb"
      LOCAL dDate    := Date()
      LOCAL hHash    := { "A" => 1, "B" => 2 }
      LOCAL lLogic   := .T.
      LOCAL nNumber  := 1.2345
      LOCAL pPointer := ( @Main() )
      LOCAL undef   := NIL

      ? ValToPrgExp( aArray   )    // result: { 1, 2, 3 }
      ? ValToPrgExp( cString  )    // result: Chr(120) + Chr(72) + Chr(98)
      ? ValToPrgExp( dDate    )    // result: sToD( '20061011' )
      ? ValToPrgExp( hHash    )    // result: {  Chr( 65) => 1,  Chr( 66) => 2 }
      ? ValToPrgExp( lLogic   )    // result: .T.
      ? ValToPrgExp( pPointer )    // result: HexToNum('4C5000')
      ? ValToPrgExp( undef    )    // result: NIL
   RETURN
```

Valtype()

Determines the data type of the value returned by an expression.

Syntax

```
Valtype( <expression >) --> cDataType
```

Arguments

`<expression>`

This is an expression of any data type.

Return

The function returns a character string identifying the data type of <*expression*> (see description).

Description

Valtype() is used to determine the data type of an arbitrary expression, including lexically scoped variables (GLOBAL, LOCAL, STATIC), and (user-defined) functions and methods. Valtype() is more powerful than Type() which determines the data type of macro expressions only.

The function returns one of the following characters:

Return values of Valtype()

Return value	Data type
A	Array
B	Code block
C	Character string
D	Date value, DateTime value
H	Hash
L	Logical value
M	Memo field
N	Numeric value
O	Object
P	Pointer to function, procedure or method
U	Undefined symbol or NIL

Note: if any part of <*expression*> does not exist, Valtype() generates a runtime error.

Important: both, Date and DateTime values, yield "D" with Valtype(). Use function HB_IsDateTime() to distinguish Date from DateTie values.

Info

See also: Type(), HB_IsArray(), HB_IsBlock(), HB_IsByRef(), HB_IsDate(), HB_IsDateTime(), HB_IsHash(), HB_IsLogical(), HB_IsMemo(), HB_IsNIL(), HB_IsNull(), HB_IsNumeric(), HB_IsObject(), HB_IsPointer(), HB_IsString()
Category: Debug functions, Pointer functions
Source: rtl\valtype.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example shows all possible return values of Valtype()

   PROCEDURE Main
      LOCAL aArray  := { 1, 2, 3 }
```

```
   LOCAL bBlock  := {|x| 1+x }
   LOCAL cChar   := "xHarbour"
   LOCAL dDate   := Date()
   LOCAL hHash   := Hash()
   LOCAL lLogic  := .F.
   LOCAL nNumber := 123.45
   LOCAL oObject := GetNew()
   LOCAL pPtr    := ( @Test() )
   LOCAL uNIL    := NIL

   USE Customer ALIAS Cust

   ? Valtype( aArray  )            // result: A
   ? Valtype( bBlock  )            // result: B
   ? Valtype( cChar   )            // result: C
   ? Valtype( dDate   )            // result: D
   ? Valtype( hHash   )            // result: H
   ? Valtype( lLogic  )            // result: L
   ? Valtype( Cust->notes )        // result: M
   ? Valtype( nNumber )            // result: N
   ? Valtype( oObject )            // result: O
   ? Valtype( pPtr    )            // result: P
   ? Valtype( uNIL    )            // result: U

   ? Valtype( Date() )             // result: D
   ? Valtype( oObject:hasFocus() ) // result: L

   Test( pPtr, hHash )             // result: P H

   ? Valtype( IIf( .T., "A", 1 ) ) // result: C

   ? Valtype( "A" )                // result: C
   ? Valtype( A )                  // result: runtime error
RETURN


PROCEDURE Test( p1, p2 )
   ? Valtype( p1 ), Valtype( p2 )
RETURN
```

Version()

Retrieves xHarbour version information.

Syntax

```
Version() --> cVersion
```

Return

The function returns a charcter string holding version information of xHarbour.

Description

Version() is an informational function that returns the version of the xHarbour runtime library.

Info

See also: HB_BuildInfo(), HB_Compiler(), Os()
Category: Environment functions
Source: rtl\version.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
PROCEDURE Main
   ? Version()      // result: xHarbour build 0.99.61 Intl. (SimpLex)
RETURN
```

VolSerial()

Returns the serial number of a disk drive.

Syntax

```
VolSerial( [<cDrive>] ) --> nVolumeSerial
```

Arguments

<cDrive>

A single character, followed by a colon and backslash, specifies the disk drive to query. If omitted, the current drive is queried.

Return

The function returns the serial number of a disk as a numeric value or -1 in case of an error.

Info

See also: CurDrive(), GetVolInfo(), Os(), Version()
Category: CT:DiskUtil, Disks and Drives
Source: ct\disk.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example lists the volume serial numbers of all drives and displays
// them in Hex format.

   PROCEDURE Main
      LOCAL cDrive, i, nSerial, cSerial

      FOR i:=1 TO 26
         cDrive  := Chr(64+i) + ":\"
         nSerial := VolSerial( cDrive )

         IF nSerial <> -1
            cSerial := NumToHex( nSerial )
            ? cDrive, Stuff( cSerial, 5, 0, ":" )
         ENDIF
      NEXT
   RETURN
```

Volume()

Sets the volume label of a disk.

Syntax

```
Volume( <cVolumeName> ) --> lSuccess
```

Arguments

`<cVolumeName>`

This is a character string starting with a drive letter, followed by a colon and the volume label of up to 11 characters. If the drive letter is omitted, the volume label of the current drive is set (eg: Volume("A:Backup01")).

Return

The function returns .T. (true) if the volume label could be set, otherwise .F. (false) is returned.

Note: the function recognizes the CSetSafety() setting. It leaves the volume label unchanged when CSetSafety() returns .T. (true).

Info

See also:	CSetSafety(), FileSeek(), GetVolInfo()
Category:	CT:DiskUtil, Disks and Drives
Source:	ct\disk.c
LIB:	xhb.lib
DLL:	xhbdll.dll

W2bin()

Converts a numeric value to an unsigned short binary integer (2 bytes).

Syntax

```
W2Bin( <nNumber> ) --> cInteger
```

Arguments

`<nNumber>`

A numeric value in the range of 0 to +(2^15) - 1.

Return

The function returns a two-byte character string representing a 16-bit unsigned short binary integer .

Description

W2Bin() is a binary conversion function that converts a numeric value (Valtype()=="N") to a two byte binary number (Valtype()=="C").

The range for the numeric return value is determined by an unsigned short integer. If *<nNumber>* is outside this range, a runtime error is raised.

Info

See also: Asc(), Bin2i(), Bin2l(), Bin2u(), Bin2w(), Chr(), I2bin(), L2bin(), U2bin()
Category: Binary functions, Conversion functions
Source: rtl\binnumx.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates return values of W2Bin().

   PROCEDURE Main
      LOCAL cInt

      cInt := W2bin(0)
      ? Asc( cInt[1] ), Asc( cInt[2] )  // result:   0    0

      cInt := W2bin(1)
      ? Asc( cInt[1] ), Asc( cInt[2] )  // result:   1    0

      cInt := W2bin(256)
      ? Asc( cInt[1] ), Asc( cInt[2] )  // result:   0    1

      cInt := W2bin(32768)
      ? Asc( cInt[1] ), Asc( cInt[2] )  // result:   0  128

      cInt := W2bin(65535)
      ? Asc( cInt[1] ), Asc( cInt[2] )  // result: 255  255

   RETURN
```

WAClose()

Closes all windows.

Syntax

```
WAClose() --> nErrorCode
```

Return

The function returns zero when all windows are closed, and -1 if no window was open.

Info

See also: WClose()
Category: CT:Window, Windows (text mode)
Source: ct\ctwin.c
LIB: xhb.lib
DLL: xhbdll.dll

WaitForThreads()

Suspends the current thread until all other threads have terminated.

Syntax

```
WaitForThreads() --> NIL
```

Return

The return value is always NIL.

Description

Function WaitForThreads() is only effective when it is called in the Main thread of an xHarbour application. This is the thread executing the first function, or start routine, of the application. WaitForThreads() suspends the Main thread until all other threads have terminated. This makes sure that only the Main thread is active when the function returns. If WaitForThreads() is called in a different thread than the Main thread, it returns immediately and does not suspend thread execution.

Note: when a thread, other than the Main thread, should wait for the termination of a second thread, function JoinThread() can be used.

Info

See also:	GetCurrentThread(), GetThreadID(), HB_MutexCreate(), JoinThread(), StartThread(), StopThread(), ThreadSleep()
Category:	Multi-threading functions, xHarbour extensions
Source:	vm\thread.c
LIB:	xhbmt.lib
DLL:	xhbmtdll.dll

WaitPeriod()

Defines a wait period and allows for time controlled loops.

Syntax

```
WaitPeriod( [<nTime>] ) --> lPeriodIsActive
```

Arguments

`<nTime>`

If specified, this is a numeric value indicating the wait period to install. The unit is 1/100th of a second.

Return

The function returns .T. (true) while the period of *<nTime>* * 100 seconds has not elapsed since the initial call. After the wait period is over, the return value is .F. (false).

Description

An initial call to WaitPeriad() defines a time span in 1/100 seconds. The function returns .T. (true) in subsequent calls until the wait period has elapsed. This allows for programming time controlled loops in this form:

```
WaitPeriod(100)

DO WHILE WaitPeriod()
   <do something>
ENDDO
```

Info

See also:	MilliSec(), Seconds()
Category:	CT:DateTime, Date and time
Source:	ct\settime.c
LIB:	xhb.lib
DLL:	xhbdll.dll

WBoard()

Determines the area that can be used for displaying windows.

Syntax

```
WBoard( [<nTop>]    , ;
        [<nLeft>]   , ;
        [<nBottom>], ;
        [<nRight>]    ) --> nErrorCode
```

Arguments

`<nTop> and <nLeft>`

Numeric values indicating the screen coordinates for the upper left corner of the screen area. Both default to zero.

`<nBottom>`

Numeric value indicating the bottom row screen coordinate. It defaults to MaxRow().

`<nRight>`

Numeric value indicating the right column screen coordinate. It defaults to MaxCol().

Return

The function returns 0 if the useable screen area could be defined, otherwise -1 is returned.

Description

WBoard() restricts the area for displaying windows on the screen to the rectangle specified with the given coordinates. Windows are not visible outside this rectangle, even if they are moved beyond its boundaries. If no parameter is passed to WBoard() the area useable for displaying windows is reset to the entire screen. WSelect(0) also selects the entire screen for screen output.

Info

See also:	WClose(), WMode(), WSetMove(), WAClose()
Category:	CT:Window, Windows (text mode)
Source:	ct\ctwin.c
LIB:	xhb.lib
DLL:	xhbdll.dll

WBox()

Draws a frame around the current window.

Syntax

```
WBox( [<cBoxChars>|<nBoxType>] ) --> nWindowID
```

Arguments

<cBoxChars>|

The appearance of the frame display can be specified as a character string holding up to nine characters. The first eight characters define the border of the frame while the ninth character is used to fill it. #define constants to be used for *<cBoxChars>* are available in the BOX.CH #include file.

Pre-defined box strings for WBox()

Constant	Description
B_SINGLE	Single-line box
B_DOUBLE	Double-line box
B_SINGLE_DOUBLE	Single-line top, double-line sides
B_DOUBLE_SINGLE	Double-line top, single-line sides

<nBoxType>

Alternatively, a numeric value in the range from 0 to 15 can be used to select predefined frames. The values 0-3 and 8-11 draw a frame that clears the window, while 4-7 and 12-15 draw a frame without clearing the window.

Return

The function returns the window ID of the window where the frame is drawn, or -1 if the window is too small for drawing a frame.

Info

See also: SetClearB(), WOpen(), WSelect(), WFormat()
Category: CT:Window, Windows (text mode)
Header: Box.ch
Source: ct\ctwin.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays all 16 predefined frame types. To see the
// difference between windows (not) clearing the screen, the
// entire screen is filled with dots using DispBox().

   PROCEDURE Main
      LOCAL nT := 2, nL := 2, nH := 10, nW := 6
      LOCAL i

      SET COLOR TO W+/B
      CLS
      Dispbox( 0, 0, MaxRow(), MaxCol(), "........." )

      FOR i:=0 TO 15
         IF i == 8
```

```
            nT += 15
            nL := 2
         ENDIF

         nWin := WOpen( nT, nL, nT+nH, nL+nW )
         WSelect( nWin )

         SET COLOR TO W+/B
         WBox( i )

         SET COLOR TO GR+/B

         SayDown( "WBox " + Str(i,2), 0, 1, 2 )

         nL += nW+2
      NEXT
   RETURN
```

WCenter()

Centers a window on the screen or makes it entirely visible.

Syntax

```
WCenter( [<lCenter>] ) --> nWindowID
```

Arguments

`<lCenter>`

This parameter defaults to .F. (false) which moves a (partially) hidden window to the screen to become entirely visible. When .T. (true) is passed, the window is centered on the screen.

Return

The function returns the window ID of the current window.

Info

See also:	WBoard(), WOpen(), WSelect()
Category:	CT:Window, Windows (text mode)
Source:	ct\ctwin.c
LIB:	xhb.lib
DLL:	xhbdll.dll

WClose()

Closes the current window and selects the next window as current window.

Syntax

```
WClose() --> nNextWindowID
```

Return

The function closes the currently selected window and selects the window with the highest window ID as current. This window ID is returned as a numeric value.

Info

See also:	WOpen(), WSelect()
Category:	CT:Window, Windows (text mode)
Source:	ct\ctwin.c
LIB:	xhb.lib
DLL:	xhbdll.dll

WCol()

Returns the left column position of the selected window.

Syntax

```
WCol( [<lCentered>] ) --> nLeftColumn
```

Arguments

<lCentered>

This parameter defaults to .F. (false) which causes the function to return the left column position of the selected window. When .T. (true) is passed, the function returns a column position as if the window was centered.

Return

WCol() returns the left column position of the current window, or the position of a centered window, as a numeric value.

Info

See also:	WMode(), WOpen(), WRow(), WSelect()
Category:	CT:Window, Windows (text mode)
Source:	ct\ctwin.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Week()

Calculates the numeric calendar week from a date.

Syntax

```
Week( [<dDate>] ) --> nWeekOfYear
```

Arguments

<dDate>

Any Date value, except for an empty date, can be passed. The default value is Date().

Return

The function returns a numeric value. It is the week number that includes *<dDate>*. If an invalid date is passed, the return value is zero.

Info

See also:	Day(), Month(), WoM(), Year()
Category:	CT:DateTime, Date and time
Source:	ct\dattime2.c
LIB:	xhb.lib
DLL:	xhbdll.dll

WfCol()

Returns the left column position of the usable area in a formatted window.

Syntax

```
WfCol( [<lWidth>] ) --> nLeftColumn
```

Arguments

`<lWidth>`

This parameter defaults to .F. (false) which causes the function to return the left column position of the formatted area in the selected window. When .T. (true) is passed, the function returns the width of the formatted area on the left side of the window.

Return

WfCol() returns the left column position of the formatted area in the current window, or the width of this area, as a numeric value.

Note: when the window with ID 0 (entire screen) is selected, the return value is determined by WBoard().

Info

See also:	WBox(), WCol(), WfLastCol(), WfLastRow(), WfRow(), WFormat()
Category:	CT:Window, Windows (text mode)
Source:	ct\ctwin.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates return values of functions for window
// coordinates.

   PROCEDURE Main
      LOCAL nWin

      SET COLOR TO W+/B
      CLS

      nWin := WOpen( 5, 10, 15, 70 )
      WSelect( nWin )
      WFormat( 6, 11, 14, 68 )
      SET COLOR TO W+/R
      CLS

      ? WRow()     , WCol()               // result:  5  10
      ? WLastRow() , WLastCol()           // result: 15  70

      ? WfRow()    , WfRow(.T.)           // result:  6   1
      ? WfCol()    , WfCol(.T.)           // result: 11   2
      ? WfLastRow(), WfLastRow( .T. )     // result: 14   1
      ? WfLastCol(), WfLastCol( .T. )     // result: 68   2

      Inkey( 0 )
      WClose()
   RETURN
```

WfLastCol()

Returns the right column position of the usable area in a formatted window.

Syntax

```
WfLastCol( [<lWidth>] ) --> nRightColumn
```

Arguments

<lWidth>

This parameter defaults to .F. (false) which causes the function to return the right column position of the formatted area in the selected window. When .T. (true) is passed, the function returns the width of the formatted area on the right side of the window.

Return

WfLastCol() returns the right column position of the formatted area in the current window, or the width of this area, as a numeric value.

Note: when the window with ID 0 (entire screen) is selected, the return value is determined by WBoard().

Info

See also: WBox(), WLastCol(), WfCol(), WfLastRow(), WfRow(), WFormat()
Category: CT:Window, Windows (text mode)
Source: ct\ctwin.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates return values of functions for window
// coordinates.

   PROCEDURE Main
      LOCAL nWin

      SET COLOR TO W+/B
      CLS

      nWin := WOpen( 5, 10, 15, 70 )
      WSelect( nWin )
      WFormat( 6, 11, 14, 68 )
      SET COLOR TO W+/R
      CLS

      ? WRow()     , WCol()              // result:  5  10
      ? WLastRow() , WLastCol()          // result: 15  70

      ? WfRow()    , WfRow(.T.)          // result:  6   1
      ? WfCol()    , WfCol(.T.)          // result: 11   2
      ? WfLastRow(), WfLastRow( .T. )    // result: 14   1
      ? WfLastCol(), WfLastCol( .T. )    // result: 68   2

      Inkey( 0 )
      WClose()
   RETURN
```

WfLastRow()

Returns the bottom row position of the usable area in a formatted window.

Syntax

```
WfLastRow( [<lHeight>] ) --> nBottomRow
```

Arguments

<lHeight>

This parameter defaults to .F. (false) which causes the function to return the bottom row position of the formatted area in the selected window. When .T. (true) is passed, the function returns the height of the formatted area at the bottom of a window.

Return

WfLastRow() returns the bottom row position of the formatted area in the current window, or the height of this area, as a numeric value.

Note: when the window with ID 0 (entire screen) is selected, the return value is determined by WBoard().

Info

See also:	WBox(), WLastRow(), WfCol(), WfLastCol(), WfRow(), WFormat()
Category:	CT:Window, Windows (text mode)
Source:	ct\ctwin.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates return values of functions for window
// coordinates.

   PROCEDURE Main
      LOCAL nWin

      SET COLOR TO W+/B
      CLS

      nWin := WOpen( 5, 10, 15, 70 )
      WSelect( nWin )
      WFormat( 6, 11, 14, 68 )
      SET COLOR TO W+/R
      CLS

      ? WRow()      , WCol()               // result:  5  10
      ? WLastRow() , WLastCol()            // result: 15  70

      ? WfRow()     , WfRow(.T.)           // result:  6   1
      ? WfCol()     , WfCol(.T.)           // result: 11   2
      ? WfLastRow(), WfLastRow( .T. )      // result: 14   1
      ? WfLastCol(), WfLastCol( .T. )      // result: 68   2

      Inkey( 0 )
      WClose()
   RETURN
```

WFormat()

Defines the usable display area inside the current window.

Syntax

```
WFormat( [<nTopMargin>]    , ;
         [<nLeftMargin>]   , ;
         [<nBottomMargin>], ;
         [<nRightMargin>]    ) --> nWindowID
```

Arguments

<nTopMargin>

This is a numeric value specifying the number of rows at the top of a window that cannot be used for screen output. The default value is zero.

<nLeftMargin>

This is a numeric value specifying the number of columns at the left of a window that cannot be used for screen output. The default value is zero.

<nBottomMargin>

This is a numeric value specifying the number of rows at the bottom of a window that cannot be used for screen output. The default value is zero.

<nRightMargin>

This is a numeric value specifying the number of columns at the right of a window that cannot be used for screen output. The default value is zero.

Return

The function restricts the usable area for display inside a window to the area inside the window coordinates plus margins defined. Calling the function without parameters resets all margins to zero. WFormat() returns the ID of the current window as a numeric value.

Note: function WBox() calls WFormat(1, 1, 1, 1) so that the frame displayed by WBox() cannot be overwritten.

Info

See also:	WBox(), WSelect()
Category:	CT:Window, Windows (text mode)
Source:	ct\ctwin.c
LIB:	xhb.lib
DLL:	xhbdll.dll

WfRow()

Returns the top row position of the usable area in a formatted window.

Syntax

```
WfRow( [<lHeight>] ) --> nTopRow
```

Arguments

<lHeight>

This parameter defaults to .F. (false) which causes the function to return the top row position of the formatted area in the selected window. When .T. (true) is passed, the function returns the height of the formatted area at the top of a window.

Return

WfRow() returns the top row position of the formatted area in the current window, or the height of this area, as a numeric value.

Note: when the window with ID 0 (entire screen) is selected, the return value is determined by WBoard().

Info

See also:	WRow(), WfCol(), WfLastCol(), WfLastRow(), WSelect()
Category:	CT:Window, Windows (text mode)
Source:	ct\ctwin.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates return values of functions for window
// coordinates.

   PROCEDURE Main
      LOCAL nWin

      SET COLOR TO W+/B
      CLS

      nWin := WOpen( 5, 10, 15, 70 )
      WSelect( nWin )
      WFormat( 6, 11, 14, 68 )
      SET COLOR TO W+/R
      CLS

      ? WRow()      , WCol()                // result:  5  10
      ? WLastRow()  , WLastCol()            // result: 15  70

      ? WfRow()     , WfRow(.T.)            // result:  6   1
      ? WfCol()     , WfCol(.T.)            // result: 11   2
      ? WfLastRow(), WfLastRow( .T. )       // result: 14   1
      ? WfLastCol(), WfLastCol( .T. )       // result: 68   2

      Inkey( 0 )
      WClose()
   RETURN
```

Wild2RegEx()

Converts a character string including wild card characters to a regular expression.

Syntax

```
Wild2RegEx( <cWildCard> [,<lCaseSensitive>] ) --> cRegEx
```

Arguments

<cWildCard>

This is the character string holding wild card characters ("*" and/or "?").

<lCaseSensitive>

This parameter defaults to .F. (false) so that a case insensitive regular expression is produced. Passing .T. (true) results in a case sensitive regular expression.

Return

The function returns a character string holding the regular expression which is equivalent to the wild card string.

Info

See also:	HB_AtX(), HB_RegEx(), HB_RegExAll(), HB_RegExComp(), HB_RegExSplit()
Category:	Character functions, Regular expressions, xHarbour extensions
Source:	rtl\regex.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example translates a file mask to a regular expression and
// matches it with file names stored in an array.

   PROCEDURE Main
      LOCAL aFiles := { "Customer.dbf", ;
                        "Customer.dbt", ;
                        "Custom01.ntx", ;
                        "Custom02.ntx", ;
                        "Custom03.ntx", ;
                        "Customer01.dbf"}

      LOCAL cFileMask := "CUSTOM??.*"
      LOCAL cRegEx    := Wild2RegEx( cFileMask )
      LOCAL cFile, cFound

      ? "Wild card:", cFileMask
      ? "RegEx    :", cRegex
      ?
      FOR EACH cFile IN aFiles
         cFound := HB_AtX( cRegEx, cFile )

         ? "Matching:", cFile
         IF cFound == NIL
            ?? " no match"
         ELSE
            ?? " OK"
         ENDIF
      NEXT
```

```
   ** Output:
   // Wild card: CUSTOM??.*
   // RegEx    : (?i)\bCUSTOM.?.?\..*\b
   //
   // Matching: Customer.dbf OK
   // Matching: Customer.dbt OK
   // Matching: Custom01.ntx OK
   // Matching: Custom02.ntx OK
   // Matching: Custom03.ntx OK
   // Matching: Customer01.dbf no match
RETURN
```

WildMatch()

Tests if a string begins with a search pattern.

Syntax

```
WildMatch( <cPattern>, <cString>, [<lFullMatch>] ) --> lFound
```

Arguments

<cPattern>

This is a character string defining the search pattern. It may include as wild card characters the question mark (?, matches one character) or the asterisk (*, matches any number of characters).

<cString>

This parameter is a character string to match against *<cPattern>*.

<lFullMatch>

The parameter defaults to .F. (false), causing the function to return when a matching pattern is found at the beginning of *<cString>*. When .T. (true) is passed, *<cString>* is matched entirely against *<cPattern>*.

Return

The function returns .T. (true) when *<cString>* contains the search pattern.

Description

WildMatch() is a pattern matching function that searches a string for a search pattern. If the search pattern is found, the function returns .T. (true).

WildMatch() operates similarly to OrdWildSeek() but can be used as part of a SET FILTER condition on an unindexed database.

Info

See also: $, At(), IN, OrdWildSeek(), SET FILTER
Category: Character functions, xHarbour extensions
Source: rtl\strmatch.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates the pattern matching algorithm employed
// by WildMatch() and how the function can be used as filter condition
// for a database

   PROCEDURE Main
      LOCAL cStr := "The xHarbour compiler"

      ? WildMatch( "bo?"  , cStr, .F. )  // result: .F.
      ? WildMatch( "bo?"  , cStr, .T. )  // result: .F.

      ? WildMatch( "*bo"  , cStr, .F. )  // result: .T.
      ? WildMatch( "*bo"  , cStr, .T. )  // result: .F.

      ? WildMatch( "The"  , cStr, .F. )  // result: .T.
      ? WildMatch( "The"  , cStr, .T. )  // result: .F.

      ? WildMatch( "The*r", cStr, .F. )  // result: .T.
```

```
   ? WildMatch( "The*r", cStr, .T. )  // result: .T.

   ? WildMatch( "The?x", cStr, .F. )  // result: .T.
   ? WildMatch( "The?x", cStr, .T. )  // result: .F.

   USE Customer
   SET FILTER TO WildMatch( "W*s", FIELD->LastName )

   GO TOP
   DbEval( {|| QOut( FIELD->LastName ) } )
   // Output: Names starting with "W" and ending with "s"
   // Walters
   // Waters

   CLOSE Customer
RETURN
```

WInfo()

Returns all coordinates of the current window.

Syntax

```
WInfo() --> aWindowInfo
```

Return

The function returns an array of eight elements holding the screen coordinates of the unformatted and formatted display area of a window.

Array holding window coordinates

Element	Description
Unformatted window	
1	Top row coordinate equivalent to WRow()
2	Left column coordinate equivalent to WCol()
3	Bottom row coordinate equivalent to WLastRow()
4	Right column coordinate equivalent to WLastCol()
Formatted display area	
5	Top row coordinate equivalent to WfRow()
6	Left column coordinate equivalent to WfCol()
7	Bottom row coordinate equivalent to WfLastRow()
8	Right column coordinate equivalent to WfLastCol()

Info

See also: WCol(), WLastCol(), WLastRow(), WOpen(), WRow(), WSelect()
Category: CT:Window, Windows (text mode), xHarbour extensions
Source: ct\ctwin.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example creates a window with a frame (fromatted window)
// and displays the contents of the WInfo() array.

   PROCEDURE Main
      LOCAL nT := 4, nL := 6, nW := 50, nH := 20

      SET COLOR TO N/W
      CLS

      WOpen( nT, nL, nT+nH, nL+nW, .T. )
      WBox()

      aInfo := WInfo()

      FOR i:=1 TO Len( aInfo )
         ? "aInfo[" + Str(i,1) + "] :=", Str(aInfo[i],2)
      NEXT
      ** result:
      // aInfo[1] :=  4
      // aInfo[2] :=  6
      // aInfo[3] := 24
      // aInfo[4] := 56
      // aInfo[5] :=  5
```

```
   // aInfo[6] :=  7
   // aInfo[7] := 23
   // aInfo[8] := 55

   Inkey(0)

   WClose()
RETURN
```

WLastCol()

Returns the right column position of the selected window.

Syntax

```
WLastCol( [<lCentered>] ) --> nRightColumn
```

Arguments

<lCentered>

This parameter defaults to .F. (false) which causes the function to return the right column position of the selected window. When .T. (true) is passed, the function returns a column position as if the window was centered.

Return

WLastCol() returns the right column position of the current window, or the position of a centered window, as a numeric value.

Info

See also:	WCol(), WSelect()
Category:	CT:Window, Windows (text mode)
Source:	ct\ctwin.c
LIB:	xhb.lib
DLL:	xhbdll.dll

WLastRow()

Returns the bottom row position of the selected window.

Syntax

```
WLastRow( [<lCentered>] ) --> nBottomRow
```

Arguments

<lCentered>

This parameter defaults to .F. (false) which causes the function to return the bottom row position of the selected window. When .T. (true) is passed, the function returns a row position as if the window was centered.

Return

WLastRow() returns the bottom row position of the current window, or the position of a centered window, as a numeric value.

Info

See also:	WRow(), WSelect()
Category:	CT:Window, Windows (text mode)
Source:	ct\ctwin.c
LIB:	xhb.lib
DLL:	xhbdll.dll

WMode()

Determines on which side windows are allowed to be moved off the screen.

Syntax

```
WMode( [<lTop>]    , ;
       [<lLeft>]   , ;
       [<lBottom>], ;
       [<lRight>]    ) --> Zero
```

Arguments

<lTop>

If this parameter is set to .T. (true), windows can be moved beyond the top row of the screen, or the area defined with WBoard().

<lLeft>

If this parameter is set to .T. (true), windows can be moved beyond the left column of the screen, or the area defined with WBoard().

<lBottom>

If this parameter is set to .T. (true), windows can be moved beyond the bottom row of the screen, or the area defined with WBoard().

<lRight>

If this parameter is set to .T. (true), windows can be moved beyond the right column of the screen, or the area defined with WBoard().

Return

The function returns always zero.

Info

See also: WBoard(), WMove(), WSetMove()
Category: CT:Window, Windows (text mode)
Source: ct\ctwin.c
LIB: xhb.lib
DLL: xhbdll.dll

WMove()

Changes the upper left coordinate for the current window.

Syntax

```
WMove( <nTop>, <nLeft> ) --> nWindowID
```

Arguments

<nTop>

This is a numeric value defining the new row coordinate for the upper left corner of the current window.

<nLeft>

This is a numeric value defining the new collumn coordinate for the upper left corner of the current window.

Return

The function returns the ID of the current window as a numeric value.

Note: to move the current window outside the WBoard() area, function WMode() must be called with four .T. (true) parameters.

Info

See also: WBoard(), WMode(), WOpen(), WSelect()
Category: CT:Window, Windows (text mode)
Source: ct\ctwin.c
LIB: xhb.lib
DLL: xhbdll.dll

WMSetPos()

Moves the mouse cursor to a new position in a window.

Syntax

```
WMSetPos( <nRow>, <nCol> ) --> NIL
```

Arguments

<nRow>

A numeric value between 0 and MaxRow() specifying the new row position of the mouse cursor in the current window.

<nCol>

A numeric value between 0 and MaxCol() specifying the new columnn position of the mouse cursor in the current window.

Return

The return value is always NIL.

Info

See also: WSetMouse()
Category: CT:Window, Mouse functions, Windows (text mode), xHarbour extensions
Source: ct\ctwin.c
LIB: xhb.lib
DLL: xhbdll.dll

WNum()

Returns the largest window ID of all windows.

Syntax

```
WNum() --> nMaxWindowID
```

Return

The function returns the largest window ID as a numeric value.

Note: window IDs are numeric values that are incremented for each window opened with WOpen().

Info

See also:	WSelect(), WStack()
Category:	CT:Window, Windows (text mode)
Source:	ct\ctwin.c
LIB:	xhb.lib
DLL:	xhbdll.dll

WoM()

Calculates the week number in a month.

Syntax

```
WoM( [<dDate>] ) --> nWeekOfMonth
```

Arguments

<dDate>

Any Date value, except for an empty date, can be passed. The default is the return value of Date().

Return

The function returns a numeric value. It is the week number of the month that includes *<dDate>*. If an invalid date is passed, the return value is zero.

Info

See also:	Day(), Month(), Week(), Year()
Category:	CT:DateTime, Date and time
Source:	ct\datetime.c
LIB:	xhb.lib
DLL:	xhbdll.dll

WOpen()

Creates a new window.

Syntax

```
WOpen( <nTop>    , ;
       <nLeft>   , ;
       <nBottom> , ;
       <nRight>  , ;
      [<lClear>]   ) --> nWindowID
```

Arguments

`<nTop>` and `<nLeft>`

Numeric values indicating the screen coordinates for the upper left corner of the new window.

`<nBottom>` and `<nRight>`

Numeric values indicating the screen coordinates for the lower right corner of the new window.

`<lClear>`

When this parameter is set to .T. (true), the screen area inside the window is cleared. The default value is .F. (false), so that all characters visible on screen are not erased within the area of the new window.

Return

The function returns the ID of the new window as a numeric value. this Window ID must be passed to WSelect() to select it as the current window.

Info

See also: WBoard(), WClose(), WSelect(), WStack()
Category: CT:Window, Windows (text mode)
Source: ct\ctwin.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example opens three windows using different colors. The
// windows are selected with Ctrl+Tab and can be moved interactively
// with the cursor keys when Alt+M is pressed.

   #include "Inkey.ch"

   #define K_CTRL_SH_TAB   423

   PROCEDURE Main
      LOCAL nKey, nWin, aWin
      LOCAL nT := 4, nL := 4, nW := 50, nH := 15
      LOCAL aColor := { "W+/B", "W+/R", "W+/G", "W+/BG" }

      SET COLOR TO (aColor[1])
      CLS

      WBoard( 2, 2, MaxRow()-2, MaxCol()-2 )
      WMode( .T., .T., .T., .T. )

      WSetMove( .F. )
```

```
WSetShadow( 8 )

SET COLOR TO N/W
@ 2, 2 CLEAR TO MaxRow()-2, MaxCol()-2

FOR i:=1 TO 3
   SetColor( aColor[i+1] )
   nWin := WOpen( nT, nL, nT+nH, nL+nW, .T. )

   WBox()
   @ 0,0 SAY "Window: " + Str(i,1)
   nT += 2
   nL += 4
NEXT

aWin := WStack()

DO WHILE .T.
   nKey := Inkey(0)

   DO CASE
   CASE nKey == K_ESC
      EXIT

   CASE nKey == K_CTRL_TAB
      i := AScan( aWin, nWin ) + 1
      IF i > Len( aWin )
         // Skip over entire screen
         i := 2
      ENDIF
      nWin := aWin[i]
      WSelect( nWin )
      SetColor( aColor[i] )

   CASE nKey == K_CTRL_SH_TAB
      i := AScan( aWin, nWin ) - 1
      IF i < 2
         // Skip over entire screen
         i := Len( aWin )
      ENDIF
      nWin := aWin[i]
      WSelect( nWin )
      SetColor( aColor[i] )

   CASE nKey == K_ALT_M
      WSetMove( .NOT. WSetMove() )
      IF WSetMove()
         ? "Cursor keys move window"
      ELSE
         ? "Moveing window switched off"
      ENDIF

   CASE nKey == K_UP
      IF WSetMove()
         nT := WRow() - 1
         nL := WCol()
         WMove( nT, nL )
      ELSE
         ? "Press Alt+M to move window"
      ENDIF

   CASE nKey == K_DOWN
```

```
            IF WSetMove()
               nT := WRow() + 1
               nL := WCol()
               WMove( nT, nL )
            ELSE
               ? "Press Alt+M to move window"
            ENDIF

         CASE nKey == K_LEFT
            IF WSetMove()
               nT := WRow()
               nL := WCol() - 1
               WMove( nT, nL )
            ELSE
               ? "Press Alt+M to move window"
            ENDIF

         CASE nKey == K_RIGHT
            IF WSetMove()
               nT := WRow()
               nL := WCol() + 1
               WMove( nT, nL )
            ELSE
               ? "Press Alt+M to move window"
            ENDIF

         OTHERWISE
            ? "Key:", nKey

         ENDCASE
      ENDDO

      WAClose()
   RETURN
```

Word()

Converts a floating point number to a 32-it integer value.

Syntax

```
Word( <nNumber> ) --> nInteger
```

Arguments

<nNumber>

A numeric floating point value in the range of -1*(2^31) to (2^31)-1.

Return

The function returns an integer value.

Description

Word() is a compatibility function and not recommended to use. Use function Int() to create integer numbers.

Info

See also:	Bin2L(), Int(), L2Bin()
Category:	Conversion functions
Source:	rtl\word.c
LIB:	xhb.lib
DLL:	xhbdll.dll

WordOne()

Removes duplicate adjacent words (2-byte sequences) from a string.

Syntax

```
WordOne( <cWordList>, <cString> ) --> cResult
```

Arguments

<cWordList>

This is a character string holding 2-byte sequences (words) to search for in *<cString>*.

<cString>

This is the character string to process.

Return

The function searches for duplicate adjacent 2-byte sequences (words) in *<cString>* and removes them in the result string.

Info

See also:	CharOne(), I2Bin()
Category:	CT:String manipulation, Character functions
Source:	ct\charone.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays results of WordOne()

   PROCEDURE Main
      LOCAL cString := "abab12cd1212cdefef"

      ? WordOne( "12"  , cString )   // result: abab12cd12cdefef

      ? WordOne( "abef", cString )   // result: ab12cd1212cdef

   RETURN
```

WordOnly()

Removes all words (2-byte sequence) but the specified ones from a string

Syntax

```
WordOnly( <cWordList>, <cString> ) --> cResult
```

Arguments

<cWordList>

This is a character string holding 2-byte sequences (words) to ignore in *<cString>*.

<cString>

This is the character string to process.

Return

The function searches for 2-byte sequences (words) in *<cString>* and removes all of the words in the result string that do not exist in *<cWordList>*.

Info

See also: CharOnly()
Category: CT:String manipulation, Character functions
Source: ct\charonly.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays results of WordOnly()

   PROCEDURE Main
      LOCAL cString := "abab12cd1212cdefef"

      ? WordOnly( "12"  , cString )   // result: 121212

      ? WordOnly( "abef", cString )   // result: ababefef

   RETURN
```

WordRem()

Deletes specified words (2-byte sequence) from a string.

Syntax

```
WordRem( <cWordList>, <cString> ) --> cResult
```

Arguments

`<cWordList>`

This is a character string holding 2-byte sequences (words) to remove from <*cString*>.

`<cString>`

This is the character string to process.

Return

The function searches for 2-byte sequences (words) in <*cString*> and removes all of the words in the result string that exist in <*cWordList*>.

Info

See also:	WordOne(), WordOnly()
Category:	CT:String manipulation, Character functions
Source:	ct\charonly.c
LIB:	xhb.lib
DLL:	xhbdll.dll

WordRepl()

Replaces words (2-byte sequences) in a string with a specified word.

Syntax

```
WordRepl( <cWordList> , ;
          <cString>   , ;
          <cReplace   , ;
          [<lSkipWords>]) --> cResult
```

Arguments

<cWordList>

This is a character string holding 2-byte sequences (words) to search for in <*cString*>.

<cString>

This is the character string to process.

<cReplace>

This is a character string holding 2-byte sequences (words) which replace the corresponding words of <*cWordList*> in <*cString*>.

<lSkipWords>

This parameter defaults to .F. (false) so that the function scans <*cString*> in steps of one byte. When set to .T. (true), the function scans the string in steps of two bytes. This is of relevance when CSetAtMuPa() is set to .T. (true).

Return

The function scans <*cString*> and searches the words specified with <*cWordList*>. All words found are replaced with the corresponding word listed in <*cReplace*>. The modified character string is returned.

Info

See also:	CSetAtMuPa(), CharRepl(), CSetRef()
Category:	CT:String manipulation, Character functions
Source:	ct\wordrepl.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example displays results of WordRepl()

   PROCEDURE Main
      LOCAL cString := "abab12cd1212cdefef"

      ? WordRepl( "12", cString, "XY" )
                       // result: ababXYcdXYXYcdefef

      ? WordRepl( "abef", cString, "RSTU" )
                       // result: RSRS12cd1212cdTUTU
   RETURN
```

WordSwap()

Exchanges adjacent words (2-byte sequences) in a string.

Syntax

```
WordSwap( <cString>, [<lSwapBytes>] ) --> cResult
```

Arguments

`<cString>`

This is the character string to process.

`<lSwapBytes>`

This parameter defaults to .F. (false) so that the function exchanged adjacent 2-byte sequences (words) in *<cString>*. When set to .T. (true), the function also exchanges the high and low bytes in each word.

Return

The function returns the modified character string.

Info

See also:	CharSort(), CharSwap()
Category:	CT:String manipulation, Character functions
Source:	ct\charswap.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example outlines the effect of WordSwap() with
// the second parameter set to .F. and .T.

   PROCEDURE Main
      LOCAL cString := "ABCD"

      ? WordSwap( cString )            // result: CDAB

      ? WordSwap( cString, .T. )       // result: DCBA
   RETURN
```

WordToChar()

Replaces words (2 byte sequence) with characters (1 byte)

Syntax

```
WordToChar( <cWordList>, ;
            <cString>  , ;
            <cCharList>  ) --> cResult
```

Arguments

<cWordList>

This is a character string holding 2-byte sequences (words) to search for in *<cString>*.

<cString>

This is the character string to process.

<cCharList>

This is a character string holding the characters (1-byte) which replace the corresponding words of *<cWordList>* in *<cString>*.

Return

The function scans *<cString>* for all words (2-byte sequence) contained in *<cWordList>* and replaces them with the corresponding character specified with *<cCharList>*. The modified character string is returned.

Info

See also: CSetAtMuPa(), CharRepl(), WordRepl()
Category: CT:String manipulation, Character functions
Source: ct\wordtoch.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example displays results of WordToChar()

   PROCEDURE Main
      LOCAL cString := "abab12cd1212cdefef"

      ? WordToChar( "12", cString, "_" )        // result: abab_cd__cdefef

      ? WordToChar( "abcdef", cString, "XYZ" )  // result: XX12Y1212YZZ

   RETURN
```

WRow()

Returns the top row position of the selected window.

Syntax

```
WRow( [<lCentered>] ) --> nTopRow
```

Arguments

<lCentered>

This parameter defaults to .F. (false) which causes the function to return the top row position of the selected window. When .T. (true) is passed, the function returns a row position as if the window was centered.

Return

WLastRow() returns the top row position of the current window, or the position of a centered window, as a numeric value.

Info

See also: WCol(), WInfo(), WLastCol(), WLastRow(), WMode(), WOpen(), WSelect()
Category: CT:Window, Windows (text mode)
Source: ct\ctwin.c
LIB: xhb.lib
DLL: xhbdll.dll

WSelect()

Returns and/or selects an open window as the current window.

Syntax

```
WSelect( [<nWindowID>] ) --> nCurrentWindow
```

Arguments

<nWindowID>

If a numeric window ID is passed, the corresponding window becomes current and all screen output is directed to the current window.

Return

The function returns the window ID of the current window as a numeric value.

Note: the window ID zero identifies the entire screen. WSelect(0) selects the entire screen as the current window.

Info

See also: WOpen(), WStack()
Category: CT:Window, Windows (text mode)
Source: ct\ctwin.c
LIB: xhb.lib
DLL: xhbdll.dll

WSetMouse()

Determines the visibility and/or position of the mouse cursor.

Syntax

```
WSetMouse( [<lOnOff>], [<nRow>], [<nCol>] ) --> lIsMouseVisible
```

Arguments

`<lOnOff>`

A logical value can be passed. .T. (true) makes the mouse cursor visible in the current window and .F. (false) hides it.

`<nRow>`

A numeric value between 0 and MaxRow() specifying the new row position of the mouse cursor in the current window.

`<nCol>`

A numeric value between 0 and MaxCol() specifying the new columnn position of the mouse cursor in the current window.

Return

The function returns .T. (true) when the mouse cursor is visible, otherwise .F. (false). If numeric values are passed, the mouse cursor is positioned accordingly in the current window.

Info

See also:	WMSetPos()
Category:	CT:Window, Mouse functions, Windows (text mode), xHarbour extensions
Source:	ct\ctwin.c
LIB:	xhb.lib
DLL:	xhbdll.dll

WSetMove()

Toggles the setting for moving windows interactively.

Syntax

```
WSetMove( [<lNewMoveMode>] ) --> lOldMoveMode
```

Arguments

<lNewMoveMode>

A logical value can be passed that defines the new mode for moving windows interactively.

Return

The function returns the previous setting as a logical value.

Description

WSetMove() maintains a single logical value indicating if windows can be moved interactively. The move operation must be implemented using WMove(). Refer to WOpen() for an example of moving windows.

Info

See also: WBoard(), WMode(), WMove()
Category: CT:Window, Windows (text mode)
Source: ct\ctwin.c
LIB: xhb.lib
DLL: xhbdll.dll

WSetShadow()

Defines the shadow color for windows.

Syntax

```
WSetShadow( [<xNewColor> ) --> nOldColor
```

Arguments

<xNewColor>

This is either a single color value (see SetColor()), or its numeric color attribute (see ColorToN()). It defines the color for drawing a shadow for all windows open after the shadw color is defined. The value -1 turns shadowing off.

Return

The function returns the previous shadow color attribute as a numeric value.

Info

See also: NtoColor(), WOpen()
Category: CT:Window, Windows (text mode)
Source: ct\ctwin.c
LIB: xhb.lib
DLL: xhbdll.dll

WStack()

Returns window IDs of all open windows.

Syntax

```
WStack() --> aWindowIDs
```

Return

The function returns an array holding the window IDs of all open windows. Window IDs are stacked in the array in the sequence they were opened with WOpen(). The first element of the array is the window ID of the entire screen, while the last element is the ID of the last window opened. When a window is closed, its ID is removed from the array.

Info

See also:	WOpen(), WSelect()
Category:	CT:Window, Windows (text mode), xHarbour extensions
Source:	ct\ctwin.c
LIB:	xhb.lib
DLL:	xhbdll.dll

WStep()

Sets the increments for horizontal and vertical window movement.

Syntax

```
WStep( <nVertical>, <nHorizontal> ) --> nErrorCode
```

Arguments

<nVertical>

A numeric value between 1 and 6 indicating the vertical step size for interactive window movement.

<nHorizontal>

A numeric value between 1 and 20 indicating the horizontal step size for interactive window movement.

Return

The function returns 0 on success and -1 otherwise.

Info

See also: WSetMove(), WAClose()
Category: CT:Window, Windows (text mode)
Source: ct\ctwin.c
LIB: xhb.lib
DLL: xhbdll.dll

XtoC()

Converts values of data type C, D, L, M, N to a string.

Syntax

```
XtoC( <xValue> ) --> cValue
```

Arguments

<xValue>

This is a value of data type C, D, L, M or N.

Return

The function returns the converted value as a character string, or a null string ("") when an invalid value is passed.

Info

See also:	CtoF(), CStr()
Category:	CT:Miscellaneous, Miscellaneous functions
Source:	ct\misc1.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Year()

Extracts the numeric year from a Date value

Syntax

```
Year( <dDate> ) --> nYear
```

Arguments

<dDate>

This is a Date value to extract the year from.

Return

The function returns a four digit year as a numeric value, or zero when *<dDate>* is an empty date.

Description

Year() extracts the year from a date value as a four digit number, without recognizing the settings SET DATE or SET CENTURY. Other parts of a date value can be extracted using function Day() or Month().

Info

See also:	CDoW(), CMonth(), CtoD(), Day(), DtoC(), Hour(), Minute(), Month(), Secs()
Category:	Conversion functions, Date and time
Source:	rtl\dateshb.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows results of the function Year()

   PROCEDURE Main

     ? Date()                           // result: 05/10/06
     ? Year( Date() )                   // result: 2006
     ? Year( Date() ) - 19              // result: 1987

   RETURN
```

Operator Reference

$

Substring operator (binary): search substring in string.

Syntax

```
<cSubString> $ <cString>
<xExp> $ <aArray>
```

Arguments

<cSubString>

<*cSubString*> is a character value that is searched for in <*cString*>.

<cString>

<*cString*> is a character value where <*cSubString*> is searched in.

<xExp>

<*xExp*> is the expression of any type to search.

<aArray>

<*aArray*> is the array to scan for a matching <*xExp*> in.

Description

The substring operator performs a case-sensitive search and returns .T. (true) when <*cSubString*> is found in <*cString*>, otherwise the result is .F. (false).

The $ operator can also be used to search for a given value within any Array.

Info

See also: <, <=, <> != #, = (comparison), ==, >, >=, IN, HAS, LIKE
Category: Character operators, Comparison operators, Operators
DLL: xhbdll.dll

Example

```
// The example demonstrates a case-sensitive search.

   PROCEDURE Main()
      LOCAL cString := "xHarbour"

      ? "X"   $ cString                // result: .F.
      ? "arb" $ cString                // result: .T.
      ? 1 $ { 3, 2, 1 }                // result: .T.
   RETURN
```

%

Modulus operator (binary): calculates the remainder of a division.

Syntax

```
<nNumber1> % <nNumber2>
```

Arguments

<nNumber1>

<nNumber1> is the dividend of the division.

<nNumber2>

<nNumber2> is the divisor of the division.

Description

The modulus operator returns the remainder of dividing *<nNumber1>* by *<nNumber2>*.

Info

See also:	*, **, +, -, /, = (compound assignment), SET FIXED
Category:	Mathematical operators, Operators
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example calculates the remainder of a division
// between two numbers.

   PROCEDURE Main
      ? 5 % 0                         // result:  0
      ? 3 % 2                         // result:  1
      ? -2 % 3                        // result: -2
      ? 4 % 2                         // result:  0

      ? 5.4 % 2.3                     // result: 0.8
   RETURN
```

& (bitwise AND)

Bitwise AND operator (binary): performs a logical AND operation.

Syntax

```
<cString> & <nMask>|<cMask> --> cCharacter

<nNumber> & <nMask>|<cMask> --> nNumeric
```

Arguments

<cString>

A character expression of which all bits of each character are processed in a logical AND operation.

<nNumber>

A numeric value all bits of which are processed. Numbers are always treated as integer values. If a number has a decimal fraction, it is truncated.

<nMask>|<cMask>

The right operand of the bitwise AND operator can be specified as a character or a numeric value.

Description

The bitwise AND operator performs a logical AND operation with the individual bits of both operands. The left operand is the value to process while the right operand provides the bit mask for the operation. The return value of the &-operator has the same data type as the left operand.

Bits at identical positions in both operands are compared. The bit at the same position is set in the return value, when both operands have the bit set at the same position. If either operand has a bit not set, the corresponding bit in the return value is also not set.

The bit mask to apply to the left operand can be specified as a numeric or as a character value. Depending on the left operand, bitwise AND operations are performed according to the following rules:

cString & cMask

When both operands are character values, the bits of individual characters of both operands are compared. If the right operand has less characters, it is repeatedly applied until all characters of the left operand are processed. The return value has the same number of characters as the left operand.

cString & nMask

When the left operand is a character value and the right operand is numeric, the bits set in the numeric value are compared with the bits of each character in the left operand.

nNumber & cMask

When the left operand is numeric and the right operand is a character value, the bits set in the numeric value are compared consecutively with the bits of each character in the right operand.

nNumber & nMask

When both operands are numeric values, the bits of both values are compared.

Note: Numeric operands are always treated as integer values. If a number has a decimal fraction, it is ignored.

Info

See also: ^^ (bitwise XOR), | (bitwise OR), .AND., .NOT., .OR.
Category: Bitwise operators, Operators, xHarbour extensions
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates bitwise AND operations using
// operands of different data types.

   #define  STAT_OFF    0
   #define  STAT_INIT   1
   #define  STAT_IDLE   2
   #define  STAT_ON     4

   PROCEDURE Main
      LOCAL nStatus := STAT_INIT + STAT_IDLE

      ? 2 & 3                          // result: 2

      ? "A" & 64                       // result: "@"

      ? 64 & "A"                       // result: 64

      ? "Z" & "A"                      // result: "@"

      ? nStatus & STAT_IDLE            // result: 2

      ? nStatus & STAT_ON              // result: 0

      ? "One" & "Two"                  // result: "Dfe"
   RETURN
```

& (macro operator)

Macro operator (unary): compiles a character string at runtime.

Syntax

```
a) &<cVarName>

b) "Text &<cVarName>. substitution"

c) &<cExpression>

d) <objVar>:&<cIVarName> [:= <xValue>]
   <objVar>:&<cMethodName>([<params,...>])

e) &<cFunctionName>([<params,...>])
```

Arguments

<cVarName>

This is a character string indicating the name of a variable whose symbolic name is known at runtime. This applies to variables of type PRIVATE, PUBLIC or FIELD.

<cExpression>

A character string holding a valid xHarbour expression to compile at runtime.

<objVar>

This is a variable holding a reference to an object value (Valtype() == "O") to which a message is sent.

<cIVarName>

A character string indicating the name of an instance variable of the object. If no assignment operator is used in the expression, the value of the instance variable is retrieved by the macro operator. Otherwise, the value *<xValue>* is assigned to the instance variable.

<cMethodName>

A character string indicating the name of a method of the object to execute. Calling a method with the macro operator requires the parentheses () be added to the expression. Optionally, a list of parameters can be specified within the parentheses. The list of *<params,...>* is passed to the method.

<cFunctionName>

A character string indicating the name of a function or procedure to execute. Calling a function with the macro operator requires the parentheses () be added to the expression. Optionally, a list of parameters can be specified within the parentheses. The list of *<params,...>* is passed to the function or procedure.

Description

One extremely powerful feature of xHarbour is the Macro operator (&). It allows for compiling and executing program code at runtime of an application. The program code is supplied in form of a character string and can be as simple as a variable name or may include very complex expressions. Despite its power, there are some limitations a programmer must be aware of when using the Macro operator.

As a first rule, no white space characters may follow the macro operator. This is required to distinguish the & sign from the bitwise AND operator. The macro expression must follow the & operator immediately.

All entities that do not have a symbolic name at runtime of an application cannot be accessed by the Macro operator. This applies to memory variables of type GLOBAL, LOCAL and STATIC, to STATIC declared functions and procedures, as well as commands and statements. Also, the Macro operator cannot resolve multiple line expressions, it can only compile expressions programmed within one line of code.

As indicated in the syntax description, there are some typical scenarios where the Macro operator is used. Examples for these scenarios follow:

a) Acessing and creating variables

The values of PRIVATE, PUBLIC and FIELD variables can be retrieved by using a character string indicating the symbolic name of a variable.

```
LOCAL cPrivate := "myPrivate"
LOCAL cPublic  := "myPublic"
LOCAL cField   := "LastName"
LOCAL x, y, z

PUBLIC  myPublic  := "A public variable"
PRIVATE myPrivate := 10

USE Customer

x := &cPrivate
y := &cPublic
z := &cField

? x, y, z
// result: 10 A public variable Miller
```

The macro operator is used in this example as the right-hand-side expression of the assignments. As a result, the values of the variables whose names are stored in LOCAL variables are retrieved and assigned to the LOCAL variables x, y and z.

When the macro operator is used on the left-hand-side of an assignment, a value is assigned to the corresponding variable. If the variable does not exist prior to the assignment, it is created as a PRIVATE variable:

```
LOCAL cPrivate := "myPrivate"
LOCAL cPublic  := "myPublic"
LOCAL cField   := "LastName"

PUBLIC  myPublic  := "A public variable"

USE Customer

&cPrivate := 10           // creates a PRIVATE variable
                          // named myPrivate
&cPublic  := "New text"
FIELD->&cField := "McBride"

? &cPrivate, &cPublic, &cField
// result: 10 New text McBride
```

Note that PUBLIC variables cannot be created using this technique while field variables must be aliased to assign a value.

b) Text substitution

Text substitution is a special situation where the macro operator appears within a character string.

```
LOCAL cText

PRIVATE cMacro := "xHarbour"
```

```
cText := "This is a &cMacro. test."
? cText
// result: This is a xHarbour test."

cMacro := CDoW(Date())

cText := "Today is &cMacro.!"
? cText
// result: Today is Tuesday!"
```

When a character string contains a macro variable, the macro is substituted with the contents of the macro variable. Note that the end of the macro variable within the text string is indicated with a period. This period does not appear in the result string but serves as "end-of-macro" marker.

c) Runtime compilation

The most common situation for using the Macro operator is the compilation of expressions at runtime. Expressions must be syntactically correct and may only refer to entities whose symbolic names are known at runtime. If, for example, a function is called only within a Macro expression and nowhere else in the program code, the function symbol must be declared with the REQUEST statement so that it is linked to the executable file.

```
PROCEDURE Main
   REQUEST CDoW, Date

   LOCAL cExpr := "CDoW(Date())"

   ? &cExpr
   // result: Tuesday
RETURN
```

The creation of code blocks as complex data types is also a very common usage scenario for the Macro operator.

```
PROCEDURE Main
   LOCAL i, cExpr, cFPos
   REQUEST FieldGet, FieldPut

   USE Customer

   aBlocks := Array( FCount() )
   FOR i:=1 TO FCount()
      cFPos := LTrim( Str(i) )
      cExpr := "{|x| IIf(x==NIL,FieldGet(" + cFPos + ")," + ;
                                "FieldPut(" + cFPos + ",x)) }"
      aBlocks[i] := &cExpr
   NEXT

   AEval( aBlocks, {|b| QOut(Eval(b)) } )
   USE
RETURN
```

Within the FOR..NEXT loop, the syntax for code blocks accessing different field variables is created as character strings. Each string differs by the value of the loop counter variable which identifies individual field variables by their ordinal position. The character strings are compiled to code blocks and stored in an array.

d) Sending messages to objects

The possibility of sending messages to objects is another field of Macro operator usage that allows for highly dynamic application development. Messages are encoded as character strings:

```
#include "hbclass.ch"
```

```
PROCEDURE Main
   LOCAL obj  := TestClass():new( "Text to display" )
   LOCAL cMsg := "display"

   obj:&cMsg()           // displays: Text to display

   cMsg := "cValue"

   obj:&cMsg := "New text"

   obj:display()        // displays: New text
RETURN

CLASS TestClass
   EXPORTED:
   DATA cValue
   METHOD new( cString ) CONSTRUCTOR
   METHOD display
ENDCLASS

METHOD new( cString ) CLASS TestClass
   ::cValue := cString
RETURN self

METHOD display() CLASS TestClass
   ? ::cValue
RETURN self
```

When the send operator (:) is followed by the Macro operator, the contents of the variable following the Macro operator is interpreted as message to send to the object. Note that calling an object's method requires the parentheses be added to the expression.

e) Calling functions

There are two possibilities for calling functions using the macro operator: one is to encode a complete function call as a macro expression, the other is to use only the function name. The difference between both possibilities lies in the type of variables that can be passed to a Macro operator called function.

```
PROCEDURE Main
   LOCAL cMacro
   LOCAL nValue := 2

   PRIVATE nNumber := 3

   cMacro := "TestFunc( nNumber )"  // result: 30

   ? &cMacro

   cMacro := "TestFunc"

   ? &cMacro( nValue )              // result: 20
RETURN

FUNCTION TestFunc( n )
RETURN 10 * n
```

When the macro expression includes parentheses for the function call, only PRIVATE, PUBLIC or FIELD variables can be passed as parameters, since the variable names appear in the macro string. LOCAL variables, in contrast, can be passed when the parentheses are "hard coded", i.e. the macro string contains only the name of the function to call.

Info

See also:	@(), ( ), {|| }, HB_MacroCompile(), HB_SetMacro()
Category:	Indirect execution, Special operators, Operators
LIB:	xhb.lib
DLL:	xhbdll.dll

()

Execution or grouping operator.

Syntax

```
(<expr,...>)
<symbol>([<params,...>])
<objVar>:<symbol>([<params,...>])
```

Arguments

`<expr,...>`

One or more expressions to execute, separated with commas.

`<symbol>`

This is the symbolic name of a function, method or procedure to execute.

`<params,...>`

An optional comma separated list of parameters to pass to the called function, method or procedure

`<objVar>`

This is a variable holding a reference to an object value (Valtype() == "O") that should execute a method.

Description

Parentheses are used to execute functions, procedures or methods, and to group expressions and define their order of execution.

When using the parentheses as a grouping operator, all items appearing inside the parentheses must be valid expressions. The grouping operator can be nested to any depth. The nesting level influences execution order of expressions, i.e. expressions on a deeper nesting level are evaluated first while expressions with the same nesting level are evaluated from left to right.

When the parentheses are used as execution operator, expressions between the parentheses are passed as arguments to the called function, method or procedure. Multiple expressions must be separated with commas. When no expression appears within the parentheses, no argument is passed.

Info

See also:	& (macro operator)
Category:	Special operators, Operators
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates different scenarios for using
// the () operator.

   PROCEDURE Main
      LOCAL x

      // changing precedence of operations
      ?  5 + 2 * 2                  // result: 9
      ? (5 + 2)* 2                  // result: 14
      ?  5 + 2 * 2 - 1              // result: 8
      ?  5 +(2 *(2 - 1))            // result: 7
```

```
   x := "  xHarbour  "

   // one function call
   ? AllTrim( x )                  // result: "xHarbour"

   // list of function calls
   ? ( x := Date(), CDow(x) )      // result: "Wednesday"
RETURN
```

*

Multiplication operator (binary): multiplys numeric values.

Syntax

```
<nNumber1> * <nNumber2>
```

Arguments

<nNumber1>

<nNumber1> is numeric expression to multiply with *<nNumber2>*.

<nNumber2>

<nNumber2> is numeric expression to multiply with *<nNumber1>*.

Description

This operator multiplies the value of *<nNumber1>* with the value of *<nValue2>* and returns the result.

Info

See also: %, **, +, -, /, = (compound assignment), SET DECIMALS, SET FIXED
Category: Mathematical operators, Operators
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example shows the behavior of the "*" operand in different
// situations.

   PROCEDURE Main
      ?  3 *  1                      // result: 3
      ?  3 * -1                      // result: -1
      ? -4 * -2                      // result: 8
   RETURN
```

**

Exponentiation (binary): raises a number to the power of an exponent.

Syntax

```
<nNumber> ** <nExponent>
<nNumber> ^ <nExponent>
```

Arguments

`<nNumber>`

<nNumber> is numeric value to raise to the power defined by *<nExponent>*.

`<nExponent>`

<nExponent> is the power to raise *<nNumber1>*.

Description

This operator is a binary operator that raises *<nNumber>* to the power of *<nExponent>*.

Info

See also: %, *, +, -, /, = (compound assignment), SET DECIMALS, SET FIXED
Category: Mathematical operators, Operators
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example shows some results of the "**" operator.

   PROCEDURE Main
      ? 5 ** 0                    // result: 1.00
      ? 5 ** 1                    // result: 5.00
      ? 5 ** 2                    // result: 25.00
      ? 5 ** -2                   // result: 0.04
      ? -5 ** 2                   // result: 25.00

      ? 5 ^ 0                     // result: 1.00
      ? 5 ^ 1                     // result: 5.00
      ? 5 ^ 2                     // result: 25.00
      ? 5 ^ -2                    // result: 0.04
      ? -5 ^ 2                    // result: 25.00
   RETURN
```

+

Plus operator: add values, concatenate values and unary positive.

Syntax

```
<nNumber1> + <nNumber2>
<dDate> + <nNumber>
<dDateTime> + <nNumber>
<cString1> + <cString2>
<hHash1> + <hHash2>
```

Arguments

`<nNumber1>`

<nNumber1> is a numeric value to which the value of *<nNumber2>* is added.

`<dDate>`

<dDate> is a date value to which *<nNumber>* days are added.

`<dDateTime>`

<dDateTime> is a DateTime value to which *<nNumber>* days are added.

`<cString2>`

<cString2> is a character string to concatenate with *<cString1>*.

Description

Depending on the data types of the operands, the Plus operator performs different operations.

Unary positive sign

A numeric expression preceeded by the "+" operator performs no operation on the operand.

Numeric addition

When both operands are numeric values, the right operand *<nNumber2>* is added to the left operand *<nNumber1>* and the result is a numeric value.

Date addition

When either operand is a date value and the other is a numeric, the value of *<nNumber>* is added as days to *<dDate>*. The returned value is a date.

DateTime addition

When either operand is a DateTime value and the other is a numeric, the value of *<nNumber>* is added as days to *<dDateTime>*. The result is a DateTime value. To add hours or minutes, the numeric operand must be coded as fraction of a day. E.g. *<dDateTime>* + 1/24 adds one hour to *<dDateTime>* (refer to the DateTime operator).

String concatenation

If both operands are character strings, the value of *<cString2>* is joined to the end of *<cString1>*. The result is a character string containing both operands.

Hash operation

If both operands are hashes, a set-oriented operation is performed so that the resulting hash value contains unique keys of both operands (refer to the hash operator).

Info

See also: %, *, **, ++, -, /, = (compound assignment), {=>}, {^ }
Category: Character operators, Mathematical operators, Operators
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates variations with the + operator:

   PROCEDURE Main
      // Unary positive sign (also included is the negative sign)
      ? 1 + + 1                          // result: 2
      ? 1 + - 1                          // result: 0
      ? 1 - - 1                          // result: 2

      // Addition
      ? 10 + 2                           // result: 12
      ? CtoD("01/01/2005") + 5           // result: 01/06/2005
      ? 31 + CtoD("01/01/2005")          // result: 02/01/2005

      ? {^ 2007/04/26 15:30:00 }         // result: 04/26/07 15:30:00.00
      ? {^ 2007/04/26 15:30:00 }+1       // result: 04/27/07 15:30:00.00
      ? {^ 2007/04/26 15:30:00 }+1/24    // result: 04/26/07 16:30:00.00
      ? {^ 2007/04/26 15:30:00 }+1/86400 // result: 04/26/07 15:30:01.00

      // String concatenation
      ? "Visit" + " " + "xHarbour.com"  // result: "Visit xHarbour.com"
   RETURN
```

++

Increment operator (unary): prefix / postfix increment.

Syntax

```
++ <Variable>
<Variable> ++
```

Arguments

```
<Variable>
```

<Variable> is the name of a memory or field variable of Numeric or Date data type. When *<Variable>* is a field variable, it must be specified with an alias name or must be declared as field variable using the FIELD statement.

Description

The increment operator increases the value of its operand by one. When used in an expression, the position of the operator is important for the result of the expression.

When the operator appears to the left of the operand (prefix notation), the operand's value is first incremented and then used in the expression.

When the operator appears to the right of the operand (postfix notation), the operand's value is first used in the expression and then incremented.

Info

See also:	+, --, :=, = (compound assignment)
Category:	Mathematical operators, Operators
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates the use of the ++ operator and
// outlines the importance of prefix and postfix notation.

PROCEDURE Main
   LOCAL nValue1 := 0
   LOCAL nValue2

   ? nValue1 ++                    // result: 0
   ? nValue1                       // result: 1

   ? ++ nValue1                    // result: 2
   ? nValue1                       // result: 2

   nValue2 := ++ nValue1
   ? nValue1                       // result: 3
   ? nvalue2                       // result: 3

   nValue2 := nValue1 ++
   ? nValue1                       // result: 4
   ? nvalue2                       // result: 3
RETURN
```

-

Minus operator: add values, concatenate values and unary negative.

Syntax

```
<nNumber1> - <nNumber2>
<dDate1> - <dDate2>
<dDate> - <nNumber>
<dDateTime> - <nNumber>
<cString1> - <cString2>
<hHash1> - <hHash2>
```

Arguments

<nNumber1>

<nNumber1> is a numeric value from which the value of *<nNumber2>* is subtracted.

<dDate1>

<dDate1> is a date value from which the date value *<dDate2>* is subtracted.

<dDate>

<dDate> is a date value from which *<nNumber>* days are subtracted.

<dDateTime>

<dDate> is a DateTime value from which *<nNumber>* days are subtracted.

<cString2>

<cString2> is a character string to add to the end of *<cString1>* after all trailing blanks from *<cString1>* are removed.

Description

Depending on the data types of the operands, the Minus operator performs the following operations:

Unary negative sign

when the Minus operator precedes a numeric operand, it performs the equivalent of multiplying the operand by -1. This changes the operand's sign from plus to minus and vice versa.

Numeric subtraction

When both operands are numeric values, the right operand *<nNumber2>* is subtracted from the left operand *<nNumber1>* and the result is a numeric value.

Date subtraction

When the left operand is a date value and the right operand is a numeric, the value of *<nNumber>* is subtracted as number of days from *<dDate>*. The returned value is a date.

When both operands are of Date data type, the operator calculates the difference in days between left and right operand and returns a numeric value.

When the left operand is a Numeric value and the right operand is a Date, a runtime error is raised since this is not allowed.

DateTime subtraction

When the left operand is a DateTime value and the right operand is a numeric, the value of *<nNumber>* is subtracted as number of days from *<dDateTime>*. The returned value is a date. To

subtract hours or minutes, the numeric operand must be coded as fraction of a day. E.g. *<dDateTime>* - 1/24 subtracts one hour from *<dDateTime>* (refer to the DateTime operator).

When both operands are of DateTime data type, the operator calculates the difference in days between left and right operand and returns a numeric value.

When the left operand is a Numeric value and the right operand is a DateTime, a runtime error is raised since this is not allowed.

String concatenation

If both operands are character data types, the value of *<cString2>* is joined to *<cString1>*, returning a character string. Trailing blanks in *<cString1>* are removed and appended to the end of the result string.

Hash operation

If both operands are hashes, a set-oriented operation is performed so that the resulting hash value contains unique keys of the left operand which are not contained in the right operand (refer to the hash operator).

Info

See also:	%, *, **, +, --, /, = (compound assignment), {=>}, {^ }
Category:	Character operators, Mathematical operators, Operators
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates variations with the - operator:

   PROCEDURE Main
      // Unary negative sign (also included is the positive sign)
      ? 1 - - 1                        // result: 2
      ? 1 + - 1                        // result: 0
      ? 1 + + 1                        // result: 2

      // Subtraction
      ? CtoD("01/20/2005") - 5         // result: 01/15/2005
      ? 10 - 2                         // result: 8

      ? {^ 2007/04/26 15:30:00 }          // result: 04/26/07 15:30:00.00
      ? {^ 2007/04/26 15:30:00 }-1        // result: 04/25/07 15:30:00.00
      ? {^ 2007/04/26 15:30:00 }-1/24     // result: 04/26/07 14:30:00.00
      ? {^ 2007/04/26 15:30:00 }-1/86400  // result: 04/26/07 15:29:59.00

      // String concatenation
      ? "A " + "B " + "C "             // result: "A B C "
      ? "A " - "B " - "C "             // result: "ABC   "
   RETURN
```

--

Decrement operator (unary): Prefix / postfix decrement

Syntax

```
-- <Variable>
<Variable> --
```

Arguments

```
<Variable>
```

<Variable> is the name of a memory or field variable of Numeric or Date data type. When *<Variable>* is a field variable, it must be specified with an alias name or must be declared as field variable using the FIELD statement.

Description

The decrement operator decreases the value of its operand by one. When used in an expression, the position of the operator is important for the result of the expression.

When the operator appears to the left of the operand (prefix notation), the operand's value is first decremented and then used in the expression.

When the operator appears to the right of the operand (postfix notation), the operand's value is first used in the expression and then decremented.

Info

See also:	++, -, :=, = (compound assignment)
Category:	Mathematical operators, Operators
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates the use of the -- operator and
// outlines the importance of prefix and postfix notation.

PROCEDURE Main
   LOCAL nValue1 := 0
   LOCAL nValue2

   ? nValue1 --                    // result: 0
   ? nValue1                       // result: -1

   ? -- nValue1                    // result: -2
   ? nValue1                       // result: -2

   nValue2 := -- nValue1
   ? nValue1                       // result: -3
   ? nvalue2                       // result: -3

   nValue2 := nValue1 --
   ? nValue1                       // result: -4
   ? nvalue2                       // result: -3
RETURN
```

->

Alias operator (binary): identifies a work area.

Syntax

```
<aliasName>   -> <fieldName>
(<nWorkArea>) -> <fieldName>

 <aliasName>  -> ( <expr,...> )
(<nWorkArea>) -> ( <expr,...> )

FIELD  -> <fieldName>
MEMVAR -> <varName>
```

Arguments

<aliasName>

This is the symbolic alias name of a work area as specified with the ALIAS option of the USE command.

<fieldName>

The field name whose value is retrieved from the work area specified with *<aliasName>*.

<nWorkArea>

Instead of using a symbolic name, a work area can be identified by its numeric ordinal position *<nWorkArea>*. This value can be obtained by calling the Select() function. The numeric work area must be enclosed in parentheses () when using it with the alias operator.

<expr,...>

One or more comma separated expressions to execute in the work area identified with *<aliasName>* or *<nWorkArea>*.

FIELD

The FIELD keyword is a reserved alias name that identifies a field variable in the current work area.

MEMVAR

The MEMVAR keyword is a reserved alias name that identifies a memory variable of type PRIVATE or PUBLIC. An abbreviation of this alias is M->.

Description

The alias operator "points" to an unselected work area identified with its symbolic name *<aliasName>* or its ordinal number *<nWorkArea>*. This allows for retrieving values of field variables from unselected work areas or executing expressions in the specified work area. The alias operator implicitly selects the work area specified with the left operand and then executes the instructions of the right operand. When the operation is complete, the original work area becomes current again.

FIELD and MEMVAR are reserved alias names that allow for resolving name conflicts between memory and field variables. If memory and field variables with the same symbolic name exist, unaliased variables are resolved as field variables. This behaviour is influenced by the compiler switch /v. It is, however, strongly recommended to avoid identical variable names for memory and field vraiables, and to use the alias operator to identify a variable unambiguously.

Info

See also: DbSelectArea(), FIELD, FieldName(), FieldPos(), FieldGet(), MEMVAR, Select(), USE
Category: Special operators, Operators
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates usage of the alias operator.
// The last aliased expression generates a runtime error.

   PROCEDURE Main
      LOCAL nArea

      USE Customer ALIAS Cust          // current work area
      ? FIELD->Lastname                // result: Miller

      nArea := Select()                // work area number
      SELECT 100                       // select free work area
      ? Used()                         // result: .F.

      ? Cust->Lastname                 // result: Miller
      ? Cust->(Recno())                // result: 1

      ? (nArea)->Lastname              // result: Miller
      ? (nArea)->(LastRec())           // result: 1576

      ? FIELD->Lastname                // runtime error
      USE
   RETURN
```

.AND.

Logical AND operator (binary).

Syntax

```
<lCondition1> .AND. <lCondition2>
```

Arguments

`<lCondition>`

<lCondition1> and *<lCondition2>* are logical expressions to AND.

Description

The logical .AND. operator yields the result of a logical AND operation with left and right operand. The result is only .T. (true) when the value of both operands is also .T. (true), in all other cases the result is .F. (false).

When multiple .AND. operators appear in an expression, the result is already determined when one operand has the value .F. (false). In this case, remaining .AND. operators are not evaluated for optimization reasons. This so called *shortcut optimization* can be switched off using the compiler switch /z.

Info

See also:	& (bitwise AND), .OR., .NOT.
Category:	Logical operators, Operators
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// This example shows results of .AND. operations
// using different operands:

   PROCEDURE Main
      ? .T. .AND. .T.                  // result: .T.
      ? .T. .AND. .F.                  // result: .F.
      ? .F. .AND. .T.                  // result: .F.
      ? .F. .AND. .F.                  // result: .F.
   RETURN
```

.NOT.

Logical NOT operator (unary).

Syntax

```
! <lCondition>
.NOT. <lCondition>
```

Arguments

```
<lCondition>
```

<lCondition> is a logical expression to logically negate.

Description

The .NOT. operator (alternatively "!" as an abbreviation) is a unary logical operator that negates the value of its operand. This yields the logical inverse of *<lCondition>*.

Info

See also: .AND., .OR.
Category: Logical operators, Operators
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example shows the logical inverse of the operands:

   PROCEDURE Main
      ? .NOT. (.T.)                    // result: .F.
      ? .NOT. (.F.)                    // result: .T.

      ? ! 1 > 2                        // result: .T.
  RETURN
```

.OR.

Logical OR operator (binary).

Syntax

```
<lCondition1> .OR. <lCondition2>
```

Arguments

<lCondition>

<lCondition1> and *<lCondition2>* are logical expressions.

Description

The logical .OR. operator yields the result of a logical OR operation with left and right operand. The result is only .T. (true) when the value of one operand is also .T. (true), in all other cases the result is .F. (false).

When multiple .OR. operators appear in an expression, the result is already determined when one operand has the value .T. (true). In this case, remaining .OR. operators are not evaluated for optimization reasons. This so called *shortcut optimization* can be switched off using the compiler switch /z.

Info

See also:	.AND., .NOT., | (bitwise OR)
Category:	Logical operators, Operators
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// This example shows results of .OR. operations
// using different operands:

   PROCEDURE Main
      ? .T. .OR. .T.                    // result: .T.
      ? .T. .OR. .F.                    // result: .T.
      ? .F. .OR. .T.                    // result: .T.
      ? .F. .OR. .F.                    // result: .F.
   RETURN
```

/

Division operator (binary): divides numeric values.

Syntax

```
<nNumber1> / <nNumber2>
```

Arguments

`<nNumber1>`

<*nNumber1*> is the dividend.

`<nNumber2>`

<*nNumber2*> is the divisor.

Description

This operator returns the division of <*nNumber1*> by <*nNumber2*> as a numeric value.

Info

See also:	%
Category:	Mathematical operators, Operators
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows the use of the "/" operator:

   PROCEDURE Main
      ?  3 / 3                        // result: 1
      ?  3 /-2                        // result: -1.50
      ? -3 / 2                        // result: -1.50
      ? -3 /-2                        // result: 1.50
      ?  3 / 0                        // result: 0
   RETURN
```

:

Send operator (unary): sends a message to an object.

Syntax

```
<object>:<message>[ ( [<params,...>] ) ]
```

Arguments

`<object>`

<object> is a variable holding an object to which the message is sent.

`<message>`

<message> is the name of an instance variable or method. When a method is called on the object, the message name must be followed with parentheses.

`<params,...>`

<params,...> is an optional comma separated list of parameters passed to a method.

Description

A class defines instance variables and methods for its objects. Accessing an object's instance variable or executing one of its methods requires the object to receive a corresponding message. This is the task of the Send operator (:).

To access an instance variable, the message is sent without following parentheses. As a result, the object returns the value of the instance variable that corresponds to the message.

Methods, in contrast, are invoked by adding parentheses to the message and passing an optional list of parameters to the method. Parameters are listed inside the parentheses.

The double send operator :: has a special meaning within the context of the program code of methods. It is an abbreviation for self:, i.e. :: sends messages to the object that executes a method.

Note that only messages are understood by an object that are declared in the object's class. In addition, only those instance variables and methods are accessible that are visible in the current program context. If either an instance variable/method is not declared or currently not visible, the message is not understood by the object and a runtime error is generated.

Info

See also:	CLASS, DATA, METHOD (Implementation)
Category:	Special operators, Operators
Header:	hbclass.ch
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates the use of the single and
// double send operator. Note that the double send operator
// is only used within the program code of a method.

   #include "hbclass.ch"

   PROCEDURE Main
      LOCAL obj := MyClass():new( "Hello World" )

      obj:display()                    // shows: Hello World
```

```
   obj:value := "Hi there"

   obj:display()                    // shows: Hi there
RETURN

CLASS MyClass
   EXPORTED:
   DATA value
   METHOD new( cString ) CONSTRUCTOR
   METHOD display
ENDCLASS

METHOD new( cString ) CLASS MyClass
   ::value := cString
RETURN self

METHOD display CLASS MyClass
  ? ::value
RETURN self
```

:=

Inline assignment to a variable.

Syntax

```
<Variable> := <Expression>
```

Arguments

`<Variable>`

<*Variable*> is the name of a variable of any type.

`<Expression>`

<*Expression*> is any valid expression whose result is assigned to <*Variable*>.

Description

The := operator assigns the value of <*Expression*> to a variable. It is the preferred assignment operator since it can be used to initialize variables within their declaration statements. This is not permitted with the simple assignment operator "=".

If the variable does not exist when the assignment operation is processed, a PRIVATE variable is automatically created and gets assigned the value of <*Expression*>.

Info

See also:	++, --, = (assignment), = (compound assignment)
Category:	Assignment operators
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates the use of the inline assignment
// operator in various situations:

   PROCEDURE Main
      LOCAL dDate  := StoD("20050701") // initializes variables
      LOCAL nMonth := Month( dDate )   // in declaration statement

      nDay := 12                       // creates
      nYear := nPreviousYear := 2000   // PRIVATE variables

      // assignment within expression
      IF (nMonth := Month(dDate)) == 7
         ? "July"
      ENDIF
      ? nMonth                         // result: 7

      // note the removed parentheses and
      // different return value of the expression
      IF nMonth := Month(dDate) == 7
         ? "July"
      ENDIF
      ? nMonth                         // result: .T.
   RETURN
```

<

Less than operator (binary): compares the size of two values.

Syntax

```
<Expression1> < <Expression2>
```

Arguments

```
<Expression>
```

The values of *<Expression1>* and *<Expression2>* must have the same data type and are compared.

Description

The "less than" operator compares two values of the same data type. It returns .T. (true) when the left operand is smaller than the right operand, otherwise the return value is .F. (false).

Only the simple data types Character, Date, Logic, Memo and Numeric can be compared. The complex data types Array, Code block, Hash, Object and the value NIL cannot be used with the "less than" operator.

Comparison rules

Data type	Comparison
Character	The comparison is based on the ASCII value of the characters.
Date	The comparison is based on the underlying date value.
Logical	False (.F.) is smaller than true (.T.).
Memo	Same as Character data type.
Numeric	Comparison is based on the value of the numerics.

SET EXACT

For character comparisons, the less than operator takes the SET EXACT setting into account. With SET EXACT OFF, characters are compared up to the length of the right operand. With SET EXACT ON, characters are compared up to the length of the left operand and trailing blank spaces are ignored.

Info

See also: $, >, <=, <> != #, = (comparison), ==, >=, SET EXACT
Category: Comparison operators, Operators
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates the result of the "<" operator.

   PROCEDURE Main
      ? "A" < "Z"                 // result: .T.
      ? "a" < "Z"                 // result: .F.

      ? CToD("12/28/2005") < CToD("12/31/2005")
                                  // result: .T.
      ? .T. < .F.                 // result: .F.

      ? 2 < 1                     // result: .F.

      SET EXACT OFF
      ? "a" < "a    "             // result: .T.
```

```
    SET EXACT ON
    ? "a" < "a    "              // result: .F.
RETURN
```

<<

Left-shift operator (binary): shifts bits to the left.

Syntax

```
<nNumber1> << <nNumber2>
```

Arguments

`<nNumber1>`

<nNumber1> is a numeric value whose bits are shifted to the left.

`<nNumber2>`

<nNumber2> is a number indicating how many places the bits are shifted to the left.

Description

The << operator accesses individual bits in the left operand and shifts them to the left as many times as specified with the right operand. Both operands are always treated as integer values. If an operand has a decimal fraction, it is truncated prior to the operation.

A shift operation involves all bits of the left operand. When the bits are shifted one place to the left, the highest bit is discarded and the lowest bit is set to 0. A numeric value has 32 bits on a 32 bit operating system.

Shifting a value to the left is equivalent with multiplying *<nNumber1>* by 2 raised to the power of *<nNumber2>*.

Info

See also: >>, HB_BitShift()
Category: Bitwise operators, Operators, xHarbour extensions
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// This example shifts the bits of the value 52 two places to
// the left so the value becomes 208.

   PROCEDURE Main
      nValue := 52           // binary: 00110100
      ? nValue << 2          // binary: 11010000 (value = 208)

      ? 52 * 2 ^ 2           // result: 208
   RETURN
```

<=

Less than or equal operator (binary): compares the size of two values.

Syntax

```
<Expression1> <= <Expression2>
```

Arguments

```
<Expression1>
```

The values of *<Expression1>* and *<Expression2>* must have the same data type and are compared.

Description

The "less than or equal" operator compares two values of the same data type. It returns .T. (true) when the left operand is smaller than or equal to the right operand, otherwise the return value is .F. (false).

Only the simple data types Character, Data, Logic, Memo and Numeric can be compared. The complex data types Array, Code block, Hash, Object and the value NIL cannot be used with the "less than or equal" operator.

Comparison rules

Data type	Comparison
Character	The comparison is based on the ASCII value of the characters.
Date	The comparison is based on the underlying date value.
Logical	False (.F.) is smaller than true (.T.).
Memo	Same as Character data type.
Numeric	Comparison is based on the value of the numerics.

SET EXACT

For character comparisons, the "less than or equal" operator takes the SET EXACT setting into account. With SET EXACT OFF, characters are compared up to the length of the right operand. With SET EXACT ON, characters are compared up to the length of the left operand and trailing blank spaces are ignored.

Info

See also: $, <, <> != #, = (comparison), ==, >, >=, SET EXACT
Category: Comparison operators, Operators
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates the result of the "<=" operator.

   PROCEDURE Main
      ? "A" <= "Z"                // result: .T.
      ? "a" <= "Z"                // result: .F.

      ? CToD("12/28/2005") <= CToD("12/31/2005")
                                  // result: .T.
      ? .T. <= .F.                // result: .F.

      ? 2 <= 1                    // result: .F.

      SET EXACT OFF
```

```
    ? "a"  <= "a    "          // result: .T.
    ? "a " <= "a"              // result: .T.
    ? ""   <= "a"              // result: .T.
    ? "a"  <= ""               // result: .T.

    SET EXACT ON
    ? "a"  <= "a    "          // result: .T.
    ? "a " <= "a"              // result: .T.
    ? ""   <= "a"              // result: .T.
    ? "a"  <= ""               // result: .F.
RETURN
```

<> !=

Not equal operator (binary): compares two values for inequality.

Syntax

```
<Expression1> <> <Expresson2>
<Expression1> != <Expresson2>
<Expression1> # <Expresson2>
```

Arguments

<Expression>

The values of *<Expression1>* and *<Expression2>* must have the same data type or can be NIL, and are compared.

Description

The "not equal" operator compares two values of the same data type or NIL. It returns .T. (true) when the left operand has not the same value as the right operand, otherwise the return value is .F. (false).

Only the simple data types Character, Date, Logic, Memo, Numeric and the value NIL can be compared. The complex data types Array, Code block and Object cannot be used with the "not equal" operator.

Comparison rules

Data type	Comparison
Character	The comparison is based on the ASCII value of the characters.
Date	The comparison is based on the underlying date value.
Logical	False (.F.) is smaller than true (.T.).
Memo	Same as Character data type.
Numeric	Comparison is based on the value of the numerics.
NIL	NIL is equal to NIL and not equal to any other data type.

SET EXACT

For character comparisons, the "not equal" operator takes the SET EXACT setting into account. With SET EXACT OFF, characters are compared up to the length of the right operand. With SET EXACT ON, characters are compared up to the length of the left operand and trailing blank spaces are ignored.

Info

See also: $, <, <=, = (comparison), ==, >, >=, SET EXACT
Category: Comparison operators, Operators
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates the result of the "<>" operator.

   PROCEDURE Main

      SET EXACT ON
      ? "ABC"   <> "ABCDE"            // result: .T.
      ? "ABCDE" <> "ABC"              // result: .T.

      ? "ABC"   <> ""                 // result: .T.
      ? ""      <> "ABC"              // result: .T.
```

```
    SET EXACT OFF
    ? "ABC"   <> "ABCDE"              // result: .T.
    ? "ABCDE" <> "ABC"                // result: .F.

    ? "ABC"   <> ""                   // result: .F.
    ? ""      <> "ABC"                // result: .T.

    ? CToD("12/28/2005") <> CToD("12/31/2005")
                                      // result: .T.

    ? NIL <> NIL                      // result: .F.
    ? 2 <> 1                          // result: .T.
RETURN
```

<| | >

Extended literal code block.

Syntax

```
<| [<params,...>] | <programcode> >
```

Arguments

<params,...>

This is an optional comma separated list of parameter names declared for use inside the extended code block. Code block parameters are only visible within the code block and must be placed between the || delimiters.

<programcode>

<programcode> is any kind of program code which is also allowed within the body of a FUNCTION, except for the declaration of STATIC variables.

Description

Extended literal code blocks are created in program code using the <// > characters. They can be used in the same way as regular code blocks but have the advantage that *<programcode>* can include any statements allowed in the body of a function. This includes statements spanning across multiple lines, such as **loops** (DO WHILE, FOR and FOR EACH), **branching** (DO CASE and SWITCH), **error handling** (BEGIN SEQUENCE and TRY...CATCH).

Even the declaration of LOCAL variables within an extended code block is supported. Only STATIC variables cannot be declared.

The program code, embedded within an extended code block, is executed by passing the code block to the Eval() function. Arguments passed to this function are passed on to the code block and are received by the code block parameters. The expressions and statements in the code block are then executed from left to right, or top to bottom, respectively. The return value of the code block must be specified with the RETURN statement.

Info

See also:	{|| }, AEval(), AScan(), ASort(), DbEval(), Eval(), HEval(), LOCAL
Category:	Indirect execution, Operators, Special operators, xHarbour extensions
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates the creation of an extended code block as a
// return value of function ConversionBlock(). The code block converts
// values to character strings. Note that the code block calls itself
// recursively within the FOR EACH loop.


   PROCEDURE Main
      LOCAL bBlock, lLogic:= .T., nNumber:= 1.23456, aArray := Directory()

      bBlock := ConversionBlock()

      ? Eval( bBlock, aArray )

      ? Eval( bBlock, lLogic )
```

```
   ? Eval( bBlock, nNumber )

   ? Eval( bBlock, GetNew() )

   ? Eval( bBlock, bBLock )
RETURN


FUNCTION ConversionBlock()
   LOCAL  bBlock

   bBlock := <| xValue |
                LOCAL cType := Valtype( xValue )
                LOCAL cValue, xElem

                SWITCH cType
                CASE "A"
                   cValue := "{"
                   FOR EACH xElem IN xValue
                      cValue += Eval( bBlock, xElem ) + ","
                   NEXT
                   cValue[-1] := "}"
                   EXIT

                CASE "B" ; cValue := "{||...}"                ; EXIT
                CASE "C" ; cValue := xValue                   ; EXIT
                CASE "D" ; cValue := DtoS(xValue)             ; EXIT
                CASE "L" ; cValue := IIf(xValue, ".T.",".F.") ; EXIT
                CASE "N" ; cValue := LTrim(Str(xValue))       ; EXIT
                CASE "O" ; cValue := xValue:className()       ; EXIT
                DEFAULT
                   cValue := "NIL"
                END

                RETURN cValue
             >

 RETURN bBlock
```

= (assignment)

Simple assignment operator (binary): assigns a value to a variable.

Syntax

```
<Variable> = <Expression>
```

Arguments

`<Variable>`

<*Variable*> is the name of a variable of any type.

`<Expression>`

<*Expression*> is any valid expression whose result is assigned to <*Variable*>.

Description

The = operator assigns the value of <*Expression*> to a variable. The recommended assignment operator, however, is the inline assignment operator (:=) which allows for initializing a variable within a variable declaration statement. This is not possible with the simple assignment operator. In addition, the simple assignment operator is interpreted as comparison operator within expressions. This can lead to subtle programming errors since the meaning of the simple assignment operator changes with the context it is used in.

If the variable does not exist when the assignment operation is processed, a PRIVATE variable is automatically created and gets assigned the value of <*Expression*>.

Info

See also:	++, --, :=, = (compound assignment)
Category:	Assignment operators, Operators
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates the use of the assignment operator
// and outlines the difference between simple and inline assignment.

   PROCEDURE Main
      LOCAL nMonth

      nMonth = 9                       // simple assignment
      ? nMonth = 9                     // result: .T.

      IF (nMonth := 10) > 0            // inline assignment
         ? "October"                   // output: October
      ENDIF

      IF (nMonth = 9) > 0              // runtime error
         ? "September"                 // (nMonth = 9) -> .F.
      ENDIF
   RETURN
```

= (comparison)

Equal operator (binary): compares two values for equality.

Syntax

```
<Expression1> = <Expression2>
```

Arguments

<Expression>

The values of *<Expression1>* and *<Expression2>* must have the same data type or can be NIL, and are compared.

Description

The "equal" operator compares two values of the same data type or NIL. It returns .T. (true) when the left operand has the same value as the right operand, otherwise the return value is .F. (false).

Only the simple data types Character, Data, Logic, Memo, Numeric and the value NIL can be compared. The complex data types Array, Code block, Hash and Object cannot be used with the "equal" operator. This is only possible with the "exact equal" operator.

Comparison rules

Data type	Comparison
Character	The comparison is based on the ASCII value of the characters.
Date	The comparison is based on the underlying date value.
Logical	False (.F.) is not equal to true (.T.).
Memo	Same as Character data type.
Numeric	Comparison is based on the value of the numerics.
NIL	NIL is equal to NIL and not equal to any other data type.

SET EXACT

For character comparisons, the "equal" operator takes the SET EXACT setting into account. With SET EXACT OFF, characters are compared up to the length of the right operand. With SET EXACT ON, characters are compared up to the length of the left operand and trailing blank spaces are ignored.

Info

See also: $, <, <=, <> != #, ==, >, >=, SET EXACT
Category: Comparison operators, Operators
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example shows results of the = operator.

   PROCEDURE Main

      SET EXACT ON
      ? "ABC"   = "ABC   "              // result: .T.
      ? "ABC   " = "ABC"                // result: .T.
      ? ""      = "ABC"                 // result: .F.
      ? "ABC"   = ""                    // result: .F.
      ? "ABC"   = "ABCDE"               // result: .F.
      ? "ABCDE" = "ABC"                 // result: .F.

      SET EXACT OFF
```

```
      ? "ABC"   = "ABC  "              // result: .F.
      ? "ABC  " = "ABC"                // result: .T.
      ? ""      = "ABC"                // result: .F.
      ? "ABC"   = ""                   // result: .T.
      ? "ABC"   = "ABCDE"              // result: .F.
      ? "ABCDE" = "ABC"                // result: .T.

      ? CToD("12/28/2005") = CToD("12/31/2005")
                                       // result: .F.
      ? .T. = .F.                      // result: .F.

      ? NIL = NIL                      // result: .T.

      ? 2 = 1                          // result: .F.
   RETURN
```

= (compound assignment)

Compound assignment (binary): inline operation with assignment.

Syntax

```
<Variable> += <cString>       // inline string concatenation
<Variable> += <nNumber>       // inline addition
<Variable> -= <cString>       // inline string concatenation
<Variable> -= <nNumber>       // inline subtraction
<Variable> -= <dDate>         // inline subtraction
<Variable> *= <nNumber>       // inline multiplication
<Variable> /= <nNumber>       // inline division
<Variable> %= <nNumber>       // inline modulus
<Variable> ^= <nNumber>       // inline exponentiation
```

Arguments

`<Variable>`

<*Variable*> is the name of an arbitrary variable. It must be initialized with a value of a data type valid for the inline operation.

`<cString>`

<*cString*> is a character string used in the inline operation.

`<nNumber>`

<*nNumber*> is a numeric expression used in the inline operation.

`<dDate>`

<*dDate*> is a date value used in the inline operation.

Description

A compound operator takes the value of the left operand and performs an inline operation with the value of the right operand, before it assigns the result to the left operand. Compound operators can therefore be used in expressions like the inline assignment operator.

The following table lists the operators and their equivalents:

Compound operators

Operator	Equivalent	Inline operation
x += y	x := (x + y)	Concatenation, Addition
x -= y	x := (x - y)	Concatenation, Subtraction
x *= y	x := (x * y)	Multiplication
x /= y	x := (x / y)	Modulus
x ^= y	x := (x ^ y)	Exponentiation

Note that there is no **= compound assignment operator, but only the ^= operator.

Info

See also: %, *, **, +, ++, -, --, /, :=
Category: Assignment operators, Mathematical operators, Operators
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The  example demonstrates the results of compound operators.

   PROCEDURE Main
      LOCAL cString := "A"
      LOCAL dDate   := StoD( "20051130" )
      LOCAL nValue  := 2

      ? cString += "B "                // result: "AB "
      ? cString -= "C "                // result: "ABC  "

      ? dDate += 24                    // result: 12/24/2005

      ? (nValue ^= 3) - 6              // result: 2.00
      ? nValue                         // result: 8.00

      nValue ^= 4
      ? nValue                         // result: 4096.00

   RETURN
```

==

Exact equal operator (binary): compares two values for identity.

Syntax

```
<Expression1> == <Expression2>
```

Arguments

<Expression>

The values of *<Expression1>* and *<Expression2>* must have the same data type or can be NIL, and are compared.

Description

The "exact equal" operator compares two values of the same data type or NIL. It returns .T. (true) when the left operand has exactly the same value as the right operand, otherwise the return value is .F. (false).

The data types Character and Memo are compared only on the basis of the ASCII values of their characters. The SET EXACT setting is ignored. That means, two operands have exactly the same character values when both have the same length and contain the exact same sequence of characters (case sensitive).

The data types Date, Logic and Numeric are compared to be exactly equal when both operands have the same value.

Variables holding the complex data types Array, Code block, Hash and Object do not store a value but a reference to the value. The "exact equal" operator returns .T. (true) when both operands have the same reference to the complex data type.

Info

See also:	$, <, <=, <> != #, = (comparison), >, >=
Category:	Comparison operators, Operators
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates resultsof the exact equal operator.

   PROCEDURE Main

      ? "ABC" == "ABC"                // result: .T.
      ? "ABC" == "XYZ"                // result: .F.
      ? "ABC" == "ABC  "              // result: .F.

      aArray1 := {1, 2, 3}            // Create array reference
      aArray2 := {1, 2, 3}            // Create new array reference
      aArray3 := aArray1              // Assign array reference

      ? aArray1 == aArray2            // result: .F.
      ? aArray1 == aArray3            // result: .T.
      ? aArray2 == aArray3            // result: .F.

   RETURN
```

>

Greater than operator (binary): compares the size of two values.

Syntax

```
<Expression1> > <Expression2>
```

Arguments

`<Expression>`

The values of *<Expression1>* and *<Expression2>* must have the same data type and are compared.

Description

The "greater than" operator compares two values of the same data type. It returns .T. (true) when the left operand is greater than the right operand, otherwise the return value is .F. (false).

Only the simple data types Character, Data, Logic, Memo and Numeric can be compared. The complex data types Array, Code block, Hash, Object and the value NIL cannot be used with the "greater than" operator.

Comparison rules

Data type	Comparison
Character	The comparison is based on the ASCII value of the characters.
Date	The comparison is based on the underlying date value.
Logical	False (.F.) is smaller than true (.T.).
Memo	Same as Character data type.
Numeric	Comparison is based on the value of the numerics.

SET EXACT

For character comparisons, the greater than operator takes the SET EXACT setting into account. With SET EXACT OFF, characters are compared up to the length of the right operand. With SET EXACT ON, characters are compared up to the length of the left operand and trailing blank spaces are ignored.

Info

See also: $, <, <=, <> != #, = (comparison), ==, >=, SET EXACT
Category: Comparison operators, Operators
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates the result of the ">" operator.

   PROCEDURE Main
      ? "A" > "Z"                // result: .F.
      ? "a" > "Z"                // result: .T.

      ? CToD("12/28/2005") > CToD("12/31/2005")
                                 // result: .F.
      ? .T. > .F.                // result: .T.

      ? 2 > 1                    // result: .T.

      SET EXACT OFF
      ? "ab" > "a"               // result: .F.
```

```
    SET EXACT ON
    ? "ab" > "a"                 // result: .T.
RETURN
```

>=

Greater than or equal operator (binary): compares the size of two values.

Syntax

```
<Expression1> >= <Expression2>
```

Arguments

```
<Expression>
```

The values of *<Expression1>* and *<Expression2>* must have the same data type and are compared.

Description

The "greater than or equal" operator compares two values of the same data type. It returns .T. (true) when the left operand is greater than or equal to the right operand, otherwise the return value is .F. (false).

Only the simple data types Character, Data, Logic, Memo and Numeric can be compared. The complex data types Array, Code block, Hash, Object and the value NIL cannot be used with the "greater than or equal" operator.

Comparison rules

Data type	Comparison
Character	The comparison is based on the ASCII value of the characters.
Date	The comparison is based on the underlying date value.
Logical	False (.F.) is smaller than true (.T.).
Memo	Same as Character data type.
Numeric	Comparison is based on the value of the numerics.

SET EXACT

For character comparisons, the "greater than or equal" operator takes the SET EXACT setting into account. With SET EXACT OFF, characters are compared up to the length of the right operand. With SET EXACT ON, characters are compared up to the length of the left operand and trailing blank spaces are ignored.

Info

See also: $, <, <=, <> != #, = (comparison), ==, >, SET EXACT
Category: Comparison operators, Operators
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates the result of the ">=" operator.

   PROCEDURE Main
      ? "A" >= "Z"               // result: .F.
      ? "a" >= "Z"               // result: .T.

      ? CToD("12/28/2005") >= CToD("12/31/2005")
                                 // result: .F.
      ? .T. >= .F.               // result: .T.

      ? 2 >= 1                   // result: .T.
```

```
    SET EXACT OFF
    ? "a"  >= "a    "          // result: .F.
    ? "a"  >= "ab"             // result: .F.
    ? ""   >= "a"              // result: .F.
    ? "a"  >= ""               // result: .T.

    SET EXACT ON
    ? "a"  >= "a    "          // result: .T.
    ? "a"  >= "ab"             // result: .F.
    ? ""   >= "a"              // result: .F.
    ? "a"  >= ""               // result: .T.
 RETURN
```

>>

Right-shift operator (binary): shifts bits to the right.

Syntax

```
<nNumber1> >> <nNumber2>
```

Arguments

`<nNumber1>`

<nNumber1> is a numeric value whose bits are shifted to the right.

`<nNumber2>`

<nNumber2> is a number indicating how many places the bits are shifted to the right.

Description

The >> operator accesses individual bits in the left operand and shifts them to the right as many times as specified with the right operand. Both operands are always treated as integer values. If an operand has a decimal fraction, it is truncated prior to the operation.

A shift operation involves all bits of the left operand. When the bits are shifted one place to the right, the lowest bit is discarded and the highest bit is set to 0. A numeric value has 32 bits on a 32 bit operating system.

Shifting a nonnegative value to the right is equivalent with dividing *<nNumber1>* by 2 raised to the power of *<nNumber2>*.

Info

See also: <<, HB_BitShift()
Category: Bitwise operators, Operators, xHarbour extensions
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// This example shifts the bits of the value 52 two places to
// the right so the value becomes 13.

   PROCEDURE Main
      nValue := 52            // binary: 00110100
      ? nValue >> 2           // binary: 00001101 (value = 13)

      ? 52 / 2 ^ 2            // result: 13
   RETURN
```

@

Pass-by-reference operator (unary): passes variables by reference.

Syntax

```
@ <Variable>
```

Arguments

<Variable>

<Variable> is the name of any valid memory variable. Field variables cannot be passed by reference.

Description

When a variable is passed by reference to a subroutine, any changes made to the variable are reflected in the calling routine once the called routine has returned.

Normally, variables are passed by value. This means that the called routine receives a copy of the passed value so that the variable's value in the calling routine remains unaffected by assignment operations in the called routine.

When a variable is prefixed with the @ operator in a function call, the called function receives a reference to the variable. The variable can then be assigned a new value in the called function. The changed value is reflected in the calling routine when the called function has finished.

Passing variables by reference is usually required when a subroutine must deliver more than one return value.

Info

See also:	FUNCTION, METHOD (Declaration), PROCEDURE
Category:	Operators, Special operators
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows the difference between a function call with
// and without the @ operator:

   PROCEDURE Main()
      LOCAL cMsg := "Hello World"

      ? GetMsg( cMsg )                   // result: Ok
      ? cMsg                             // result: Hello World

      ? GetMsg( @cMsg )                  // result: Ok
      ? cMsg                             // result: Hi there
   RETURN NIL

   FUNCTION GetMsg( cString )
      cString := "Hi there"
   RETURN "Ok"
```

@()

Function-reference operator (unary): obtains a function pointer.

Syntax

```
( @<funcName>() )  -->  pFuncPointer
```

Arguments

`<funcName>`

<funcName> is the symbolic name of a function or procedure to obtain the function pointer from. Note that the function-reference operator can only be used within parentheses. If this syntactical rule is not complied with, a syntax error is reported by the compiler.

Description

The function-reference operator retrieves the pointer to a function or procedure. The resulting value is of Valtype()=="P".

A function pointer can be used with function HB_Exec() or HB_ExecFromArray() to execute the function indirectly. This type of indirect execution is similar to using *<funcName>* within a macro-expression, or embedding the call within a code block which is then passed to the Eval() function. If a function is to be called indirectly for multiple times, however, the usage of the function-pointer along with HB_Exec() or HB_ExecFromArray() has a considerable performance advantage.

Note: The function-reference operator can resolve FUNCTIONs or PROCEDUREs declared as STATIC only if the referenced *<funcName>* is declared in the same PRG file where the operator is used. Pointers to methods of objects can only be obtained using function HB_ObjMsgPtr().

Info

See also:	{|| }, ( ), FUNCTION, HB_Exec(), HB_ExecFromArray(), HB_FuncPtr(), HB_ObjMsgPtr(), METHOD (Declaration), PROCEDURE
Category:	Operators, Indirect execution, Special operators, xHarbour extensions
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows the difference of indirect execution using a
// code block and a function pointer

   PROCEDURE Main
      LOCAL pFunc  := ( @Test() )
      LOCAL bBlock := {|x,y| Test( x, y ) }

      ? Valtype( bBlock )       // result: B
      ? Valtype( pFunc )        // result: P

      EVal( bBlock,  "Hello", "World" )

      HB_Exec( pFunc, NIL, "Hello", "World" )
   RETURN


   PROCEDURE Test( ... )
      LOCAL xVal
      ? PCount()

      FOR EACH xVal IN HB_AParams()
```

```
        ?? xVal
     NEXT
  RETURN
```

HAS

Searches a character string for a matching regular expression.

Syntax

```
<cString> HAS <cRegEx>
```

Arguments

`<cString>`

This is the character string being searched.

`<cRegEx>`

This is a character string holding the search pattern as a literal regular expression. Alternatively, the return value of HB_RegExComp() can be used.

Description

The HAS operator searches the character string *<cString>* for a substring based on the regular expression *<cRegExp>*. If the string contains a substring that matches the search pattern, the result of the operator is .T. (true). When no match is found, the result is .F. (false).

If the HAS operator is applied often in a search routine with the same regular expression, a literal regular expression *<cRegEx>* can be compiled with HB_RexExComp() to speed up operations.

Info

See also:	$, AScan(), IN, LIKE, HB_RegEx(), HB_RegExComp(), HScan()
Category:	Character operators, Comparison operators, Operators, Regular expressions, xHarbour extensions
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example uses two regular expressions to detect if
// a character string contains a number or a Hex number.
// Note that one regular expression is a literal string
// while the second is compiled.

   PROCEDURE Main
      LOCAL aExpr := { "0xEA", "3.14", "aChar", DtoC( Date() ), "0x0d" }
      LOCAL cHex  := HB_RegExComp( "0[xX][A-Fa-f0-9]+" )
      LOCAL cString


      FOR EACH cString IN aExpr
         ? cString, ""

         IF cString HAS "[0-9].?[0-9]*"
            IF cString HAS cHex
               ?? "contains a Hex number"
            ELSE
               ?? "contains a number"
            ENDIF

         ELSEIF cString HAS cHex
            ?? "contains a Hex number"

         ELSE
```

```
        ?? "contains no number"
      ENDIF
   NEXT

   ** Output:
   // 0xEA contains a Hex number
   // 3.14 contains a number
   // aChar contains no number
   // 09/21/06 contains a number
   // 0x0d contains a Hex number

RETURN
```

IN

Searches a value in another value.

Syntax

```
<cSubString> IN <cString>
<xValue>     IN <aArray>
<xKey>       IN <hHash>
```

Arguments

`<cSubString>`

<cSubString> is a character or other value that is searched for in *<cString>*.

`<aArray>`

<aArray> is an array with random values.

`<xKey>`

<xKey> is a value to search in a hash.

`<hHash>`

<hHash> is a hash value whose keys are searched for *<xKey>*.

Description

The IN operator searches the left operand in the right operand and returns .T. (true) if the value of the left operand is contained in the value of the right operand, otherwise .F. (false) is returned.

Substring search

When both operands are character strings, the IN operator performs a case-sensitive substring search and returns .T. (true) if *<cSubString>* is found in *<cString>*, otherwise .F. (false). This is the same search as with the $ operator, but IN is much faster.

Array search

If the right operand is an array, the IN operator scans the elements of *<aArray>* in the first dimension and returns .T. (true) if an element contains the value of the left operand. *<xValue>* may be of any data type.

Hash key search

If the right operand is a hash, the IN operator scans the keys of *<hHash>* (which is similar to the first column of a two column array) and returns .T. (true) if the hash has the key *<xKey>*. *<xKey>* may be of any data type.

Info

See also:	$, AScan(), HAS, LIKE, HScan()
Category:	Character operators, Comparison operators, Operators, xHarbour extensions
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates results of the IN operator with values
// of different data types.

   PROCEDURE Main
```

```
   LOCAL cString := "xHarbour"
   LOCAL hHash   := { "A" => Date(), 1 => "B" }
   LOCAL aSub    := { 1, 2 }
   LOCAL aArray  := { 1, Date(), aSub }

   ? "A"   IN cString               // result: .F.
   ? "arb" IN cString               // result: .T.

   ?  "A"  IN hHash                 // result: .T.
   ?  "B"  IN hHash                 // result: .F.
   ?   1   IN hHash                 // result: .T.

   ? "A"    IN aArray               // result: .F.
   ? Date() IN aArray               // result: .T.
   ? aSub   IN aArray               // result: .T.
RETURN
```

LIKE

Compares a character string with a regular expression.

Syntax

```
<cString> LIKE <cRegEx>
```

Arguments

`<cString>`

This is the character string being compared.

`<cRegEx>`

This is a character string holding the comparison rule as a literal regular expression. Alternatively, the return value of HB_RegExComp() can be used.

Description

The LIKE operator compares the character string *<cString>* with the regular expression *<cRegExp>*. If the string matches the regular expression, the result of the operator is .T. (true). When no match is given, the result is .F. (false).

If the LIKE operator is applied often in a search routine with the same regular expression, a literal regular expression *<cRegEx>* can be compiled with HB_RexExComp() to speed up operations.

Info

See also:	$, AScan(), IN, HAS, HB_RegEx(), HB_RegExComp(), HScan()
Category:	Character operators, Comparison operators, Operators, Regular expressions, xHarbour extensions
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example uses a regular expression to detect if a character
// string is a valid eMail address.

   PROCEDURE Main
      LOCAL cRegEx := "^[a-zA-Z0-9._%-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}$"
      LOCAL aString
      LOCAL cEMail

      aString := { "info@xHarbour.com"       , ;
                   "info_xHarbour.com"       , ;
                   "info@xHarbour,com"       , ;
                   "Mail to info@xHarbour.com" }

      FOR EACH cEMail IN aString
         IF cEMail LIKE cRegEx
            ? cEMail, "is valid"
         ELSE
            ? cEMail, "is not valid"
         ENDIF
      NEXT

      ** Output:
      // info@xHarbour.com is valid
      // info_xHarbour.com is not valid
      // info@xHarbour,com is not valid
```

```
   // Mail to info@xHarbour.com is not valid
RETURN
```

[ ] (array)

Array element operator (unary): retrieves array elements.

Syntax

```
<aArray>[<nSubScript,...>]
<aArray>[<nSubScript1>][nSubScript2][<...>]
```

Arguments

`<aArray>`

<aArray> is an expression which returns a reference to an array. This can be a memory variable, an instance variable or a function call that returns an array.

`<nSubScript>`

<nSubScript> is a numeric expression which indicates the ordinal position of the array element to retrieve. One subscript must be passed for each dimension of the array.

Description

The array element operator retrieves a value stored in an array element. The ordinal position of the desired element must be specified as a numeric value. Elements of multi-dimensional arrays are specified using one subscript value per array dimension.

Note: if *<nSubScript>* is a negative value, the operator retrieves the element Abs(*<nSubScript>*) from the end of the array.

Info

See also:	[ ] (string), { }, Array()
Category:	Operators, Special operators
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows some variations of the use of the subscript operator.

   PROCEDURE Main
      LOCAL aArray := { "One", "Two", "Three" }
                                  // one dimensional array
      ? aArray[1]                 // result: One
      ? aArray[2]                 // result: Two
      ? aArray[3]                 // result: Three

      ? aArray[-1]                // result: Three
      ? aArray[-3]                // result: One

      USE Customer                // two dimensional array
                                  // combined with function call
      ? DbStruct()[1,1]           // result: FIRSTNAME
   RETURN
```

[ ] (string)

Character operator (unary): retrieves a character from a string.

Syntax

```
<cString>[<nPos>]
```

Arguments

<cString>

<cString> is a character string containing at least one character.

<nPos>

<nPos> is a numeric expression which indicates the ordinal position of the character to retrieve.

Description

The character retrieves a single character at position *<nPos>* from the string *<cString>*. The value for *<nPos>* mus be in the range of 1 to Len(*<cString>*).

Note: if *<nPos>* is a negative value, the operator retrieves the character Abs(*<nPos>*) from the end of the the string.

Info

See also:	[ ] (array), SubStr()
Category:	Character operators, Operators, Special operators, xHarbour extensions
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example shows how to access and change characters in a string.
// Two functiosn are programmed which convert a numeric value to/from
// a string containing only "1" and "0".

   PROCEDURE Main
      LOCAL cStr

      cStr := "xHarbour"
      ? cStr[5]           // result: b

      cStr[5] := "B"
      ? cStr              // result: xHarBour

      cStr[-1] := "B"
      ? cStr              // result: xHarBouB

      ? cStr := Num2Bin( 52 )
                          // result: 00000000000000000000000000110100
      ? Bin2Num( cStr )   // result: 52.00
   RETURN

   FUNCTION Num2Bin( n )
      LOCAL i, cBin := Padr("", 32, "0" )

      FOR i:=1 TO 32
         IF IsBit( n, i )
            cBin[33-i] := "1"
         ENDIF
```

```
   NEXT
RETURN cBin

FUNCTION Bin2Num( cBin )
   LOCAL n := 0
   LOCAL i, imax := Len(cBin)
   FOR i:=1 TO imax
      IF cBin[imax+1-i]=="1"
         n += 2^(i-1)
      ENDIF
   NEXT
RETURN n
```

^^

Bitwise XOR operator (binary): performs a logical XOR operation.

Syntax

```
<cString> ^^ <nMask>|<cMask> --> cCharacter

<nNumber> ^^ <nMask>|<cMask> --> nNumeric
```

Arguments

`<cString>`

A character expression of which all bits of each character are processed in a logical XOR operation.

`<nNumber>`

A numeric value all bits of which are processed. Numbers are always treated as integer values. If a number has a decimal fraction, it is truncated.

`<nMask>|<cMask>`

The right operand of the bitwise XOR operator can be specified as a character or a numeric value.

Description

The bitwise XOR operator performs a logical XOR operation with the individual bits of both operands. The left operand is the value to process while the right operand provides the bit mask for the operation. The return value of the &-operator has the same data type as the left operand.

Bits at identical positions in both operands are compared. The bit at the same position is set in the return value, when both operands have the bit set differently at the same position. If both operands have the same bit set or not set, i.e. when both bits are the same, the corresponding bit in the return value is not set.

A special feature of a logical XOR operation is that the left operand can be recovered from the result value when the result value is XORed with the right operand in a second operation.

The bit mask to apply to the left operand can be specified as a numeric or as a character value. Depending on the left operand, bitwise XOR operations are performed according to the following rules:

cString ^^ cMask

When both operands are character values, the bits of individual characters of both operands are compared. If the right operand has less characters, it is repeatedly applied until all characters of the left operand are processed. The return value has the same number of characters as the left operand.

cString ^^ nMask

When the left operand is a character value and the right operand is numeric, the bits set in the numeric value are compared with the bits of each character in the left operand.

nNumber ^^ cMask

When the left operand is numeric and the right operand is a character value, the bits set in the numeric value are compared consecutively with the bits of each character in the right operand.

nNumber ^^ nMask

When both operands are numeric values, the bits of both values are compared.

Note: Numeric operands are always treated as integer values. If a number has a decimal fraction, it is ignored.

Info

See also: & (bitwise AND), | (bitwise OR), .AND., .NOT., .OR., HB_BitXOr()
Category: Bitwise operators, Operators, xHarbour extensions
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// This example encrypts and decrypts the value 'Secure sentence.'
// with a key 'Secret password'.

   PROCEDURE MAIN()
      LOCAL cEncrypted := "Secure sentence." ^^ "Secret password"
      LOCAL cDecrypted := cEncrypted ^^ "Secret password"

      ? cEncrypted
      ? cDecrypted
   RETURN NIL
```

{ }

Literal array.

Syntax

```
{[<expressions,...>]}
```

Arguments

```
<expressions,...>
```

<expressions,...> is a comma separated list of expressions that may yield any data type, including arrays. When an expression evaluates to an array, or when another literal array is specified, a nested, or multi-dimensional array is created.

Description

This operator is used as a delimiter for creating references to literal arrays. Expressions listed within the curly braces are evaluated and their return value is stored in the elements of the newly created array.

Note: When a variable is declared as STATIC, the expressions may only be constant values that can be resolved at compile time.

Info

See also: [ ] (array), {=>}, Array(), Hash()
Category: Operators, Special operators
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example creates arrays of different dimensions.

   PROCEDURE Main
      LOCAL aAge := {12, 15, 11, 9, 11, 14} // one dimensional
      LOCAL aUsers := { {"John", 31}, ;     // two dimensional
                        {"Marc", 28}, ;
                        {"Bill", 33}  }

      LOCAL aVal := { Date(), ;             // three dimensional
                     { 1, 2, {.T., "Ok"} }, ;
                    "Third Element" }

      ? aAge[3]                        // result: 11
      ? aUsers[2][1]                   // result: Marc
      ? aVal[2,3,2]                    // result: Ok
      ? aVal[3]                        // result: Third Element
   RETURN
```

{=>}

Literal hash.

Syntax

```
{[<key1>] => [<value1>][,<keyN> => <valueN>]}
```

Arguments

`<key1> .. <keyN>`

<key> is one or more key values to initialize the Hash with. Key values may be of data type Character, Date or Numeric. Other data types are not allowed for *<key>*.

`<value1> .. <valueN>`

<value> is one or more values of any data type associated with *<key>* when the Hash is initialized. Multiple key/value pairs are separated with commas.

Description

An empty literal Hash is created with the characters {=>}. The Hash can be initialized with *key => value* pairs upon creation.

A Hash is similar to a two column array where the keys are stored in the left and the associated values in the right column. For this reason, a Hash is often referred to as "associative array". A Hash, however, is a genuine data type in xHarbour and allows for different operations than with the Array data type. The most significant difference between Hash and Array is that while values stored in an array are retrieved using a numeric ordinal position of an array element, a Hash value is retrieved by its associated key, and not by its ordinal position.

For this reason, data types for keys must be orderable. This restricts keys to the data types Character, Date and Numeric. A key, however, can be associated with a value of any data type, including Hash.

The most common data type for keys is Character. This makes a Hash to a kind of "lightweight object" that has only data and no methods. Since the key strings are used to retrieve the associated values, a key string can be understood like a message sent to an object.

The "dual" nature of the Hash data type, being partly similar to arrays and partly to objects, finds its expression also in the syntactical notation that can be used for retrieving a value from a Hash. Both operators are supported, the array element operator [ ] and the : send message operator. The latter, however, requires keys to be of data type Character.

Important: when using the : send message operator to create or retrieve a value from a Hash, the message is case-sensitive unless HSetCaseMatch() is set to .F. (false) for the Hash.

Info

See also:	{ }, Array(), CLASS, Hash()
Category:	Operators, Special operators, xHarbour extensions
LIB:	xhb.lib
DLL:	xhbdll.dll

Examples

```
// The example demonstrates how to create and populate hash values.

   PROCEDURE Main                       // literal hash
      LOCAL hHash := { "COM1" => 1, "COM2" => 2 }

      // object notation
      ? hHash:COM1                      // result: 1
```

```
      // array notation
      ? hHash["COM2"]                 // result: 2

      // adding a new key/value pair using object notation
      hHash:COM3 := 3

      ? hHash:COM3                    // result: 3

      ? hHash:COM4                    // runtime error
   RETURN

// The example demonstrates comparison operators with hashes

   PROCEDURE Main
      LOCAL hHash1 := { "A" => 1, "B" => 2, "C" => 3, "D" => 4 }
      LOCAL hHash2 := { "B" => 5, "C" => 6, "E" => 7, "F" => 8 }
      LOCAL hHash3 := hHash1

      ? hHash1 == hHash2      // result: .F.
      ? hHash1 <> hHash2      // result: .T.

      ? hHash1 == hHash3      // result: .T.
      ? hHash1 <> hHash3      // result: .F.
   RETURN

// The example demonstrates set-oriented operations with hashes
// using Plus and Minus operators

   PROCEDURE Main
      LOCAL hHash1 := { "A" => 1, "B" => 2, "C" => 3, "D" => 4 }
      LOCAL hHash2 := { "B" => 5, "C" => 6, "E" => 7, "F" => 8 }
      LOCAL hHash3

      hHash3 := hHash1 + hHash2
      ? ValToPrg( hHash3 )
      // { "A" => 1, "B" => 5, "C" => 6, "D" => 4, "E" => 7, "F" => 8 }

      hHash3 := hHash1 - hHash2
      ? ValToPrg( hHash3 )
      // { "A" => 1, "D" => 4 }
   RETURN
```

{^ }

Literal DateTime value.

Syntax

```
{^ [<YYYY/MM/DD>] [<hh:mm:ss[.ccc>]] [AM|PM] }
```

Arguments

<YYYY/MM/DD>

The Date part of a DateTime value must be coded with digits in the sequence Year/Month/Day where the slash must be used as delimiting character.

<hh:mm:ss.ccc>

The Time part must be coded in the sequence Hour:Minute:Second where the colon is used as delimiter. Optionally, the seconds value can be followed by a period and up to three digits (.ccc) indicating fractions of a second in milliseconds. The Time part follows the Date part and must be separated with a blank space.

AM|PM

When the Time part is coded using a 12 hour clock, the suffix AM or PM can be added. The suffix must be separated from the Time part with a blank space.

Description

The DateTime operator begins with an open curly brace, followed by a caret and ends with a closing curly brace. It combines a Date value with a Time. DateTime values can be coded as literal values, consisting either of the Date part, the Time part, or both of them. However, if the Time part is omitted, it defaults to 00:00h. When the Date part is omitted, the date defaults to December 30th, 1899 (which is the start date for COM/OLE defined by Microsoft). The AM or PM suffix allows for coding a literal DateTime value using a 12 or 24 hour clock.

In contrast to regular Date value, DateTime values are independent of the SET DATE or SET EPOCH settings. A literal DateTime value must always be coded in the same way and does not follow country specific date formats.

DateTime values can be stored in a database when the corresponding database field is created with "T" as field type. The length of a "T" field is the same as for a regular Date field: 8 bytes.

Besides these differences, DateTime values can be used in the same way as regular Date values. When a value of Valtype()=="D" must be passed as parameter, it can be either a Date or a DateTime value.

Info

See also:	CtoD(), CtoT(), DateTime(), HB_IsDateTime(), TtoC(), TtoS()
Category:	Operators, Special operators, xHarbour extensions
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example outlines basic operations with DateTime values and
// lists their results.

   PROCEDURE Main
      LOCAL d1, d2, nDiff

      SET CENTURY ON
      SET TIME FORMAT TO "hh:mm:ss.ccc"
```

```
   ? DateTime()                       // result: [today] [systemtime]
   ? {^ 2007/04/26}                   // result: 04/26/2007
   ? {^ 05:30:12.345}                 // result: 12/30/1999 05:30:12.345
   ? {^ 05:30:12.345 PM}              // result: 12/30/1999 17:30:12.345

   ** Empty value
   ? d1 := {^ 0/0/0 }                 // result:   /  /
   ? Empty( d )                       // result: .T.

   ** difference between DateTime and Date
   ? d1 := {^ 2007/04/26 18:30:00 }  // result: 04/26/2007 18:30:00.000
   ? d2 := StoD("20070426")          // result: 04/26/2007
   ? nDiff := d1-d2, "days"          // result: 0.77 days
   ? TString( nDiff*86400 )          // result: 18:30:00

   ** Adding 2 days to DateTime
   ? d1 + 2                          // result: 04/28/2007 18:30:00.000

   ** Adding 2 hours to DateTime
   ? d1 + 2/24                       // result: 04/26/2007 20:30:00.000

   ** Adding 2 minutes to DateTime
   ? d1 + 2/(24*60)                  // result: 04/26/2007 18:32:00.000

   ** Adding 2 seconds to DateTime
   ? d1 + 2/(24*3600)                // result: 04/26/2007 18:30:02.000

RETURN
```

{| | }

Literal code block.

Syntax

```
{|[<params,...>]| <expressions,...> }
```

Arguments

`<params,...>`

This is an optional comma separated list of parameter names declared for use inside the code block. Code block parameters are only visible within the code block and must be placed between the || delimiters.

`<expressions,...>`

<expressions,...> is a list of comma separated expressions to execute in the code block. The return value of the code block is the return value of the last expression in the list.

Description

Literal code blocks are created in program code using the {|| } characters. A code block is an extremely powerful instrument since it allows a programmer to separate the definition of program code from its execution time. A code block is a piece of executable code that can be assigned to a variable. This makes a code block similar to a macro expression but in contrast to a macro, a code block is resolved at compile time. This makes code blocks considerably faster than macro expressions that are compiled at runtime.

The program code, embedded within a code block, is executed by passing the code block to the Eval() function. Arguments passed to this function are passed on to the code block and are received by the code block parameters. The expressions in the code block are then executed from left to right. The return value of the last expression is finally returned from the code block.

When LOCAL variables exist at the time of code block creation and are included in the expression list, they are embedded in the code block. This has a far reaching consequence: normally, LOCAL variables cease to exist when the function returns in which the LOCALs are declared. If, however, the function returns a code block with embedded LOCAL variables, they continue to exist inside the code block and for the lifetime of it. This way, LOCAL variables can become detached from the declaring function context.

Info

See also:	<|| >, AEval(), AScan(), ASort(), DbEval(), Eval(), HEval(), LOCAL
Category:	Indirect execution, Operators, Special operators
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates two basic techniques for using code blocks.
// The first code block receives parameters from outside (the Eval()
// function), while the second code block uses embedded LOCAL variables
// that are detached from function MyDispbox().

   PROCEDURE Main
      LOCAL nTop   :=  5
      LOCAL nLeft  := 10
      LOCAL nBot   := 20
      LOCAL nRight := 70
      LOCAL bBlock := {|t,l,b,r| DispBox(t,l,b,r) }
```

```
   CLS
   // draw a box at screen
   Eval( bBlock, nTop, nLeft, nBot, nRight )

   @ MaxRow()-1, 0
   WAIT "press key to clear screen"

   CLS
   @ MaxRow()-1, 0
   WAIT "press key to redraw box"

   bBlock := MyDispBox( nTop, nLeft, nBot, nRight )

   @ MaxRow()-1, 0
   WAIT "press key to restore screen"

   Eval( bBlock )

   @ MaxRow()-1, 0
   WAIT "press key to end           "
RETURN

// this function creates a code block with detached LOCALs
FUNCTION MyDispBox( nT, nL, nB, nR )
   LOCAL cScreen := SaveScreen( nT, nL, nB, nR )

   Dispbox(  nT, nL, nB, nR )
RETURN {|| RestScreen( nT, nL, nB, nR, cScreen ) }
```

| (bitwise OR)

Bitwise OR operator (binary): performs a logical OR operation.

Syntax

```
<cString> | <nMask> --> Character
<cString> | <cMask> --> Character
<nNumber> | <nMask> --> Numeric
<nNumber> | <cMask> --> Numeric
```

Arguments

<cString>

A character expression of which all bits of each character are processed in a logical OR operation.

<nNumber>

A numeric value all bits of which are processed. Numbers are always treated as integer values. If a number has a decimal fraction, it is truncated.

<nMask>|<cMask>

The right operand of the bitwise OR operator can be specified as a character or a numeric value.

Description

The bitwise OR operator performs a logical OR operation with the individual bits of both operands. The left operand is the value to process while the right operand provides the bit mask for the operation. The return value of the | operator has the same data type as the left operand.

Bits at identical positions in both operands are compared. The bit at the same position is set in the return value, when one of both operands have the bit set at the same position. If either operand has a bit not set, the corresponding bit in the return value is also not set.

The bit mask to apply to the left operand can be specified as a numeric or as a character value. Depending on the left operand, bitwise OR operations are performed according to the following rules:

cString | cMask

When both operands are character values, the bits of individual characters of both operands are compared. If the right operand has less characters, it is repeatedly applied until all characters of the left operand are processed. The return value has the same number of characters as the left operand.

cString | nMask

When the left operand is a character value and the right operand is numeric, the bits set in the numeric value are compared with the bits of each character in the left operand.

nNumber | cMask

When the left operand is numeric and the right operand is a character value, the bits set in the numeric value are compared consecutively with the bits of each character in the right operand.

nNumber | nMask

When both operands are numeric values, the bits of both values are compared.

Note: Numeric operands are always treated as integer values. If a number has a decimal fraction, it is ignored.

Info

See also: & (bitwise AND), ^^ (bitwise XOR), .AND., .NOT., .OR., HB_BitOr()
Category: Bitwise operators, Operators, xHarbour extensions
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example sets all characters in lower case and performs a bitwise
// OR operation between 123 as character/numeric and 3.

   PROCEDURE Main()

      ? "MiXeD CaSe StRiNg" | 32          // result: mixed case string

      ? "123" | 3                         // result: 333
      // "1" binary: 00110001 | 00000011  => 00110011  "3"
      // "2" binary: 00110010 | 00000011  => 00110011  "3"
      // "3" binary: 00110011 | 00000011  => 00110011  "3"

      ? 123 | 3                           // result: 123
      // 123 binary: 01111011 | 00000011  => 01111011

   RETURN NIL
```

Preprocessor Reference

#command | #translate

User defined command or translation rule for the preprocessor.

Syntax

```
#command    <searchPattern>  =>  <resultPattern>
#translate  <searchPattern>  =>  <resultPattern>
```

Arguments

`<searchPattern>`

This is the definition of a pattern to search for in the PRG source code. The => characters are part of the preprocessor directive and must be used as separator between <*searchPattern*> and <*resultPattern*>.

`<resultPattern>`

This is the definition of a pattern to replace <*searchPattern*> with in the PRG source code.

Description

The directives #command and #translate specify translation rules for the preprocessor. Source code is modified in a compilation cycle according to these rules by the preprocessor. The modified source code is then processed by the compiler. To obtain the result of the preprocessor, the compiler switch /p or /pt must be used. This causes the translation result of the preprocessor be output to a file with the PPO extension (PPO stands for PreProcessed Output).

Both directives provide a powerful automated transformation tool, that allow for extending the xHarbour language with user-defined commands, pseudo functions, and for source code patterns to be simplified by the programmer and later expanded to verbose expressions with the aid of the preprocessor.

While #command is used to define a complete statement, #translate instructs the preprocessor to find even portions of statements. Both directives identify keywords only up to the first four characters and are not case-sensitive. To match keywords to their entire length, the directives #xcommand or #xtranslate must be used (note the 'x' prefix).

Directives are usually programmed in #include files having the CH extension. The rules they define for translating source code are applied according to their preference and sequence in which they appear in an include file. The directive with highest preference is #define, followed by #translate/#xtranslate and #command/#xcommand. That means, #define directives are processed first until no more #define matches are found. In a next step, #translate/#xtranslate rules are applied and finally #command/#xcommand is processed. This process is recurring, so that each transformation forces a re-scan of the whole new line, starting again with #define directives.

The sequence for processing directives is determined by their order of definition in the program code. The most recently specified directive overrides/hides any prior directive that would have otherwise matched the same input.

Because directives defined at the end of a file are processed prior to those at the beginning of a file, when a set of rules depend one on the output of the other, the first rule to expand, should be listed last.

This is important to note when having to use multiple rules in order to implement user-defined commands that have various syntactical forms or use mutual exclusive options, thus requiring multiple rules. The most general form of a user-defined command must appear towards the top of an #include file, while specialized forms must be placed below (towards the end of a file). This makes sure that spezialized forms of a user-defined command are given the opportunity to match first, before the more general rules are evaluated.

When a directive results in a pattern of program code for which another directive is defined, the translation procedure is repeated until all translation directives are resolved by the preprocessor. This

allows for defining complex translation rules where a particualer sequence of program code is translated in multiple intermediate steps using different directives.

Directives may be defined directly within a given source files, or within any of the files included into the compiled source file, by means of the #include directive.

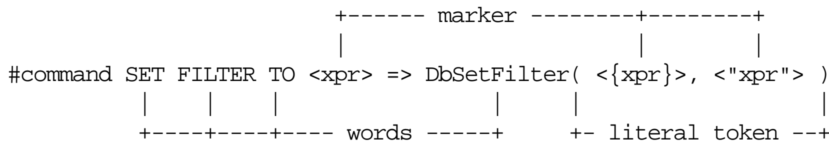
The xHarbour compiler includes a set of pre-loaded standard rules, compatible with the CA-Clipper 'std.ch' file. You may choose to exclude these default rules, by using the compiler -u command line switch. You may also "inject" a set of rules from any command header file, into the compliation of any source, by means of the -u+*<filename>* command line switch. This will load such command header file as if it was directly included in the compiled source, by means of the #include directive.

When the compilation of one PRG file is complete, all translation rules valid for this file are discarded and the preprocessor operates with a new set of translation rules for the next PRG file.

Translation rules

The *<searchPattern>* and *<resultPattern>* portions of a translation rule are syntactically programmed using literal tokens, words and markers. A directive must begin with a word or a literal token, it cannot begin with a marker.

Markers are a place holders in a translation rule for variable portions of patterns in the program code to translate. They can be compared to memory variables that hold character strings during the translation process. Markers have symbolic names and are always enclosed in angled brackets.



This translation rule instructs the preprocessor to search for the words SET FILTER TO and match the following expression with the marker *<xpr>*. This marker has the symbolic name *xpr* and is included in the resulting code in two syntactically different ways. The words SET FILTER TO are completely removed and replaced with *DbSetFilter()*.

The symbolic name for markers must follow the same rules as for memory variables, i.e. it must begin with an alphabetic character or an underscore, followed by alphabetic characters, digits or underscores. Unlike symbolic variable names, **symbolic names for markers are case sensitive**.

Markers on the left side of the => characters are referred to as Match markers, while those on the right side are called Result markers. There is a set of Match markers to distinguish syntactically different patterns of program code to search for, and a set of Result markers that define syntactically different forms of how the matched input string should appear in the output.

The large number of possible combinations of different Match and Result markers makes the preprocesser a very flexible tool for translating almost any kind of PRG source code.

Match marker

Match markers are used in the *<searchPattern>* part of a directive and instruct the preprocessor about the pattern to match in the PRG code. The following table lists the available Match markers:

Match Markers

Syntax	Name
<marker>	Regular match marker
<marker,...>	List match marker
<marker:word list,...>	Restricted match marker
*<*marker*>*	Wild match marker
<(marker)>	Extended Expression match marker
<!marker!>	Single token match marker

Regular match marker: This marker is the most commonly used one. It matches any valid expression in the program code.

List match marker: Expressions in program code that are comma separated lists are matched with the List match marker. It is used when a command has a variable number of arguments separated with commas.

Restricted match marker: This marker limits the number of words or literals valid for a command option. One or more words valid for an option are included following a colon after the symbolic name of the marker. Multiple words must be separated with commas. The words are matched case-insensitive.

Wild match marker: The Wild match marker matches everything from the current position to the end of a statement. This includes input text that may not be a valid expression.

Extended expression match marker: This marker is used for command arguments that may or may not be enclosed in parentheses.

Single token match marker: This marker identifies single tokens appearing adjacent in an expression. It is required to match words and literal tokens when they are not separated with blank spaces.

Optional clauses: It is possible to define optional clauses for *<searchPattern>* to match program code with optional arguments or keywords that may be absent. Optional clauses for *<searchPattern>* must be enclosed in square brackets [] and can be nested. They are matched in program code irrespective of the order they appear in a search rule and a user defined command. That means, if the search rule defines optional clauses in the order A,B,C and they are used in PRG code in C,A,B order, there will still be a match for all three optional clauses.

Adjacent optional clauses in a search rule must consist either of a keyword or a keyword/literal token plus Match marker. They cannot contain only Match markers without keyword or literal token.

Escape character: When the characters [and < are part of the PRG code to match, they must be escaped in the search rule by prepending a backslash \.

Line continuation: When the definition of a search pattern spans across multiple lines, each line must end with a semicolon. The ; serves as line continuation character.

Result marker

A result marker receives the input text matched by the Match marker having the same symbolic name. The input text is written to the output text at the place where the Result marker appears in the *<resultPattern>* portion of a translation directive. Different Result markers allow for modifying the syntactical notation written to the output. The following table lists the available Result markers:

Result Markers

Syntax	Name
<marker>	Regular result marker
#<marker>	Dumb stringify result marker
<"marker">	Normal stringify result marker
<(marker)>	Smart stringify result marker
<{marker}>	Blockify result marker
<.marker.>	Logify result marker

Regular result marker: This marker is the most general one. It receives matched input and writes it unmodified to the output. If there is no match in the search, nothing is written to the output.

Dumb stringify result marker: This marker encloses matched input in quotes and writes a literal character string to the output. If there is no match in the search, a null string ("") is written to the output. If the input is matched with the List match marker, the entire list is enclosed in quotes, not individual list elements.

Normal stringify result marker: This marker encloses matched input in quotes and writes a literal character string to the output. If there is no match in the search, nothing is written to the output. If the input is matched with the List match marker, each individual list element is written as a quoted character string to the output.

Smart stringify result marker: This marker encloses matched input only in quotes when it is not enclosd in parentheses. If the input is enclosed in parentheses, it is written unmodified to the output. When the matched input is a macro expression, it is optimized by the Smart stringify result marker.

Blockify result marker: This marker embeds matched input in code block literals {|| }. The resulting code block has no parameters but contains only the matched expression. If the input is matched with the List match marker, each element of the list is output as an individual code block.

Logify result marker: The marker writes literal logical values to the output. If there is a match, the output is .T. (true), otherwise .F. (false).

Optional clauses: Optional clauses for <*resultPattern*> must be enclosed in square brackets []. Unlike optional clauses for <*searchPattern*> the ones for <*resultPattern*> cannot be nested. If an optional clause is matched multiple times in <*searchPattern*> it is output as many times in <*resultPattern*>.

Escape character: When the characters [and < are part of the PRG code to output, they must be escaped in the result rule by prepending a backslash \.

Line continuation: When the definition of a result pattern spans across multiple lines, each line must end with a semicolon. The ; serves as line continuation character. If the output should be created as multiple statements, use two ;; semicolons to separate each statement.

Examples for preprocessor directives

The examples below demonstrate various combinations of Match and Result markers as they are used in the definition of preprocessor directives. Note that each example has three sections: the directive (CH file), the match pattern (PRG file) and the resulting output (PPO file).

Info

See also: #define, #include, #uncommand | #untranslate, #xcommand | #xtranslate
Category: Preprocessor directives

Examples

Regular match marker => Regular result marker

```
// CH file
   #translate IsNegative(<num>)  =>  (<num> \< 0)

// PRG file
   IF IsNegative( nValue )
      <statements>
   ENDIF

// PPO file
   IF (nValue < 0)
      <statements>
   ENDIF
```

List match marker => Regular result marker

```
// CH file
   #command ? [<list,...>]  =>  QOut( <list> )

// PRG file
   ? "Today is", Date(), "!"
```

```
// PPO file
   QOut("Today is",Date(),"!" )
```

Restricted match marker => Smart stringify result marker

```
// CH file
   #command SET DELETED <x:ON,OFF,&>  =>  Set( _SET_DELETED, <(x)> )

// PRG file
   cVar := "ON"

   SET DELETED OFF
   SET DELETED &cVar

// PPO file
   cVar := "ON"

   Set(11,"OFF" )
   Set(11,cVar )
```

Restricted match marker => Logify result marker

```
// CH file
   #command SET ALTERNATE TO <(file)> [<add: ADDITIVE>] => ;
            Set( _SET_ALTFILE, <(file)>, <.add.> )

// PRG file
   SET ALTERNATE TO test.log
   SET ALTERNATE TO test.log ADDITIVE

// PPO file
   Set(19,"test.log",.F. )
   Set(19,"test.log",.T. )
```

List match marker => Smart stringify result marker

```
// CH file
   #command COPY STRUCTURE [TO <(file)>] [FIELDS <fields,...>] => ;
            __dbCopyStruct( <(file)>, { <(fields)> } )

// PRG file
   COPY STRUCTURE TO Temp.dbf FIELDS Lastname, Firstname

   cDbFile  := "Customer.dbf"
   cFName1  := "Firstname"
   cFName2  := "Lastname"

   COPY STRUCTURE TO (cDbFile) FIELDS (cFname1), (cFname2)

// PPO file
   __dbCopyStruct("Temp.dbf",{ "Lastname","Firstname" } )

   cDbFile := "Customer.dbf"
   cFName1 := "Firstname"
   cFName2 := "Lastname"

   __dbCopyStruct((cDbFile),{ (cFname1),(cFname2) } )
```

Regular match marker => Blockify result marker

```
// CH file
   #command SET FILTER TO <xpr> => DbSetFilter( <{xpr}>, <"xpr"> )
```

```
// PRG file
   SET FILTER TO FIELD->City = "New York"

// PPO file
   DbSetFilter({||FIELD->City = "New York"},'FIELD->City = "New York"' )
```

Wild match marker => Smart stringify result marker

```
// CH file
   #command SET PATH TO <*path*>  => Set( _SET_PATH, <(path)> )

// PRG file
   cTemp := "C:\temp"

   SET PATH TO c:\xhb\source\samples
   SET PATH TO (cTemp)

// PPO file
   cTemp := "C:\temp"

   Set(6,"c:\xhb\source\samples" )
   Set(6,(cTemp) )
```

Single token match marker => Regular result marker

```
// CH file
   #translate := <!const!> <op:?> <a>,<b>  => ;
              := IIF( .NOT. Empty(<const>), <a>, <b>)

// PRG file
   lLogic := (Val(Time()) < 12)

   ? c := lLogic?"am", "pm"

// PPO file
   lLogic := (Val(Time()) < 12)

   QOut(c := IIF(.NOT. Empty(lLogic),"am","pm") )
```

Optional match clauses => Multiple statements

```
// CH file
   #command REPLACE <fld1> WITH <val1> ;
                 [, <fldN> WITH <valN> ]  => ;
                    <fld1> := <val1> ;
                 [; <fldN> := <valN>]

// PRG file
   REPLACE FIELD->LastName  WITH "Miller"

   REPLACE FIELD->LastName  WITH "Miller", ;
           FIELD->FirstName WITH "John"

// PPO file
   FIELD->LastName := "Miller"

   FIELD->LastName := "Miller" ; FIELD->FirstName := "John"
```

#define

Defines a symbolic constant or pseudo-function.

Syntax

```
#define <Constant>                  [<value>]
#define <Function>([<params,...>])  [<expr>]
```

Arguments

`<Constant>`

This is the symbolic name of the constant to define. #define constants are case sensitive.

`<value>`

This is the value all occurances of *<Constant>* are replaced with in the PRG source code by the preprocessor. If *<value>* is omitted, *<Constant>* is removed from the PRG source code.

`<Function>`

This is the symbolic name of the pseudo function to define. It is case sensitive and must be followed by parentheses.

`<params,...>`

A comma separated list of parameters can optionally be specified for the pseudo-function.

`<expr>`

This is the expression the pseudo-function is translated to by the preprocessor. All occurrances of *<Function>*(*<params,...>*) are substituted with *<expr>*.

Description

One of the most important directives is the #define directive. It is used to define a symbolic name and optionally associate it with a constant value, or to define a pseudo-function. Both, #define constants and pseudo-functions, help a programmer to maintain PRG source code better, and enhance its readability. In addition, #define constants are used for the purpose of conditional compilation in conjunction with the #if, #ifdef and #ifndef directives.

The #define directive is processed by the preprocessor, similar as #command and #translate. The major difference, however, is that #define directives are case sensitive and are coded without the => characters as separators.

Symbolic constants

A common usage for the #define directive is the definition of symbolic constants that are optionally associated with a constant value. By convention, the names of symbolic constants are coded in upper case so they are easily recognized in the PRG source code. When a symbolic constant is associated with a value, the symbolic name of the constant value can be used similar as the usage of a named memory variable. The difference is, that the value of a symbolic constant cannot change at runtime of a program. This is because symbolic constants are resolved by the preprocessor, while memory variables are run-time entities.

The following code example explains the difference between a symbolic constant and a memory variable:

```
// PRG code

#define  K_INS          22
#define  _SET_INSERT    29

nKey := Inkey(0)
```

```
IF nKey == K_INS
   Set( _SET_INSERT, .NOT. Set( _SET_INSERT ) )
ENDIF
```

This code queries a key press using the Inkey() function. When the user presses the *Ins* key, the setting for insert/overwrite mode is toggled. The code relies on two symbolic constants, K_ESC and _SET_INSERT, and both have a numeric value associated. This allows a programmer to use a symbolic name for the IF condition and the Set() function call.

The code is processed by the preprocessor. As a result, the compiler processes the following:

```
// PPO code

nKey := Inkey(0)

IF nKey == 22
   Set(29, .NOT. Set(29 ) )
ENDIF
```

A programmer could have coded the preprocessor result in the first place. It is, however, more intuitive for a programmer to use symbolic names in PRG code than to memorize numeric constants for a variety of reasons. It becomes also clear that while *nKey*, as a memory variable, can change its value later in the program, symbolic #define constants cannot: they are replaced with their associated constant value at compile time.

Pseudo-functions

A #define directive can be coded in functional syntax. Function calls that match with such an directive are translated by the preprocessor and replaced with the result pattern of the directive. The following example illustrates this:

```
// PRG code

#define InRange( val, low, high )   (val >= low .AND. high >= val)

n     := 10
nLow  := 5
nHigh := 11

IF InRange( n, nLow, nHigh )
   ? "value fits in range"
ENDIF
```

The IF condition looks syntactically like a function call accepting three memory variables as arguments. However, there is no function call due to the #define directive. The compiler processes an .AND. expression instead:

```
// PPO code

n := 10
nLow := 5
nHigh := 11

IF (n >= nLow .AND. nHigh  >= n)
   QOut("value fits in range" )
ENDIF
```

The usage of pseudo-functions is advantageous when a function call can be coded as a complex one-line expression. This can relieve a programmer from much typing and reduces the size of the resulting executable file since the overhead of a function call can be avoided. Note, however, that the name of the pseudo-function is case-sensitive, and that the function must be coded with the exact number of arguments specified in the #define directive.

Conditional compilation

A third area where #define directives play an important role is conditional compilation. Program code can be compiled in different ways depending on the presence or absence of #define constants and/or their values. This is accomplished in conjunction with the directives #if, #ifdef and #ifndef. Please refer to these topics for examples of Conditional compilation.

Info

See also: #command | #translate, #if, #ifdef, #ifndef, #undef, #xcommand | #xtranslate
Category: Preprocessor directives

#error

Raises an explicit compiler error along with a message.

Syntax

```
#error [<errMessage>]
```

Arguments

<errMessage>

If specified, this is a text to display when the #error directive is processed.

Description

The #error directive instructs the preprocessor to raise an error and display the message text specified with *<errMessage>*. The message text is output on Stdout so that it can be collected in a file.

This directive is only used in conjunction with conditional compilation.

Info

See also: #command | #translate, #define, #if, #ifdef, #ifndef, #stdout
Category: Preprocessor directives

Example

```
// The example demonstrates a scenario where the #error directive
// is useful. An error message is output when a particular #define
// constant exists.

   #define  DEMO_VERSION

   PROCEDURE Main

      #ifdef DEMO_VERSION
         #error Do not include DEMO code in production version
      #endif

      ? "Normal statements"
   ENDIF
```

#if

Compile a section of code based on a condition.

Syntax

```
  #if <condition1>
   <statements1>
[ #elif <conditionN>
   <statmentsN> ]
[ #else
   <statments>  ]
  #endif
```

Arguments

`<condition1> .. <conditionN>`

<condition> is one or more logical conditions that are resolved by the preprocessor. Conditions are formulated using #define constants, numeric or logical literal values. A condition expression may include comparison operators.

`<statements1> .. <statementsN>`

<statements> is the program code to include in the preprocessor output when a condition is true.

Description

The #if..#elif..#else..#endif directives are used for conditional compilation. They work analogous to the IF..ELSEIF..ELSE..ENDIF statements, but are resolved by the preprocessor, rather than the compiler. In contrast to the #ifdef and #ifndef directives, which test only the presence or absence of #define constants, #if allows for testing the values of such constants and comparing them with other values. In addition, multiple conditions can be formulated using #if..#elif. When the #else directive is used, it must be the last one, followed by #endif.

The statements following the first condition that resolves to True is included in the preprocessor output. All other statements are discarded. That means, even if multiple conditions are True, only one of the code statements appears in the preprocessor output. When all conditions are False, the statements following the #else directive, if present, are used.

The following rules are applied by the preprocessor to evaluate a condition:

1. A numeric literal <> 0 yields True, while exact zero yields False.
2. The logical literal .T. is True, .F. is False.
3. #define constants with no associated value yield False.
4. The literal value NIL yields False.
5. Other literal values cannot be used and result in a compile error.
6. Numeric values can be compared using the comparison operators !=, #, >, >=, =, ==, =< and <. The True/False result of the comparison is evaluated by the preprocessor.

Info

See also: #define, #ifdef, #ifndef
Category: Preprocessor directives, xHarbour extensions

Example

```
// The example demonstrates how the same source code file can be
// compiled for different program versions. Depending on the #define
// constant MAKE_VERSION, differen initilization routines are called.
```

```
#define  FULL_VERSION        3
#define  RESTRICTED_VERSION  2
#define  DEMO_VERSION        1

#define  MAKE_VERSION        RESTRICTED_VERSION

PROCEDURE Main
   #if MAKE_VERSION > RESTRICTED_VERSION
       InitFullPackage()
   #elif MAKE_VERSION > DEMO_VERSION
       InitRestrictedPackage()
   #else
       InitDemoPackage()
   #endif

   ? "Common code for all packages"
RETURN
```

#ifdef

Compiles a section of code depending on the presence of a #define constant.

Syntax

```
#ifdef <constant>
  <statements1>
[ #else
  <statements2> ]
#endif
```

Arguments

<constant>

This is the symbolic name of a #define constant.

<statements1> .. <statements2>

These are program statements to include or exclude in the preprocessor output. *<statements1>* is included when *<constant>* is defined. Otherwise, the statements following #else, if present, are included.

Description

The #ifdef..#else..#endif directives are used for conditional compilation. They work similar to the IF..ELSE..ENDIF statements, but are resolved by the preprocessor, rather than the compiler. #ifdef tests if a #define constant exists. When the constant is defined, the statements following #ifdef are included in the preprocessor output. When the constant does not exist, the statements folowing #else, if present, are used. #endif closes the conditional compilation block.

Info

See also: #define, #if, #ifndef, #undef
Category: Preprocessor directives

Example

```
// The example demonstrates how to implement an extended error
// checking in the DEBUG version of a program. When DEBUG is not
// defined, all ErrCheck() pseudo-functions are removed from
// the PRG code by the preprocessor.

   #define DEBUG

   #ifdef DEBUG
      #xtranslate ErrCheck(<errCond>,<msg>) => ;
         IF (<errCond>) ;;
            IF Alert( "Error:;"+<(msg)>,{"Continue","Quit"}) <> 1 ;;
               QUIT ;;
            ENDIF ;;
         ENDIF
   #else
      #xtranslate ErrCheck(<errCond>,<msg>) =>
   #endif

   PROCEDURE Main( cParams )
      ErrCheck( Empty(cParams), "Command line params missing" )

      ? "Error check OK"
   RETURN
```

#ifndef

Compiles a section of code depending on the absence of a #define constant.

Syntax

```
#ifndef <constant>
  <statements1>
[ #else
  <statements2> ]
#endif
```

Arguments

`<constant>`

This is the symbolic name of a #define constant.

`<statements1> .. <statements2>`

These are program statements to include or exclude in the preprocessor output. *<statements1>* is included when *<constant>* is **not** #define'd. Otherwise, the statements following #else, if present, are included.

Description

The #ifdef..#else..#endif directives are used for conditional compilation. They work similar to the IF..ELSE..ENDIF statements, but are resolved by the preprocessor, rather than the compiler. #ifndef tests if a #define constant does not exist. When the constant is undefined, the statements following #ifndef are included in the preprocessor output. When the constant does exist, the statements folowing #else, if present, are used. #endif closes the conditional compilation block.

Info

See also: #define, #if, #ifdef
Category: Preprocessor directives

Example

```
// The example demonstrates a common technique used to avoid multiple
// inclusion of an #include file. If, by accident, an include file is
// included more than once, the directives are skipped by the preprocesor
// since the #define constant MY_CH_FILE is created on first inclusion.

   #ifndef  MY_CH_FILE
      #define MY_CH_FILE

      <preprocessor directives>
   #endif
```

#include

Inserts the contents of a file into the current source code file.

Syntax

```
#include "<fileName>"
```

Arguments

`<fileName>`

This is the name of the file to insert into the file which contains the #include directive. The file name must be enclosed in double quotes.

Description

The #include directive instructs the preprocessor to insert the contents of the file *<fileName>* into the currently processed source file, passing the merged result to the compiler. *<fileName>* is the name of the include file, and can contain full path information. If no path is specified with *<fileName>*, the preprocessor searches for this file in the following locations and order:

1. the current directory.
2. the directory specified with the /i compiler switch.
3. the list of directories specified with the SET INCLUDE= environment variable.

Include files, often also referred to as Header files, have the CH extension in xHarbour. They are generally used to collect preprocessor directives. This way, directives can be maintained in one central place only. The #include directive is then used to make directives available for many PRG source code files.

Header files are only included to source code files that contain the #include directive. An exeption is the STD.CH file or the file specified with the /u compiler switch, which are included in all PRG files.

An include file may contain the #include directive so that one header file can include the next. This chained inclusion, however, is limited to a series of 15 files, each of which includes a new one.

By using the -u+*<filename>* compiler switch you may "force" a header file to be included, as if the source to be compiled has #include "*<filename>*" as it's first line.

Unless you use the -u (by itself) compiler switch, all source files are compiled with the set of default rules (comparable to CA-Clipper's std.ch file) loaded.

Info

See also: #command | #translate, #define
Category: Preprocessor directives

Example

```
// The example illustrates inclusion of a file required for
// declaring classes with xHarbour.

   #include "hbclass.ch"

   PROCEDURE Main
      LOCAL obj := MyClass():new( "Hello World" )

      obj:display()

      obj:value := "Hi there"
```

```
   obj:display()
RETURN

CLASS MyClass
   EXPORTED:
   DATA value
   METHOD new( cString ) CONSTRUCTOR
   METHOD display
ENDCLASS

METHOD new( cString ) CLASS MyClass
   ::value := cString
RETURN self

METHOD display CLASS MyClass
  ? ::value
RETURN self
```

#pragma

Controls compiler switches at compile time.

Syntax

```
#pragma <keyword> = ON|OFF
#pragma /<switch>    +|-

#pragma push /<switch>  +|-
#pragma pop  /<switch>
```

Arguments

<keyword>

This is the keyword identifying the compiler option to switch ON or OFF.

<switch>

As an alternative, the compiler switch can be specified in abbreviated notation followed by a plus sign (ON) or minus sign (OFF).

push

Pushes a new pragma on top of the stack.

pop

Removes the last pragma from the top of the stack.

Description

The #pragma directive is the only one processed by the compiler instead of the preprocessor. #pragma allows for changing compiler options while a PRG file is being compiled. It becomes possible to change a compiler switch for a single line of source code and set it back to its original value when that line is processed. The directive supports the following pragmas:

xHarbour pragmas

Keywords	Switches	Description
AUTOMEMVARS = On|Off	/A +|-	automatic memvar declaration
DEBUGINFO = On|Off	/B +|-	include debug info
ENABLEWARNINGS = On|Off	/W +|-	enable warnings
EXITSEVERITY = nLevel	/E<*nLevel*>	set exit severity
FORCEMEMVARS = On|Off	/V +|-	variables are assumed M->
LINEINFO = On|Off	/L +|-	suppress line number information
NOSTARTPROC = On|Off	/N +|-	no implicit starting procedure
PREPROCESSING = On|Off	/P +|-	generate pre-processed output (.ppo) file
WARNINGLEVEL = nLevel	/W<*nLevel*>	sets warning level
SHORTCUTTING = On|Off	/Z +|-	supress shortcutting
TRACEPRAGMAS = On|Off		trace pragma settings

Note: there is no switch notation for the TRACEPRAGMAS pragma.

Info

See also: #error, #stdout
Category: Preprocessor directives, xHarbour extensions

Examples

```
// The example demonstrates the result of pragmas using two
// identical IF statements. The first IF statement generates a
// warning and leaves the variable c2 unassigned due to shortcut
// optimization. The second IF statement generates no warning and
// assigns a value to c2 since warning and shortcut optimization
// are both switched off.

   #pragma TRACEPRAGMAS = ON
   #pragma ENABLEWARNINGS = ON

   PROCEDURE Main
      LOCAL c1 := "A"

      IF IsAlpha( c1 ) .OR. ;
         IsDigit( c2 := "2" )  // Warning: Ambiguous referece C2

         ? c1, Type( "c2" )    // result: A U
      ENDIF

      #pragma /Z-
      #pragma /W-

      IF IsAlpha( c1 ) .OR. ;
         IsDigit( c2 := "2" )  // No Warning

         ? c1, Type( "c2" )    // result: A C
      ENDIF
   RETURN

// The example is identical to the above, but uses #pragma push/pop

   PROCEDURE Main
      LOCAL c1 := "A"

      #pragma push /w3         // Switch warning level 3 on

      IF IsAlpha( c1 ) .OR. ;
         IsDigit( c2 := "2" )  // Warning: Ambiguous referece C2

         ? c1, Type( "c2" )    // result: A U
      ENDIF

      #pragma pop /w           // Switch warning level off
      #pragma push /z+         // Switch shortcut optimization off

      IF IsAlpha( c1 ) .OR. ;
         IsDigit( c2 := "2" )  // No Warning

         ? c1, Type( "c2" )    // result: A C
      ENDIF
   RETURN
```

#stdout

Sends a compiler message to StdOut.

Syntax

```
#stdout [<infoMessage>]
```

Arguments

`<infoMessage>`

This is a literal text string to send to the standard output device. If no text is specified, an empty line is output.

Description

The #stdout directive instructs the preprocessor to display the message text specified with *<infoMessage>*. The message text is output on Stdout so that it can be collected in a file.

Info

See also: #error
Category: Preprocessor directives

Example

```
// The example demonstrates the use of #stdout

   #define DEMO_VERSION

   PROCEDURE Main

      #ifdef DEMO_VERSION             // output at compile time
         #stdout Creating the Demo version of the program
      #else
         #stdout This is the FULL  version of the program
      #endif

      ? "Hi there!"                   // output at run time

   RETURN
```

#uncommand | #untranslate

Voids a previously defined #command or #translate directive.

Syntax

```
#uncommand    <searchPattern>
#untranslate  <searchPattern>
```

Arguments

<searchPattern>

This is the definition of the search pattern of a previously defined #command or #translate directive. The search pattern must be specified in the same manner as used in the definition of the #command or #translate directive to remove.

Description

The directives #uncommand and #untranslate remove a previously defined #command or #translate directive. The removed translation rule becomes unavailable and can be redefined.

#uncommand/#untranslate work similar to #xuncommand/#xuntranslate but match only the first keyword in the previously defined #command or #translate directive.

Note: #command directives built into the xHarbour compiler cannot be removed using #uncommand.

Info

See also: #command | #translate, #undef, #xuncommand | #xuntranslate
Category: Preprocessor directives, xHarbour extensions

Example

```
// see the example for #xuncommand | #xuntranslate
```

#undef

Voids a #define constant or pseudo-function.

Syntax

```
#undef <Constant>
```

Arguments

`<Constant>`

This is the symbolic name of the constant or pseudo-function to undefine. It is case sensitive.

Description

The #undef directive removes a symbolic name from the list of previously defined constants. The symbolic name becomes unavailable and can be redefined.

Info

See also: #define, #if, #ifdef, #ifndef
Category: Preprocessor directives

Example

```
// The example demonstrates a typical pattern for #ifdef, #undef and
// #define directives.

   #ifdef DEBUG_LEVEL
      #undef DEBUG_LEVEL
      #define DEBUG_LEVEL     3
   #endif
```

#xcommand | #xtranslate

User defined command or translation rule for the preprocessor.

Syntax

```
#xcommand    <searchPattern>  =>  <resultPattern>
#xtranslate  <searchPattern>  =>  <resultPattern>
```

Arguments

<searchPattern>

This is the definition of a pattern to search for in the PRG source code. The => characters are part of the preprocessor directive and must be used as separator between *<searchPattern>* and *<resultPattern>*.

<resultPattern>

This is the definition of a pattern to replace *<searchPattern>* with in the PRG source code.

Description

The #xcommand and #xtranslate directives work exactly like #command and #translate with the exception, that they match all characters of keywords. #command and #translate match keywords only up to the first four characters.

Info

See also: #command | #translate, #xuncommand | #xuntranslate
Category: Preprocessor directives

Example

```
//   See the example for #command | #translate
```

#xuncommand | #xuntranslate

Voids a previously defined #xcommand or #xtranslate directive.

Syntax

```
#xuncommand    <searchPattern>
#xuntranslate  <searchPattern>
```

Arguments

<searchPattern>

This is the definition of the search pattern of a previously defined #xcommand or #xtranslate directive. The search pattern must be specified in the same manner as used in the definition of the #xcommand or #xtranslate directive to remove.

Description

The directives #xuncommand and #xuntranslate remove a previously defined #xcommand or #xtranslate directive. The removed translation rule becomes unavailable and can be redefined.

#xuncommand/#xuntranslate match the entire translation rule previously defined with #xcommand or #xtranslate. The similar to #uncommand/#untranslate directives, in contrast, match only the first keyword in the corresponding #command or #translate directive.

Info

See also: #command | #translate, #uncommand | #untranslate
Category: Preprocessor directives, xHarbour extensions

Example

```
// The example demonstrates the effect of removing a translation rule.

   #xcommand  COMMAND  <x>          =>  A_Command( <x> )
   #xcommand  COMMAND  <x> WITH <y> =>  B_Command( <x>, <y> )

   PROCEDURE Main
                                    **  PPO file:
      COMMAND x                     //  A_Command(x )
      COMMAND x WITH y              //  B_Command(x,y )

      #xuncommand COMMAND <x>

      COMMAND x                     //  COMMAND x
      COMMAND x WITH y              //  B_Command(x,y )

   RETURN
```

Statement Reference

(struct)

Creates a new structure object

Syntax

```
<var> := (struct <StructureName>)
or
<var> := (struct <StructureName>*) <pointer>
```

Arguments

`<var>`

This is the name of a memory variable to assign the structure object to.

`<StructureName>`

This is the symbolic name of a structure declared with typedef struct.

`<pointer>`

This is the name of a memory variable holding a pointer. The pointer holds the memory address of the structure. The resulting structure object is initialized with the structure data pointed to by *<pointer>*.

Description

The *(struct)* statement provides the most convenient way of creating a structure object of a declared C structure. *(struct)* is not a statement in its strongest definition, since it is used on the right side of the assignment operator and produces a return value: a new C Structure object. The name of a declared structure *<StructureName>* must be provided with the *(struct)* statement. A corresponding structure object is then instantiated and assigned to the memory variable *<var>*. This structure object maintains an uninitialized C structure.

When the *(struct)* statement is coded with an asterisk, an additional memory variable *<pointer>* must follow the closing brace. This variable must be of Valtype()=="P", i.e. it is a pointer holding the memory address of structure data. The created structure object is then initialized with the structure data stored at the memory address of *<pointer>*. This is required in the special case when an external function called via DllCall() produces a pointer to a new structure. Data pointed to by *<pointer>* is then transferred into a new xHarbour structure object.

Note: refer to the typedef struct statement for a usage example of *(struct)*.

Info

See also:	C Structure class, DllCall(), pragma pack(), typedef struct
Category:	C Structure support, xHarbour extensions
Header:	cstruct.ch, winapi.ch, wintypes.ch
Source:	rtl\cstruct.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

ACCESS

Declares an ACCESS method of a class.

Syntax

```
ACCESS <MethodName> [ INLINE <expression> | VIRTUAL ]
```

Arguments

`<MethodName>`

This is the symbolic name of the access method to implement. It must begin with a letter or underscore followed by digits, letters or underscores. The symbolic name can contain up to 63 characters.

`INLINE <expression>`

Normally, access methods are implemented outside the class declaration, using a regular METHOD (implementation). When an access method is implemented as INLINE method, *<expression>* is executed when the access method is invoked. *<expression>* must be one line of code. The code cannot contain commands but only function and method calls.

`VIRTUAL`

An access method can be declared as VIRTUAL method. The VIRTUAL and INLINE clauses are mutually exclusive.

Description

An ACCESS method is a special method since it is invoked when an object receives a message "as if" an instance variable is queried. Sending *<MethodName>* without including parentheses to an object results in the method call. ACCESS methods "simulate" an instance variable for the calling context.

An object can have "computed" instance variables whose values are the result of a method call. An ACCESS method "hides" the method call from the calling context.

Another special method type can be declared with ASSIGN which "simulates" the write access to an instance variable via a method call.

Info

See also:	ASSIGN, CLASS, DATA, EXPORTED:, METHOD (declaration), METHOD...VIRTUAL
Category:	Class declaration, Declaration, xHarbour extensions
Header:	hbclass.ch
Source:	vm\classes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example implements a class that uses ACCESS methods
// for simulating instance variables.

   #include "hbclass.ch"

   PROCEDURE Main
      LOCAL obj := Someday():new()

      // real instance variable
      ? obj:date              // result: 08/31/06
```

```
   // "simulated" instance variables
   ? obj:yesterday        // result: Wednesday
   ? obj:today            // result: Thursday
   ? obj:tomorrow         // result: Friday

   ? obj:nextWeek         // result: 09/04/06
RETURN

CLASS Someday
   EXPORTED:
   DATA   date INIT Date()

   ACCESS today
   ACCESS tomorrow
   ACCESS yesterday
   ACCESS nextWeek
ENDCLASS

METHOD today
RETURN CDoW( ::date )

METHOD tomorrow
RETURN CDoW( ::date + 1 )

METHOD yesterday
RETURN CDoW( ::date - 1 )

METHOD nextWeek
   LOCAL nDays := -1

   // find next Monday
   DO WHILE DoW( ++nDays + ::date ) <> 2
   ENDDO
RETURN ::date + nDays
```

ANNOUNCE

Declaration of a module identifier name.

Syntax

```
ANNOUNCE <ModuleId>
```

Arguments

```
<ModuleId>
```

<ModuleId> is the symbolic name of a module.

Description

The ANNOUNCE statement declares a symbolic name for the linker which is not declared as FUNCTION, PROCEDURE or CLASS in the PRG code of an xHarbour project. This way, an entire module can be assigned a symbolic name that is later resolved by the linker.

ANNOUNCE must appear prior to any executable statement in a PRG source code file. The *<ModuleID>* identifier must be unique for the entire xHarbour project.

Info

See also: #include, EXTERNAL, REQUEST
Category: Declaration, Statements

Examples

```
// The example shows how the default replaceable database driver
// is linked into an application.

   ANNOUNCE RDDSYS

   REQUEST _DBF
   REQUEST DBFNTX
   REQUEST DBFFPT

   PROCEDURE Main

      ? "This program has the default RDD linked in"

   RETURN

// The example demonstrates how to produce an executable that does not
// require/use a replaceable database driver.

   ANNOUNCE RDDSYS

   PROCEDURE Main

      ? "This program has no RDD linked in"

   RETURN
```

ASSIGN

Declares an ASSIGN method of a class.

Syntax

```
ASSIGN <MethodName>( <param> ) INLINE <expression>
```

Arguments

<MethodName>

This is the symbolic name of the assign method to implement. It must begin with a letter or underscore followed by digits, letters or underscores. The symbolic name can contain up to 63 characters.

(<param>)

An access method receives exactly one parameter which must be declared in parentheses.

INLINE <expression>

ACCESS methods must be declared and implemented as INLINE methods. *<expression>* is executed when the access method is invoked. It must be one line of code which cannot contain commands but only function and method calls.

Description

An ASSIGN method is a special method since it is invoked when a value is assigned to an instance variable of an object. Instead of the assignment operation, the access method is invoked and the assigned value is passed as parameter to the method where it is processed. ASSIGN methods "simulate" the assignment of a value to an instance variable for the calling context.

Another special method type can be declared with ACCESS which "simulates" the read access to an instance variable via a method call.

Info

See also:	ACCESS, CLASS, DATA, EXPORTED:, METHOD (declaration)
Category:	Class declaration, Declaration, xHarbour extensions
Header:	hbclass.ch
Source:	vm\classes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example implements a class whose instances have three states
// represented as numeric -1, 0 and 1. The state can be changed by
// assigning values of different data types.

   #include "Hbclass.ch"
   #include "Error.ch"

   PROCEDURE Main
      LOCAL obj := ThreeState():new()

      ? obj:state              // result: -1
      obj:state := "ON"

      ? obj:state              // result:  1

      obj:state := "OFF"
```

```
   ? obj:state              // result:  0

   obj:state := "UNDEF"
   ? obj:state              // result: -1

   obj:state := .T.
   ? obj:state              // result:  1

   obj:state := NIL
   ? obj:state              // result: -1

   obj:state := 1
   ? obj:state              // result:  1

   obj:state := 3           // runtime error

RETURN

CLASS ThreeState
   PROTECTED:
   DATA _value  INIT  -1

   EXPORTED:
   ACCESS state
   ASSIGN state( x ) INLINE ::state( x )
ENDCLASS


METHOD state( xState ) CLASS ThreeState
   LOCAL lError := .F.

   IF PCount() == 0
      RETURN ::_value
   ENDIF

   IF Valtype( xState ) == "N"
      IF xState IN { -1, 0, 1 }
         ::_value := xState
      ELSE
         lError := .T.
      ENDIF

   ELSEIF Valtype( xState ) == "C"
      DO CASE
      CASE Upper( xState ) == "ON"
         ::_value := 1
      CASE Upper( xState ) == "OFF"
         ::_value := 0

      CASE Upper( xState ) == "UNDEF"
         ::_value := -1
      OTHERWISE
         lError := .T.
      ENDCASE

   ELSEIF Valtype( xState ) == "L"
      ::_value := IIF( xState, 1, 0 )

   ELSEIF Valtype( xState ) == "U"
      ::_value := -1

   ELSE
```

```
      lError := .T.
   ENDIF

   IF lError
      RETURN ::error( "illegal data assigned", ;
                      ::className(), "state" , ;
                      EG_ARG, {xState}         )
   ENDIF

RETURN ::_value
```

ASSOCIATE CLASS

Defines a scalar class for a native data type.

Syntax

```
ASSOCIATE CLASS <ClassName> WITH TYPE <datatype>
```

Arguments

`<ClassName>`

This is the symbolic name of the user-defined class to associate with a native data type.

`WITH TYPE <datatype>`

The following keywords are recognized for *<datatype>*: ARRAY, BLOCK, CHARACTER, DATE, HASH, LOGICAL, NIL, NUMERIC and POINTER.

Description

The ASSOCIATE CLASS statement associates a user defined scalar class with a native data type. Objects of such class are used in place of the data type *<datatype>*. A scalar class can have only CLASSDATA and METHODs but no DATA, since a scalar object itself is the data.

A scalar class must be declared and implemented following the CLASS statement. Default implementations of scalar classes can be enabled with the ENABLE TYPE CLASS. Note, however, that ASSOCIATE CLASS and ENABLE TYPE CLASS are mutually exclusive statements.

Info

See also:	ENABLE TYPE CLASS, CLASS, EXTEND CLASS...WITH METHOD
Category:	Class declaration, Declaration, xHarbour extensions
Header:	hbclass.ch
Source:	rtl\tclass.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example implements a class that extends the data type
// Date with advanced date and time formatting capabilities.
// It distinguishes upper and lower case letters for the Day,
// Month and time. A formatting template is built with the
// following characters:
//
// d    - numeric day without leading zero
// dd   - numeric day with leading zero
// ddd  - numeric day without leading zero and postfix (st, nd, rd, th)
// D    - first letter of weekday
// DD   - name of weekday abbreviated to two characters
// DDD  - name of weekday abbreviated to three characters
// DDDD - complete name of weekday
// M    - numeric month without leading zero
// MM   - numeric month with leading zero
// MMM  - month name abbreviated to three letters
// MMMM - complete month name
// yy   - year without century
// yyyy - year with century
// h    - hour without leading zero
// hh   - hour with leading zero
// m    - minute without leading zero
```

```
// mm   - minute with leading zero
// s    - second without leading zero
// ss   - second with leading zero
// \    - escape character for d, D, m, M, y, h, m, s

   #include "HbClass.ch"

   PROCEDURE Main
      LOCAL dDate

      ASSOCIATE CLASS DateFormatter WITH TYPE DATE

      dDate := Ctod( "09/01/2006" )

      ? dDate
      // result: 09/01/06

      ? dDate:shortDay()
      // result: Fri

      ? dDate:shortMonth()
      // result: Sep

      ? dDate:format( "dd MMM. yyyy" )
      // result: 01 Sep. 2006

      ? dDate:format( "dd-MM-yy hh:mm\h", Time() )
      // result: 01-09-06 18:07h

      dDate += 2

      ? dDate:format( "DDDD, MMMM ddd, at h \hour\s an\d m \minute\s" )
      // result: Sunday, September 3rd, at 18 hours and 7 minutes

      ? dDate:httpDate()
      // result: Sun, 03 Sep 2006 18:07:45
   RETURN


   CLASS DateFormatter
      EXPORTED:
      METHOD shortDay   INLINE Left( CDow(self), 3 )
      METHOD shortMonth INLINE Left( CMonth(self), 3 )
      METHOD time       INLINE Time()
      METHOD asString   INLINE DtoS(self)
      METHOD httpDate
      METHOD format
   ENDCLASS


   METHOD httpDate( cTime ) CLASS DateFormatter
   RETURN ::format( "DDD, dd MMM yyyy hh:mm:ss", cTime )


   METHOD format( cFormat, cTime ) CLASS DateFormatter
      LOCAL aToken := {}
      LOCAL cToken := ""
      LOCAL cChar

      IF cTime == NIL
         cTime := ::time()
      ENDIf
```

```
FOR EACH cChar IN cFormat
   IF .NOT. cChar IN "DdMyhms"
      IF Len( cToken ) > 0
         AAdd( aToken, cToken )
      ENDIF
      IF cChar == "\"
         cToken := cChar
      ELSE
         AAdd( aToken, cChar )
         cToken := ""
      ENDIF
   ELSE
      cToken += cChar
   ENDIF
NEXT
AAdd( aToken, cToken )

cFormat := ""
FOR EACH cToken IN aToken
   IF cToken == ""
      LOOP
   ENDIF

   SWITCH cToken[1]
   CASE "y"
      IF Len( cToken ) == 4
         cFormat += Left( ::asString(), 4 )
      ELSE
         cFormat += SubStr( ::asString(), 3, 2 )
      ENDIF
      EXIT

   CASE "M"
      SWITCH Len( cToken )
      CASE 1
         cToken := SubStr( ::asString(), 5, 2 )
         IF cToken[1] == "0"
            cFormat += cToken[2]
         ELSE
            cFormat += cToken
         ENDIF
         EXIT
      CASE 2
         cFormat += SubStr( ::asString(), 5, 2 )
         EXIT
      CASE 3
         cFormat += ::shortMonth()
         EXIT
      DEFAULT
         cFormat += CMonth(self)
      END
      EXIT

   CASE "D"
      IF Len( cToken ) <= 3
         cFormat += Left( CDoW(self), Len(cToken) )
      ELSE
         cFormat += CDoW(self)
      ENDIF
      EXIT
```

```
      CASE "d"
         SWITCH Len(cToken)
         CASE 1
            cToken := Right( ::asString(), 2 )
            IF cToken[1] == "0"
               cFormat += cToken[2]
            ELSE
               cFormat += cToken
            ENDIF
         CASE 2
            cFormat += Right( ::asString(), 2 )
            EXIT
         DEFAULT
            cToken := Right( ::asString(), 2 )
            cFormat += IIf( cToken[1] == "0", cToken[2], cToken )

            DO CASE
            CASE cToken == "11" .OR. ;
                 cToken == "12"   ; cFormat += "th"
            CASE cToken[2] == "1" ; cFormat += "st"
            CASE cToken[2] == "2" ; cFormat += "nd"
            CASE cToken[2] == "3" ; cFormat += "rd"
            OTHERWISE
               cFormat += "th"
            ENDCASE
         END
         EXIT

      CASE "h"
      CASE "m"
      CASE "s"
         SWITCH cToken[1]
         CASE "h" ; cChar := Left  (cTime,2)   ; EXIT
         CASE "m" ; cChar := SubStr(cTime,4,2) ; EXIT
         CASE "s" ; cChar := Right (cTime,2)   ; EXIT
         END

         IF Len( cToken ) == 1 .AND. cChar[1] == "0"
            cChar := cChar[2]
         ENDIF
         cFormat += cChar
         EXIT

      CASE "\"
         cFormat += SubStr( cToken, 2 )
         EXIT

      DEFAULT
         cFormat += cToken
      END
   NEXT
RETURN cFormat
```

BEGIN SEQUENCE

Declares a control structure for error handling.

Syntax

```
BEGIN SEQUENCE
 <statements_Normal>

 [ BREAK [<expression>] ]

 [ RECOVER [USING <errorVar>]
   <statements_Error>
 ]
END[SEQUENCE]
```

Arguments

`<statements_Normal>`

The statements to execute when no error occurs.

`BREAK [<expression>]`

When a BREAK statement or the Break() function is executed within a BEGIN SEQUENCE block, program flow branches to the next statement following the RECOVER statement. The value *<expression>* passed to BREAK or the Break() function is assigned to *<errorVar>*, if specified.

`RECOVER [USING <errorVar>]`

The RECOVER statement specifies the point in a program where it resumes execution if a runtime error occurs. Optionally, a variable *<errorVar>* can be given that receives the value passed to BREAK or the Break() function.

`<statements_Error>`

The statements to execute when an error occurs and a BREAK or Break() is executed.

Description

BEGIN SEQUENCE starts the declaration of an error handling control structure. It is a Clipper compatible control structure that is superseded with xHarbour's TRY...CATCH control structure.

The statements BEGIN SEQUENCE and ENDSEQUENCE determine a sequence of statements in PRG code where the programmer expects a runtime error to occur. The runtime error is gracefully handled in the routine defined with the error codeblock. The default error codeblock is defined in ERRORSYS.PRG.

If a runtime error occurs in a program between BEGIN SEQUENCE and ENDSEQUENCE, program flow is controlled by the BREAK statement or the Break() function. Both send an error recovery value to the RECOVER USING *<errorVar>* statement, where *<errorVar>* receives the value of the expression passed to Break() or BREAK. The error recovery value is usually an error object.

When the RECOVER statement is absent within BEGIN SEQUENCE and ENDSEQUENCE, and a runtime error occurs, a program resumes with the statement following ENDSEQUENCE.

Note: It is not possible to use the EXIT, LOOP or RETURN statements within the *<statements_Normal>* block of statements. Program flow cannot leave the sequence of statements declared with BEGIN SEQUENCE unless the RECOVER or ENDSEQUENCE statement is reached.

Info

See also: Break(), ErrorBlock(), ErrorNew(), RETURN, TRY...CATCH
Category: Control structures, Statements

Example

```
// The example demonstrates a typical situation where BEGIN SQEUENCE
// is used. A database file is opened without testing the existence
// of an index file. When the index file does not exist, it is created
// in the RECOVER section of the control structure.

   PROCEDURE Main
      CLS
      OpenDatabase()

      WAIT
      Browse()
      DbCloseAll()
   RETURN

   PROCEDURE OpenDatabase
      LOCAL bError := ErrorBlock( {|e| Break(e) } )

      DO WHILE .T.

         BEGIN SEQUENCE
            // Try to open the database with index
            USE Customer ALIAS Cust INDEX CustA.ntx SHARED
         RECOVER
            // Error: index missing, Build index
            USE Customer ALIAS Cust EXCLUSIVE
            INDEX ON Upper(LastName) TO CustA.ntx
            USE
            LOOP
         ENDSEQUENCE

         EXIT
      ENDDO

      ErrorBlock( bError )
   RETURN
```

CLASS

Declares the class function of a user-defined class.

Syntax

```
CLASS <ClassName> [ FROM <SuperClass,...> ] [ STATIC ]
  [EXPORTED: | PROTECTED: | HIDDEN:]
  <Member declarations>
  <...>
ENDCLASS
```

Arguments

`<ClassName>`

This is the symbolic name of the class to declare. It must begin with a letter or underscore followed by digits, letters or underscores. The symbolic name can contain up to 63 characters.

`FROM <SuperClass,...>`

The FROM option specifies one or more classes the new class is derived from. The names of these classes must be comma separated and must exist in an application. <*ClassName*> inherits all member variables and methods from its super class(es). A synonym for FROM is INHERIT (CLASS <*ClassName*> INHERIT <*SuperClass,...*>)

`STATIC`

When this option is specified, the class function is created as STATIC FUNCTION. In this case, objects of the class can only be created in the PRG module declaring the class.

`EXPORTED: | PROTECTED: | HIDDEN:`

The EXPORTED:, PROTECTED: and HIDDEN: options modify the visibility attribute for the member variables and methods of the class. The default attribute is EXPORTED:. See an explanation for the visibility attribute in the description.

Note that the colon is mandatory at the end of this option. If it is missing, the compiler reports a syntax error.

`<Member declarations>`

Member variables and methods of the class must be declared separately using the statements DATA and METHOD. These declarations define the symbolic names for accessing member variables and executing methods for objects of a class.

`ENDCLASS`

This terminates the class declaration and is mandatory. If the CLASS declaration does not end with ENDCLASS, the compiler reports a syntax error.

Description

The CLASS statement declares the Class function of a user-defined class. This function is required to instatiate a class, i.e. to create objects of a class. Objects are always created with the **:new()** method, which is part of the abstract base class HBObject() of xHarbour's OOP model.

The CLASS declaration ends with ENDCLASS. Between CLASS and ENDCLASS, the symbolic names for the members of a class must be declared with DATA (instance variables) and/or METHOD (instance methods) and/or CLASSDATA (class variables) and/or CLASSMETHOD (class methods). When the class declaration is complete, declared methods must be implemented with another METHOD declaration. It is an error when a method is declared but not implemented, and vice versa.

Inheritance

A new class can be declared on the basis of existing classes. The name of one or more classes is specified with the FROM option. In this case, the new declared class inherits all members of <*SuperClass,...*> and can access them unless their visibility attribute is set to HIDDEN:.

If no superclass is specified, HBObject() is used as implicit superclass, i.e. all methods of HBObject() exist in all user-defined classes.

Visibility of members

Access to member variables and methods of an object can be restricted by their visibility attribute, also known as *Scope*. When an object is used outside the PRG module declaring the class, the following rules apply for the visibility attribute:

EXPORTED: Access to member variables and methods with this attribute is not restricted. They can be accessed in any context, including FUNCTION and PROCEDURE.

PROTECTED: Protected member variables and methods are accessible in the class(es) declared in the PRG module and all subclasses. Subclasses are all classes that inherit directly or indirectly from the declared class. A protected member is accessible within the context of a METHOD implemented in the declaring PRG module or in the PRG module of a subclass. Protected members are not accessible within the context of FUNCTION and PROCEDURE.

HIDDEN: This attribute restricts access to member variables and methods to the methods of the class(es) declared in the PRG module. It is the most restrictive attribute. Hidden members cannot be accessed in the context of a FUNCTION or PROCEDURE, only in the context of a METHOD implemented in the declaring PRG module.

Info

See also: ACCESS, ASSIGN, DATA, EXPORTED:, HBObject(), HIDDEN:, METHOD (declaration), PROTECTED:, VAR
Category: Class declaration, Declaration, xHarbour extensions
Header: hbclass.ch
Source: vm\classes.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates accessibility of member variable using a
// base class, a subclass and a PROCEDURE. Note that the example consists of
// four files: MAIN.PRG, PRINT_PROC.PRG, BASECLASS.PRG and SUBCLASS.PRG.
// The output of the example is given at the end of the listings.

   ************************* MAIN.PRG ****************************
   PROCEDURE Main
      LOCAL obj

      obj := BaseClass():new()

      ? "=================="
      ? "Testing base class"
      ? "=================="
      Print_Proc( obj )            // prints in a PROCEDURE
      obj:print()                  // prints in a METHOD

      obj := SubClass():new()

      ? "==============="
```

```
   ? "Testing subclass"
   ? "================"
   Print_Proc( obj )            // prints in a PROCEDURE
   obj:print()                  // prints in an overloaded METHOD
   obj:super:print()            // prints in the original METHOD

RETURN

********************** PRINT_PROC.PRG *************************
// prints the contents of an object's instance variables
PROCEDURE Print_Proc( o )
   LOCAL oErr
   ?
   ? "---", ProcName() , "---"

   TRY
     ? "EXPORTED:  "
     ?? o:iExported
   CATCH oErr
     ?? oErr:description
   END

   TRY
     ? "PROTECTED: "
     ?? o:iProtected
   CATCH oErr
     ?? oErr:description
   END

   TRY
     ?  "HIDDEN:   "
     ?? o:iHidden
   CATCH oErr
     ?? oErr:description
   END
RETURN


********************** BASECLASS.PRG **************************
// prints the contents of instance variables in a method
#include "Hbclass.ch"

CLASS BaseClass
   HIDDEN:
   VAR iHidden    INIT "iHidden"

   PROTECTED:
   VAR iProtected INIT "iProtected"

   EXPORTED:
   VAR iExported  INIT "iExported"

   METHOD print
ENDCLASS


METHOD Print CLASS BaseClass
   LOCAL oErr
   ?
   ? "---", ProcName() , "---"

   TRY
```

```
      ? "EXPORTED:  "
      ?? ::iExported
    CATCH oErr
      ?? oErr:description
    END

    TRY
      ? "PROTECTED: "
      ?? ::iProtected
    CATCH oErr
      ?? oErr:description
    END

    TRY
      ?  "HIDDEN:   "
      ?? ::iHidden
    CATCH oErr
      ?? oErr:description
    END
RETURN self


**********************  SUBCLASS.PRG **************************
// prints the contents of instance variables in an overloaded
// method
#include "Hbclass.ch"

CLASS SubClass FROM BaseClass
   METHOD print
ENDCLASS


METHOD Print CLASS SubClass
   LOCAL oErr
   ?
   ? "---", ProcName() , "---"

   TRY
     ? "EXPORTED:  "
     ?? ::iExported
   CATCH oErr
     ?? oErr:description
   END

   TRY
     ? "PROTECTED: "
     ?? ::iProtected
   CATCH oErr
     ?? oErr:description
   END

   TRY
     ?  "HIDDEN:   "
     ?? ::iHidden
   CATCH oErr
     ?? oErr:description
   END
RETURN self

************************* Output ****************************
===================
Testing base class
```

```
===================

--- PRINT_PROC ---
EXPORTED:  iExported
PROTECTED: Scope Violation <PROTECTED>
HIDDEN:    Scope Violation <HIDDEN>

--- BASECLASS:PRINT ---
EXPORTED:  iExported
PROTECTED: iProtected
HIDDEN:    iHidden

================
Testing subclass
================

--- PRINT_PROC ---
EXPORTED:  iExported
PROTECTED: Scope Violation <PROTECTED>
HIDDEN:    Scope Violation <HIDDEN>

--- SUBCLASS:PRINT ---
EXPORTED:  iExported
PROTECTED: iProtected
HIDDEN:    Scope Violation <HIDDEN>

--- BASECLASS:PRINT ---
EXPORTED:  iExported
PROTECTED: iProtected
HIDDEN:    iHidden
```

CLASSDATA

Declares a class variable of a class.

Syntax

```
CLASSDATA <MemberName> [ INIT <expression> ] ;
                       [ READONLY ] [ SHARED ]
```

Arguments

`<MemberName>`

This is the symbolic name of the class variable to declare. It must begin with a letter or underscore followed by digits, letters or underscores. The symbolic name can contain up to 63 characters.

`INIT <expression>`

This is the expression which is executed when the class function is called. The return value of *<expression>* is assigned to *<MemberName>*.

`READONLY`

This option declares a class variable as read only. I.e. when an instance exists outside the declaring PRG module, no value can be assigned to *<MemberName>*. The value of *<MemberName>* can be changed within methods of the class.

`SHARED`

This option specifies that the class variable will be shared in all sub-classes of the declared class. If SHARED is omitted, sub-classes receive their own copy of the declared class variable.

Description

The CLASSDATA statement can only be used in the class declaration between CLASS and ENDCLASS. It declares the symbolic name of a class variable. This name must be sent to an object to access the class variable.

Class variables are a special type of member variables since the value of a class variable is the same for all instances of a class. Two objects of a class can hold different values in their instance variables but have the same values in their class variables.

An alternative initialization of class variables is provided with the :initClass() method (see HBObject()).

Info

See also:	ACCESS, ASSIGN, CLASS, DATA, EXPORTED:
Category:	Class declaration, Declaration, xHarbour extensions
Header:	hbclass.ch
Source:	vm\classes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Examples

```
// The example desmonstrates the nature of class variables using
// two objects of the same class. The class variable is changed
// with one object. This change is reflected also in the second
// object.

   #include "HbClass.ch"
```

```
PROCEDURE Main
   LOCAL objA, objB

   objA := Test():new( "One" )
   objB := Test():new( "Two" )

   ? objA:value, objA:config        // result: One Initial value
   ? objB:value, objB:config        // result: Two Initial value

   objB:config := "New value"

   ? objA:value, objA:config        // result: One New value
   ? objB:value, objB:config        // result: Two New value
RETURN

CLASS Test
   EXPORTED:
   CLASSDATA  config  INIT  "Initial value"

   DATA value

   METHOD init(x) INLINE ( ::value := x, self )
ENDCLASS
```

```
// The example outlines the difference between a SHARED and
// an unshared class variable. Class variables are assigned
// new values for the SubClass object. The changed value
// is only reflected in the SHARED class variable of the
// BaseClass object.

   #include "HbClass.ch"

   PROCEDURE Main
      LOCAL objA, objB

      objA := BaseClass( "First" ):new( "One" )
      objB := SubClass ( "Second"):new( "Two" )


      ? objA:value, objA:cPrivate, objA:cPublic
      objB:display()

      objB:cPrivate := "Hello"
      objB:cPublic  := "World"

      ? objA:value, objA:cPrivate, objA:cPublic
      objB:display()

      // ******** Output **********
      // One First Global
      // Two Second Global
      // One First World
      // Two Hello World
   RETURN

   // ------------
   CLASS BaseClass
      EXPORTED:
      CLASSDATA  cPrivate
      CLASSDATA  cPublic   INIT "Global" SHARED

      DATA value
```

```
   METHOD initClass
   METHOD init
ENDCLASS

METHOD initClass( x ) CLASS BaseClass
   ::cPrivate := x
 RETURN self

METHOD init( x ) CLASS BaseClass
   ::value := x
RETURN self

// --------------------------
CLASS SubClass FROM BaseClass
   EXPORTED:
   METHOD display
ENDCLASS

METHOD display CLASS SubClass
   ? ::value, ::cPrivate, ::cPublic
RETURN self
```

CLASSMETHOD

Declares the symbolic name of a class method.

Syntax

```
CLASSMETHOD <MethodName> [ (<params,...>] ) ] ;
   [ INLINE <expression> ]
```

Arguments

<MethodName>

This is the symbolic name of a class method to declare. It must begin with a letter or underscore followed by digits, letters or underscores. A symbolic name can contain up to 63 characters.

(<params,...>)

A declaration of formal parameters enclosed in parentheses is required when the class method is declared as INLINE. If INLINE is not used, the method can be declared without parameters.

INLINE <expression>

This is the expression which is executed when an INLINE method is invoked. *<expression>* must be one line of code. The code cannot contain commands but only function and method calls.

Description

The CLASSMETHOD statement exists for compatibility reasons. It is not different from the METHOD statement since the *self* object visible in methods is the same for METHOD and CLASSMETHOD.

There is one exception: :initClass() declared as CLASSMETHOD receives the parameters passed to the class function (see HbObject()).

Info

See also:	ACCESS, ASSIGN, CLASS, DATA, DELEGATE, EXPORTED:, HIDDEN:, INLINE METHOD, MESSAGE, METHOD (declaration), METHOD (implementation), PROTECTED:
Category:	Class declaration, Declaration, xHarbour extensions
Header:	hbclass.ch
Source:	vm\classes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

DATA

Declares an instance variable of a class.

Syntax

```
DATA <MemberName> [ INIT <expression> ] ;
                  [ READONLY ] [ PERSISTENT ]
```

Arguments

`<MemberName>`

This is the symbolic name of the instance variable to declare. It must begin with a letter or underscore followed by digits, letters or underscores. The symbolic name can contain up to 63 characters.

`INIT <expression>`

This is the expression which is executed when an object is created. The return value of *<expression>* is assigned to *<MemberName>*.

`READONLY`

This option declares an instance variable as read only. I.e. when the object exists outside the declaring PRG module, no value can be assigned to *<MemberName>*. The value of *<MemberName>* can be changed within methods of the class.

`PERSISTENT`

This option specifies that the value assigned to *<MemberName>* is persistent. When an object is serialized with HB_Serialize() and later restored with HB_Deserialize(), the value of *<MemberName>* is also restored. All instance variables not declared as PERSISTENT are initialized with NIL when a serialized object is restored from its character representation.

Description

The DATA statement can only be used in the class declaration between CLASS and ENDCLASS. It declares the symbolic name of an instance variable. This name must be sent to an object to access the instance variable.

All objects of a class have the same set of instance variables, but two objects can hold different values in their instance variables. They are normally initialized with default values using the INIT clause. The value of *<expression>* is assigned to an instance variable when the object is created.

An alternative initialization of instance variables is provided with the :init() method (see HBObject()).

Info

See also:	ACCESS, ASSIGN, CLASS, EXPORTED:, HBObject(), OVERRIDE METHOD, VAR
Category:	Class declaration, Declaration, xHarbour extensions
Header:	hbclass.ch
Source:	vm\classes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example implements a class whose objects hold file information.
// Two protected instance variables are used to collect file name and
// the results of function FileStats(). Exported instance variables
// are simulated with ACCESS and ASSIGN methods.
```

```
#include "HbClass.ch"

PROCEDURE Main
   LOCAL obj := FileInfo():new( "Test.prg" )

   ? obj:fileName           // result: Test.prg
   ? obj:fileSize           // result:       1718
   ? obj:fileAttr           // result: A
   ? obj:createDate         // result: 09/01/06
   ? obj:createTime         // result: 14:08:23

   obj:fileName := "Test2.prg"

   ? obj:fileName           // result: Test2.prg
   ? obj:fileSize           // result:        846
   ? obj:fileAttr           // result: A
   ? obj:createDate         // result: 11/17/05
   ? obj:createTime         // result: 09:44:34
RETURN


CLASS FileInfo
   PROTECTED:
   DATA    cFileName    INIT ""
   DATA    aInfo        INIT { "", 0, CtoD(""), 0, CtoD(""), 0 }
   METHOD  getInfo

   EXPORTED:
   METHOD  init

   ACCESS  fileName     INLINE ::cFileName
   ASSIGN  fileName(c)  INLINE ::getInfo(c)

   ACCESS  fileAttr     INLINE ::aInfo[1]
   ACCESS  fileSize     INLINE ::aInfo[2]
   ACCESS  createDate   INLINE ::aInfo[3]
   ACCESS  createTime   INLINE TString( ::aInfo[4] )
   ACCESS  changeDate   INLINE ::aInfo[5]
   ACCESS  changeTime   INLINE TString( ::aInfo[6] )
ENDCLASS

METHOD init( cFileName ) CLASS FileInfo
   IF Valtype( cFileName ) == "C"
      ::getInfo( cFileName )
   ENDIF
RETURN self

METHOD getInfo( cFileName ) CLASS FileInfo
   IF .NOT. File( cFileName )
      RETURN ::error( "File not found"         , ;
                      ::className(), "getInfo" , ;
                      EG_ARG, {cFileName}        )
   ENDIF

   ::cFileName := cFileName
   FileStats( cFileName, @::aInfo[1], ;
                         @::aInfo[2], ;
                         @::aInfo[3], ;
                         @::aInfo[4], ;
                         @::aInfo[5], ;
                         @::aInfo[6]  )
RETURN self
```

DELEGATE

Declares a message to be directed to a contained object.

Syntax

```
DELEGATE <Message> TO <ContainedObject>
```

Arguments

<Message>

This is the symbolic name of the message to delegate. It must begin with a letter or underscore followed by digits, letters or underscores. A symbolic name can contain up to 63 characters.

<ContainedObject>

This is the symbolic name of an instance variable holding the object that receives the message.

Description

The DELEGATE statement can only be used within the class declaration between CLASS and ENDCLASS. It declares a message that is automatically delegated to the object held in the instance variable *<ContainedObject>*. The message must exist in the class of *<ContainedObject>*, otherwise a runtime error is raised.

Message delegation is commonly used when the relationship between objects is based on *containment* rather than *inheritance*.

Info

See also:	ACCESS, CLASS, :
Category:	Class declaration, Declaration, xHarbour extensions
Header:	hbclass.ch
Source:	vm\classes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example outlines a typical usage scenario for message delegation.
// The Container class does not have the method :display(), but a
// Container object understands this message and delegates it to the
// contained Item object.

   #include "Hbclass.ch"

   PROCEDURE Main
      LOCAL obj := Item():new( "Hello World" )

      obj := Container():new( "xHarbour", obj )

      ? obj:name              // result: xHarbour
      obj:display()           // result: Hello World

      obj:item := Item():new( "New item" )

      obj:display()           // result: New item
   RETURN


   CLASS Container
```

```
   EXPORTED:
   DATA item
   DATA name

   METHOD init(x, o) INLINE (::name := x, ::item := o, self )

   DELEGATE display TO item
ENDCLASS


CLASS Item
   EXPORTED:
   DATA name

   METHOD init(x) INLINE (::name := x, self )

   METHOD display INLINE ( QOut(::name), self )
ENDCLASS
```

DESTRUCTOR

Declares a method to be called by the garbage collector.

Syntax

```
DESTRUCTOR <MethodName>
```

Arguments

<MethodName>

This is the symbolic name of the destructor method. It must begin with a letter or underscore followed by digits, letters or underscores. The symbolic name can contain up to 63 characters.

Description

The DESTRUCTOR statement can only be used within the class declaration between CLASS and ENDCLASS. It declares a method that is automatically called when an object becomes subject to garbage collection. This is the case when there is no memory variable left holding a reference to an object.

The implementation of the destructor method must follow the ENDCLASS statement and must use the PROCEDURE statement, rather than METHOD. This is required since a destructor method cannot have a return value. The implementation of a destructor method follows this coding pattern:

```
PROCEDURE <MethodName> CLASS <ClassName>
   <cleanup code>
RETURN
```

Destructor methods provide a convenient way of explicitely "cleaning up" memory or calling maintenance routines, like closing open files managed by an object.

Info

See also:	CLASS, ERROR HANDLER, PROCEDURE
Category:	Class declaration, Declaration, xHarbour extensions
Header:	hbclass.ch
Source:	vm\classes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example outlines usage of a destructor method
// being called when an object becomes subject to
// garbage collection. The output of the example shows
// the sequence of destruction as the objects become
// inaccessible in the course of the program.

   #include "HbClass.ch"

   PROCEDURE Main
      LOCAL obj := Test():new( "First object" )

      TestProc()
   RETURN

   // object stored in LOCAL variable
   PROCEDURE TestProc()
      LOCAL obj    := Test():new( "Second object" )
      LOCAL bBlock := TestFunc()
```

```
RETURN

// object stored in PUBLIC variable is released
// object stored in detached LOCAL is returned
FUNCTION TestFunc()
   LOCAL  bBlock := TestBlock()
   object := NIL
RETURN bBlock

// objects stored in detached LOCAL variable and PUBLIC variable
FUNCTION TestBlock()
   LOCAL obj := Test():new( "Third object" )

   PUBLIC object := Test():new( "Fourth object" )
RETURN {|| obj:name }


CLASS Test
   EXPORTED:
   DATA   name
   METHOD init

   DESTRUCTOR destroy                  // declaration of destructor
ENDCLASS

METHOD init( cName ) CLASS Test
   ::name := cName
RETURN self

PROCEDURE destroy CLASS Test           // implementation of destructor
  ? "Object:", ::name, "Destroyed in:", ProcName(1)
RETURN

// **************** Output ********************
// Object: Fourth object Destroyed in: TESTFUNC
// Object: Third object Destroyed in: TESTPROC
// Object: Second object Destroyed in: TESTPROC
// Object: First object Destroyed in: MAIN
```

DO CASE

Executes a block of statements based on one or more conditions.

Syntax

```
DO CASE
   CASE <Condition1>
        <Statements1>
 [ CASE <ConditionN>
        <StatementsN> ]
 [ OTHERWISE
        <defaultStatements> ]
END[CASE]
```

Arguments

`CASE <Condition>`

<Condition1> to *<ConditionN>* are logical expressions. The statements following the first expression that yields the value .T. (true) are executed.

`OTHERWISE`

If no condition results in the value .T. (true), the statements following the OTHERWISE option, if present, are executed.

Description

This statement executes a block of statements if the condition, related to the block, evaluates to true. If a condition evaluates to true, the following statements are executed until the next CASE, OTHERWISE or END[CASE] statement is encountered.

If no condition in the entire DO CASE structure evaluates to true, control is passed to the first statement following the ENDCASE statement.

It is possible to include other control structures like DO WHILE and FOR within a single DO CASE structure.

Note: The DO CASE...ENDCASE statement is equal to the IF...ELSEIF...ENDIF statement.

An alternative to DO CASE is the SWITCH statement which operates with constant values rather than logical conditions.

Info

See also: BEGIN SEQUENCE, DO WHILE, FOR, IF, SWITCH
Category: Control structures, Statements

Example

```
// This example shows the use of the DO CASE statement.

   PROCEDURE Main
      ACCEPT "Enter a number: " TO nNumber

      nNumber := Val(Alltrim(nNumber))

      DO CASE
         CASE nNumber = 1
            ? "the number was 1"
         CASE nNumber = 2
            ? "the number was 2"
     OTHERWISE
```

```
            ? "the number was not 1 or 2"
    ENDCASE
RETURN
```

DO WHILE

Executes a block of statements while a condition is true.

Syntax

```
[DO] WHILE <Condition>
  <Statements>
  [EXIT]
  <Statements>
  [LOOP]
  <Statements>
END[DO]
```

Arguments

`DO WHILE <Condition>`

<*Condition*> is a logical expression which controls the DO WHILE loop. Statements within the loop are repeated until <*Condition*> results in .F. (false).

`EXIT`

The EXIT statement unconditionally terminates a DO WHILE loop. Program flow branches to the first statement following the ENDDO statement.

`LOOP`

The LOOP statement branches control unconditionally to the DO WHILE statement, i.e. to the begin of the loop.

Description

The DO WHILE statement is a control structure that executes a block of statements repeatedly while a condition is true, or until the EXIT statement is executed. The logical expression <*Condition*> is evaluated as many times as the statements between DO WHILE and ENDDO are executed. When the ENDDO or LOOP statement is encountered, a next execution cycle begins with the evaluation of the <*Condition*> expression.

The EXIT statement terminates a DO WHILE loop unconditionally and control is branched to the nearest ENDDO statement without evaluating <*Condition*>.

Info

See also: AEval(), BEGIN SEQUENCE, DbEval(), DO CASE, FOR, FOR EACH, IF, RETURN
Category: Control structures, Statements

Example

```
// The example demonstrates a typical usage of a DO WHILE loop.
// A database file is scanned until the end-of-file is reached.

   PROCEDURE Main
      LOCAL aRecno := {}

      USE Customer

      DO WHILE ! Eof()
         IF Customer->City = "New York"
            AAdd( aRecno, Recno() )
         ENDDO
         SKIP
      ENDDO
```

```
      ? Len( aRecno ), "customers in New York"
      USE
   RETURN
```

ENABLE TYPE CLASS

Activates scalar classes for native data tyes.

Syntax

```
ENABLE TYPE CLASS ALL

or

ENABLE TYPE CLASS <datatype,...>
```

Arguments

`ALL`

This option enables all scalar classes for native data types.

`CLASS <datatype,...>`

A subset of scalar classes can be enabled by specifying one or more data types in a comma separated list. The following keywords are recognized for *<datatype>*: ARRAY, BLOCK, CHARACTER, DATE, HASH, LOGICAL, NIL, NUMERIC and POINTER.

Description

When scalar classes are enabled for native data types, a variable of the respective data type can be used like an object. Messages can be sent to the variable using the send operator. All scalar classes understand the messages :isScalar() and :asString().

The documentation on scalar classes is preliminary and may change. Refer to the file source\rtl\tclass.prg for the current implementation of scalar classes.

Info

See also:	ASSOCIATE CLASS, CLASS, EXTEND CLASS...WITH METHOD, PROCEDURE
Category:	Class declaration, Declaration, xHarbour extensions
Header:	hbclass.ch
Source:	rtl\tclass.prg
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates usage of scalar classes.

   #include "HbClass.ch"

   PROCEDURE Main()
      LOCAL aVar := {}
      LOCAL bDisplay := {|x| Qout( x:asString() ) }
      LOCAL pPointer := ( @Main() )

      ENABLE TYPE CLASS ALL

      CLS

      aVar:Init( 2 )
      aVar:AtPut( 1, "One" )
      aVar:AtPut( 2, 2 )

      aVar:InsertAt( 3, "Three" )
```

```
    Alert( "Found at pos: " + aVar:IndexOf( "Three" ):asString() )

    aVar:Do( bDisplay )

    aVar:AddAll( { 4, "Five", 6 } )

    ? aVar:AsString
    ? bDisplay:asString()
    ? pPointer:asString()
RETURN
```

ERROR HANDLER

Declares the method for error handling within a class.

Syntax

```
ERROR HANDLER <MethodName>
```

Arguments

`<MethodName>`

This is the symbolic name of the error handling method. It must begin with a letter or underscore followed by digits, letters or underscores. The symbolic name can contain up to 63 characters.

Description

The ERROR HANDLER statement can only be used within the class declaration between CLASS and ENDCLASS. It declares a method that replaces the default method invoked when an object receives a message not declared in the object's class or super class(es).

A message is not understood by an object, when the message name identifies neither a declared member variable nor a declared method in the class or its super classes.

The default method for handling such an error condition is :msgNotFound(*<cMsg>*).

The implementation of the error handling method must follow the ENDCLASS statement using METHOD, just as with regular methods.

Info

See also:	CLASS, DESTRUCTOR, METHOD (implementation)
Category:	Class declaration, Declaration, xHarbour extensions
Header:	hbclass.ch
Source:	vm\classes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example implements an error handler which collects
// the message sent to an object along with parameters and forwards
// them to the overloaded :msgNotFound() method. All messages
// sent to the object in Main do not exist in the Test class.

   #include "HbClass.ch"

   PROCEDURE Main
      LOCAL obj := Test():new()

      ? obj:one
      ? obj:two := 2
      ? obj:three( 1, 2, 3 )
   RETURN


   CLASS Test
      EXPORTED:
      METHOD msgNotFound
      ERROR HANDLER noMessage
   ENDCLASS
```

```
METHOD msgNotFound( cMessage, aParams ) CLASS Test
   LOCAL x

   ? "Message not found:", cMessage
   FOR EACH x IN aParams
      ?? "" , ValToPrg( x )
   NEXT
RETURN self

METHOD noMessage( ... ) CLASS Test
   LOCAL aParams := HB_AParams()
RETURN ::msgNotFound( __GetMessage(), aParams )
```

EXIT PROCEDURE

Declares a procedure to execeute when a program terminates.

Syntax

```
EXIT PROCEDURE <procName>
  [FIELD <fieldName,...> [IN <aliasName>]]
 [MEMVAR <var_Dynamic,...>]
  [LOCAL <var_Local> [:= <expression>] ,... ]

       <Statements>

[RETURN]
```

Arguments

`EXIT PROCEDURE <procName>`

This is the symbolic name of the declared procedure. It must begin with a letter or underscore followed by digits, letters or underscores. The symbolic name can contain up to 63 characters.

`FIELD <fieldName>`

An optional list of field variables to use within the EXIT procedure can be declared with the FIELD statement.

`MEMVAR <var_Dynamic>`

If dynamic memory variables, i.e. PRIVATE or PUBLIC variables, are used in the EXIT procedure, they are declared with the MEMVAR statement.

`LOCAL <var_Local> [:= <expression>]`

Local variables are declared and optionally initialized using the LOCAL statement.

`RETURN`

The RETURN statement terminates an EXIT procedure and branches control to the next EXIT procedure. After the last EXIT procedure has returned, control goes back to the operating system.

Description

The EXIT PROCEDURE statement declares a procedure that is automatically called when a program terminates regularly, i.e. when the QUIT statement is executed or when control goes back to the operating system following a RETURN statement in the main routine of a program. When a program ends due to a non-recoverable runtime error, or when ALT+C is pressed, no EXIT procedure is called.

EXIT procedures do not have a list of arguments and no parameters are passed on program termination. The execution order of EXIT procedures is undetermined. It is only guaranteed that they are called when a program ends.

An EXIT procedure cannot be called explicitely during runtime of a program since its symbolic name is resolved at compile time and does not exist at runtime.

Info

See also: INIT PROCEDURE, PROCEDURE
Category: Declaration, Statements

Example

```
// The example uses INIT and EXIT procedures to make sure that all
// database and index files are open at program start and properly
```

```
// closed on program termination.

   PROCEDURE Main
      WAIT
      Browse()
   RETURN

   INIT PROCEDURE OpenDatabase
      CLS
      ? "Init program"
      IF ! File( "Cust1.ntx" )
         USE Customer ALIAS Cust EXCLUSIVE
         INDEX ON Upper( LastName + FIrstName ) TO Cust1.ntx
         USE
      ENDIF

      IF ! File( "Cust2.ntx" )
         USE Customer ALIAS Cust EXCLUSIVE
         INDEX ON Upper( City ) TO Cust2.ntx
         USE
      ENDIF

      USE Customer ALIAS Cust SHARED INDEX Cust1.ntx, Cust2.ntx
   RETURN

   EXIT PROCEDURE CloseDatabase
      ? "Exit program"
      SELECT Cust
      COMMIT
      CLOSE DATABASE
   RETURN
```

EXPORTED:

Declares the EXPORTED attribute for a group of member variables and/or methods.

Syntax

```
EXPORTED:
```

Description

EXPORTED: is the default visibility attribute, or Scope, for member variables and methods of classes. It is used within the class declaration to specify that subsequent DATA and METHOD declarations declare EXPORTED member variables and methods of a class.

Access to exported member variables and methods is not restricted. The visibility attributes PROTECTED: and HIDDEN: restrict class member access.

Info

See also:	CLASS, DATA, HIDDEN:, METHOD (declaration), PROTECTED:
Category:	Class declaration, Declaration, xHarbour extensions
Header:	hbclass.ch
Source:	vm\classes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

EXTEND CLASS...WITH DATA

Adds a new member variable to an existing class.

Syntax

```
EXTEND CLASS <ClassName> WITH DATA <MemberName> [ PERSISTENT ]
```

Arguments

<ClassName>

This is the symbolic name of an existing class to add a new instance variable to.

<MemberName>

This is the symbolic name of the new instance variable.

PERSISTENT

This option specifies that the value assigned to *<MemberName>* is persistent. When an object is serialized with HB_Serialize() and later restored with HB_Deserialize(), the value of *<MemberName>* is also restored. All instance variables not declared as PERSISTENT are initialized with NIL when a serialized object is restored from its character representation.

Description

The EXTEND CLASS...WITH DATA statement allows for adding new instance variables to a declared class outside the class declaration. This way, a class can be enhanced even if the source code of the class is not available. The EXTEND CLASS statemend can modify all classes existing in xHarbour, except for scalar classes. Scalar classes cannot be extended with data, since a scalar object is the data.

Note: an alternative for enhancing an existing class is to use the FROM option of the CLASS statement and create a sub-class of an existing class.

Info

See also:	CLASS, EXTEND CLASS...WITH METHOD, METHOD...OPERATOR, OPERATOR, OVERRIDE METHOD
Category:	Class declaration, Declaration, xHarbour extensions
Header:	hbclass.ch
Source:	vm\classes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example adds an instance variable to the "root" class of
// xHarbour: HBObject(). All new classes inherit from this class.
// As a result, the user-defined Test() class inherits this
// new member, although :extraData is neither declared in
// HBObject() nor in Test().

   #include "HbClass.ch"

   PROCEDURE Main
      LOCAL obj

      EXTEND CLASS HBObject WITH DATA extraData

      obj := Test():new( "xHarbour" )
```

```
   ? obj:extraData := "a new data slot for all classes"
RETURN


CLASS Test
   DATA value
   METHOD init(x) INLINE (::value := x, self)
ENDCLASS
```

EXTEND CLASS...WITH METHOD

Adds a new method to an existing class.

Syntax

```
EXTEND CLASS <ClassName>    WITH ;
   [ MESSAGE <MessageName>] METHOD <FunctionName>

or

EXTEND CLASS <ClassName>    WITH ;
     MESSAGE <MessageName>  INLINE <expression>
```

Arguments

<ClassName>

This is the symbolic name of an existing class to add a new method to.

MESSAGE <MessageName>

This is the symbolic name of the message which invokes the new method.

METHOD <FunctionName>

This is the symbolic name of the function implementing the code for the new method. If *<MessageName>* is omitted, *<FunctionName>* defines also the message that must be sent to an object for invoking the method.

INLINE <expression>

This is the expression which is executed when an INLINE method is invoked. *<expression>* must be one line of code. The code cannot contain commands but only function and method calls.

Description

The EXTEND CLASS...WITH METHOD statement allows for adding new methods to a declared class outside the class declaration. This way, a class can be enhanced even if the source code of the class is not available. The EXTEND CLASS statemend can modify all classes existing in xHarbour.

The new method is either implemented as a STATIC FUNCTION or as an INLINE expression. The *self* object is retrieved in both cases with function HB_QSelf().

Note: an alternative for enhancing an existing class is to use the FROM option of the CLASS statement and create a sub-class of an existing class.

Info

See also:	ASSOCIATE CLASS, CLASS, EXTEND CLASS...WITH DATA, HB_QSelf(), METHOD...OPERATOR, OPERATOR, OVERRIDE METHOD
Category:	Class declaration, Declaration, xHarbour extensions
Header:	hbclass.ch
Source:	vm\classes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example adds a method to the "root" class of xHarbour: HBObject().
// All new classes inherit from this class. This is used in the example
// for tracing variables depending on their data types in different log
// files.
```

```
#include "HbClass.ch"

PROCEDURE Main
   LOCAL obj, nValue := 2, dDate := Date()

   EXTEND CLASS HBObject WITH MESSAGE log METHOD LogData

   ENABLE TYPE CLASS ALL

   obj := Test():new( "xHarbour" )

   obj:log( "Objects.log" )

   nValue:log( "Numerics.log" )

   dDate:log( "Dates.log" )
RETURN


CLASS Test
   EXPORTED:
   DATA value
   METHOD init(x) INLINE (::value := x, self)
ENDCLASS


STATIC FUNCTION LogData( cLogFile )
   LOCAL self := HB_QSelf()
   LOCAL cOldFile

   IF .NOT. Set( _SET_TRACE )
      RETURN self
   ENDIF

   IF Valtype( cLogFile ) == "C"
      cOldFile := Set( _SET_TRACEFILE, cLogFile )
   ENDIF

   Tracelog( self )

   IF cOldFile <> NIL
      Set( _SET_TRACEFILE, cOldFile )
   ENDIF
RETURN self
```

EXTERNAL

Declares the symbolic name of an external function or procedure for the linker.

Syntax

```
EXTERNAL <name1> [,<nameN>]
```

Arguments

`EXTERNAL <name>`

This is the symbolic name of a function or procedure to declare for the linker. When multiple names are declared, they must be separated with commas.

Description

The EXTERNAL statement declares a symbolic name of a function or procedure for the linker. This is usually required when there is no direct call of a function or procedure in PRG code, for example when a function is only called within a macro-expression using the macro-operator. By declaring the symbolic name of a function as EXTERNAL, the linker is forced to link the corresponding function to the executable file.

Note: It is common practice to use the EXTERNAL declaration within an #include file. This way it is assured that functions are available in all PRG files that may call them indirectly.

Info

See also: #include, ANNOUNCE, REQUEST
Category: Declaration, Statements

Example

```
// The example forces the linker to link three functions that are only
// called within a macro expression.

   EXTERNAL Memoedit, Memoread, Memowrit
   MEMVAR   fileName
   STATIC   cEditor := "Memowrit(fileName,Memoedit(Memoread(fileName)))"

   PROCEDURE Main( cFile )
      fileName := cFile
      &cEditor
   RETURN
```

FIELD

Declares a field variable

Syntax

```
FIELD <fieldName,...> [IN <aliasName>]
```

Arguments

`FIELD <fieldNames,...>`

This is a comma separated list of symbolic names identifying field variables.

`IN <aliasName>`

Optionally, the alias name of the work area that holds field variables can be specified.

Description

The FIELD statement declares symbolic names of variables whose values are stored in database files. This instructs the compiler to resolve unaliased variable names to field variables, not memory variables. All variables listed in *<fieldNames,...>* that appear in program code without alias name and alias operator, are treated as if they are listed with the FIELD-> alias name, or the one specified with the IN clause.

The FIELD statement does not verify if the field variables exist at runtime or if a database is open with *<aliasName>* as name. When the declared field variable does not exist at runtime, an error is generated.

The scope of the FIELD variable depends on the place of declaration:

1. When the FIELD declaration appears at the top of a PRG file before any other executable statement, the declaration has file wide scope, i.e. it is valid throughout the entire PRG file.
2. When the FIELD declaration follows a FUNCTION, METHOD or PROCEDURE declaration, the variable is treated as FIELD variable only in the routine that declares the FIELD.

The lines in PRG source code preceding the FIELD declaration may not call executable code. They can only contain declaration statements, i.e. only the FUNCTION, METHOD, PROCEDURE statements, and the LOCAL, MEMVAR, PARAMETERS, or STATIC variable declarations are allowed to precede the FIELD statement.

Info

See also: FUNCTION, GLOBAL, LOCAL, MEMVAR, PROCEDURE, STATIC
Category: Declaration, Statements

Example

```
PROCEDURE Main
   FIELD InvoiceNo, AmountDue, Payment  IN Inv
   FIELD CustNo   , FirstName, LastName IN Cust

   USE Customer ALIAS Cust SHARED
   INDEX ON CustNo TO CustA

   USE Invoice ALIAS Inv SHARED NEW
   SET RELATION TO CustNo INTO Cust
   GO TOP

   SET ALTERNATE TO NotPaid.txt
   SET ALTERNATE ON
```

```
    DO WHILE .NOT. Eof()
       IF Payment == 0
          ? CustNo, Lastname, FirstName, InvoiceNo, AmountDue
       ENDIF
       SKIP
    ENDDO

    SET ALTERNATE TO
    DbCloseAll()
 RETURN
```

FOR

Executes a block of statements a specific number of times.

Syntax

```
FOR <nCounter> := <nStart> TO <nEnd> [STEP <nIncrement>]
  <Statements>
  [EXIT]
  <Statements>
  [LOOP]
NEXT
```

Arguments

FOR <nCounter> := <nStart>

<*nCounter*> is the name of the variable that holds a counter value used to control the number of iterations of the FOR...NEXT loop. If <*nCounter*> does not exist when the loop is entered, it is created as a PRIVATE variable. The value <*nStart*> is a numeric start value to initialize <*nCounter*> with when the FOR statement is executed for the first time.

TO <nEnd>

<*nEnd*> is a numeric value that controls the termination of the FOR...NEXT loop. As soon as <*nCounter*> is greater than or equal to <*nEnd*>, the statements between FOR and NEXT are no longer executed.

STEP <nIncrement>

<*nIncrement*> is a numeric value <*nCounter*> is incremented with for each iteration of the loop. If not specified, it defaults to 1. When <*nIncrement*> is a negative value, the loop ends when <*nCounter*> is smaller than or equal to <*nEnd*>.

EXIT

The EXIT statement unconditionally terminates a FOR...NEXT loop. Program flow branches to the first statement following the NEXT statement.

LOOP

The LOOP statement branches control unconditionally to the FOR statement, i.e. to the begin of the loop.

Description

The FOR...NEXT statements form a control structure that executes a block of statements for a specific number of times. This is controlled by the <*nCounter*> counter variable which gets assigned the value <*nStart*> when the loop is entered for the first time. The loop iterates the statements between FOR and NEXT and adds the value <*nIncrement*> to <*nCounter*> each time the NEXT statement is reached or a LOOP statement is executed.

The value <*nEnd*> identifies the largest value for <*nCounter*> (<*nIncrement*> is positive) or its smallest value (<*nIncrement*> is negative) after which the FOR...NEXT loop terminates.

Note: The expressions in the FOR statement are evaluated each time a new iteration begins. As a consequence, new values can be assigned to <*nCounter*> or <*nEnd*> while the loop executes.

Info

See also: AEval(), FOR EACH, DO CASE, DO WHILE, IF, SWITCH, WITH OBJECT
Category: Control structures, Statements

Example

```
// This example shows a typical scenario for a FOR..NEXT loop
// where an array is filled with data, and stored data is output.

   PROCEDURE Main
      LOCAL aFields, i

      USE Customer
      aFields := Array( FCount() )

      // fill array in regular order
      FOR i:=1 TO FCount()
         aArray[i] := FieldName(i)
      NEXT

      USE

      // output data in reverse order
      FOR i:=Len( aArray ) TO 1 STEP -1
         ? aArray[i]
      NEXT
   RETURN
```

FOR EACH

Iterates elements of data types that can be seen as a collection.

Syntax

```
FOR EACH <element> IN <array>|<object>|<string>
   <statements>
   [LOOP]
   <statements>
   [EXIT]
NEXT
```

Arguments

<element>

The name of a variable that gets assigned a new value on each iteration.

IN <array>

This is a value of data type Array. The FOR EACH loop iterates all array elements in the first dimension and assigns their values to <*element*>.

IN <object>

This is a value of data type Object. The FOR EACH loop iterates all instance variables of the object and assigns their values to <*element*>.

IN <string>

This is a value of data type Character string. The FOR EACH loop iterates all individual characters of the string and assigns them to <*element*>.

LOOP

The LOOP statement unconditionally branches to the FOR EACH statement, i.e. to the begin of the loop, where the next value is assigned to <*element*>.

EXIT

The EXIT statement unconditionally terminates the FOR EACH loop and branches to the statement following NEXT.

Description

The FOR EACH statement forms a control structure that executes a block of statements for a data type containing multiple elements. This can be data of type Array, Object or Character string. The loop iterates all elements contained in the data and assigns the value of the next element to <*element*> on each iteration.

FOR EACH is similar to the regular FOR loop. But it completes considerably faster than a FOR loop, since there is no explicit loop counter. In contrast, FOR EACH uses an implicit loop counter whose value can be queried using the function HB_EnumIndex(). Other than this, LOOP and EXIT statements within the loop are treated the same as in a FOR loop.

When FOR EACH statements are nested, each loop maintains its own counter, i.e. HB_EnumIndex() retrieves the counter of the loop that is currently executed. When the FOR EACH loop is finished, its loop counter is set to 0.

Info

See also: AEval(), HB_EnumIndex(), FOR, DO CASE, DO WHILE, IF, SWITCH, WITH OBJECT
Category: Control structures, Statements, xHarbour extensions

Example

```
// The example demonstrates the FOR EACH statement using two
// nested loops. The outer loop iterates an array while the
// inner loop iterates character strings.

   PROCEDURE Main()
      LOCAL aArray := { "Hello", "World" }
      LOCAL cString, cChar

      FOR EACH cString IN aArray
         ? "----- Outer loop -----"
         ? HB_EnumIndex(), cString

         ? "----- Inner loop -----"
         FOR EACH cChar IN cSTring
            ? HB_EnumIndex(), cChar
         NEXT
      NEXT

      ? "-------- End ---------"
      ? HB_EnumIndex()
   RETURN

/* Output of example:
   ----- Outer loop -----
         1 Hello
   ----- Inner loop -----
            1 H
            2 e
            3 l
            4 l
            5 o
   ----- Outer loop -----
            2 World
   ----- Inner loop -----
            1 W
            2 o
            3 r
            4 l
            5 d
   -------- End ---------
            0
*/
```

FUNCTION

Declares a function along with its formal parameters.

Syntax

```
[STATIC] [UTILITY] FUNCTION <funcName>( [<params,...>] )
           [FIELD <fieldName,...> [IN <aliasName>]]
          [MEMVAR <var_Dynamic,...>]
           [LOCAL <var_Local>  [:= <expression>] ,... ]
          [STATIC <var_Static> [:= <expression>] ,... ]

            <Statements>

           RETURN <retVal>
```

Arguments

`FUNCTION <funcName>( [<params,...>] )`

This is the symbolic name of the declared function. It must begin with a letter or underscore followed by digits, letters or underscores. The symbolic name can contain up to 63 characters.

Optionally, the names of parameters *<params,...>* accepted by the function can be specified as a comma separated list. These function parameters have LOCAL scope within the function.

When the function is declared as STATIC FUNCTION, it is only visible within the PRG file that contains the function declaration and cannot be invoked from elsewhere.

`UTILITY`

The UTILITY attribute indicates that the declared function should not be used as startup code even if it is the first declared function in the PRG source file.

`FIELD <fieldName>`

An optional list of field variables to use within the FUNCTION can be declared with the FIELD statement.

`MEMVAR <var_Dynamic>`

If dynamic memory variables, i.e. PRIVATE or PUBLIC variables, are used in the function, they are declared with the MEMVAR statement.

`LOCAL <var_Local> [:= <expression>]`

Local variables are declared and optionally initialized using the LOCAL statement.

`STATIC <var_Static> [:= <expression>]`

Static variables are declared and optionally initialized using the STATIC statement.

`RETURN <retVal>`

The RETURN statement terminates a function and branches control back to the calling routine, returning the value *<retVal>* to it.

Description

The FUNCTION statement declares a function along with an optional list of parameters accepted by the function. Statements programmed in the function body form a self-contained part of a program that is executed when a function is called. Thus, tasks of a program can be split into several functions, each of which performs a sub-task when invoked.

The body of a function ends with the next FUNCTION, PROCEDURE or CLASS declaration, or at the end of file, which implies that function declarations cannot be nested.

The execution of a function ends when a RETURN statement is encountered in the function body, which must return a value <*retVal*> to the calling routine. This value is mandatory for functions and can be of any data type. The RETURN value is the only difference between functions and procedures.

When a function is declared with the STATIC modifier, its visibility is restricted to the PRG file that contains the STATIC FUNCTION declaration. The names of STATIC functions are resolved by the compiler and do not exist at runtime of a program. The names of non-STATIC functions, also referred to as public functions, are resolved by the linker and do exist at runtime. Thus, public functions can be accessed by the Macro operator (&) while STATIC functions cannot.

It is possible to declare STATIC functions with the same symbolic name in different PRG files. A name conflict to a public function with the same name declared in another PRG file does not arise. However, the symbolic names of public functions, procedures or classes must always be unique.

When a function is invoked with values being passed to it, they are assigned to the formal parameters declared with <*params,...*>. All variables declared in this list are LOCAL variables and their visibility is restricted to the statements programmed in the function body.

The number of values passed to a function does not need to match the number of parameters declared. When fewer values are passed, the corresponding parameters are initialized with NIL. When more values are passed, the additional values are not asssigned to parameters but can be retrieved using function HB_AParams().

Undefined parameter list a function can be declared with an undefined parameter list using the ellipsis sign (...) as a placeholder for the parameter list. A function declared in such way can receive an unknown number of parameters.

Info

See also: FIELD, HB_AParams(), LOCAL, MEMVAR, METHOD (declaration), PARAMETERS, PCount(), PRIVATE, PROCEDURE, PUBLIC, RETURN, STATIC
Category: Declaration, Statements

Examples

```
// The example shows two functions used to calculate the number of
// days for a month.

   PROCEDURE Main
      ? DaysOfMonth( StoD( "20000201" ) ) // Result: 29
      ? DaysOfMonth( StoD( "20010201" ) ) // Result: 28
      ? DaysOfMonth( StoD( "20040201" ) ) // Result: 29
      ? DaysOfMonth( StoD( "20050301" ) ) // Result: 31
      ? DaysOfMonth( StoD( "20051101" ) ) // Result: 30
   RETURN


   FUNCTION DaysOfMonth( dDate )
      LOCAL nMonth

      IF Valtype( dDate ) <> "D"
         dDate := Date()
      ENDIF

      nMonth := Month( dDate )

      IF nMonth == 2
         RETURN IIf( IsLeapYear( dDate ), 29, 28 )
      ELSEIF nMonth $ {4,6,9,11}
         RETURN 30
      ENDIF
   RETURN 31
```

```
   STATIC FUNCTION IsLeapYear( dDate )
      LOCAL nYear := Year( dDate )
   RETURN ( nYear % 4 == 0 ) .OR. ( nYear % 400 == 0 )

// The example demonstrates how an unknown number of command line
// arguments passed to an xHarbour application can be processed.

    PROCEDURE Main( ... )
       LOCAL aArg := HB_AParams()
       LOCAL cArg

       FOR EACH cArg IN aArg
          DO CASE
          CASE Upper( cArg ) IN ("-H/H-?/?")
             ? "Help requested"

          CASE .NOT. cArg[1] IN ( "-/" )
             ?? " argument:", cArg

          CASE Upper( cArg ) IN ("-X/X")
             ? "Execution requested"

          OTHERWISE
             ? "Unknown:", cArg
          ENDCASE
       NEXT

    RETURN
```

GLOBAL

Declares and optionally initializes a GLOBAL memory variable.

Syntax

```
GLOBAL <varName> [:= <xValue> ]
GLOBAL EXTERNAL <extVarName>
```

Arguments

`GLOBAL <varName>`

<varName> is the symbolic name of the global variable to declare.

`<xValue>`

<xValue> is an optional value to assign to the GLOBAL variable after being declared. To assign a value, the inline assignment operator (:=) must be used. The simple assignment operator (=) cannot be used. Only literal values are allowed for *<xValue>* when declaring GLOBAL variables.

`GLOBAL EXTERNAL <extVarName>`

This is the name of a GLOBAL variable that is declared in another PRG file and must be referred to in the current PRG file.

Description

The GLOBAL statement declares a memory variable that has GLOBAL scope. Global variables are resolved by the compiler, i.e. their symbolic name cannot be retrieved during runtime. This makes access to GLOBAL variables much faster than to memory variables of PRIVATE or PUBLIC scope, whose symbolic variable names exist at runtime.

The names of GLOBAL variables cannot be included in macro-expressions since they cannot be resolved by the macro operator (&). This operator requires the symbolic name of a variable to exist at runtime.

The lifetime of GLOBAL variables is identical with the lifetime of a program. They are visible and accessible throughout the functions and procedures of all PRG files that declare or refer to global variables.

Variable of other types cannot have the same symbolic name as a GLOBAL variable since a global variable's name must be unique for all modules accessing such a variable. In addition, a GLOBAL can only be declared in one PRG file using the GLOBAL statement. When a global variable must be accessed in another PRG file, it must be referred to using the GLOBAL EXTERNAL statement.

Info

See also: LOCAL, MEMVAR, PRIVATE, PUBLIC, STATIC
Category: Declaration, Statements, xHarbour extensions

Example

```
// The example demonstrates how to declare/initialize a global variable
// in one file, while it is referred to in a second file.

   ** Global1.prg

   GLOBAL gName := "My Global"

   PROCEDURE Main
      ? gName                // result: My Global
```

```
   Test()
   ? gName                  // result: New Name
RETURN

** Global2.prg

GLOBAL EXTERNAL gName

PROCEDURE Test
   gName := "New Name"
RETURN
```

HIDDEN:

Declares the HIDDEN attribute for a group of member variables and/or methods.

Syntax

```
HIDDEN:
```

Description

The HIDDEN: attribute restricts access to member variables and methods of a class. Hidden members can only be accessed in the methods of the class(es) declared in a PRG module. It is the most restrictive attribute. Hidden members cannot be accessed in the context of a FUNCTION or PROCEDURE, only in the context of a METHOD implemented in the declaring PRG module.

A less restrictive visibility attribute is PROTECTED:.

Info

See also: CLASS, DATA, EXPORTED:, METHOD (declaration), PROTECTED:
Category: Class declaration, Declaration, xHarbour extensions
Header: hbclass.ch
Source: vm\classes.c
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example uses two files to demonstrate the protection
// of a hidden member variable outside the declaring PRG module.

   ** MAIN.PRG uses the class
   PROCEDURE Main
      LOCAL obj

      obj := Test():new( "Hello", "World" )
      obj:print()             // result: Hello World

      ? obj:varTwo            // result: World
      ? obj:varOne            // Scope Violation <HIDDEN>: VARONE
   RETURN


   ** TEST.PRG declares the class
   #include "Hbclass.ch"

   CLASS Test
      HIDDEN:
      DATA varOne

      EXPORTED:
      DATA varTwo

      METHOD init
      METHOD print
   ENDCLASS

   METHOD init( p1, p2 ) CLASS Test
        ::varOne := p1
        ::varTwo := p2
   RETURN self
```

```
METHOD print CLASS Test
   ? ::varOne, ::varTwo
RETURN self
```

IF

Executes a block of statements based on one or more conditions.

Syntax

```
IF <Condition1>
  <Statements>
[ ELSEIF <ConditionN>
  <Statements> ]
[ ELSE
  <Statements> ]
END[IF]
```

Arguments

`IF <Condition1> [ELSEIF <ConditionN>]`

<Condition1> to *<ConditionN>* are logical expressions. The statements following the first expression that yields the value .T. (true) are executed.

`ELSE`

If no condition results in the value .T. (true), the statements following the ELSE statement, if present, are executed.

Description

This statement executes a block of statements if the condition, related to the block, evaluates to true. If a condition evaluates to true, the following statements are executed until the next ELSEIF, ELSE or END[IF] statement is encountered.

If no condition in the entire IF...ELSEIF structure evaluates to true, control is passed to the first statement following the ELSE statement.

It is possible to include other control structures like DO WHILE and FOR within a single IF...ELSE structure.

Note: The IF...ELSEIF...ENDIF statement is equal to the DO CASE...ENDCASE statement.

Info

See also: BEGIN SEQUENCE, DO CASE, DO WHILE, FOR, If() | IIf(), SWITCH
Category: Control structures, Statements

Example

```
// The example shows different forms of the IF END[IF] statement.

   PROCEDURE Main
      LOCAL nSalary := 1600

      IF .F.
         ? "This will never be executed"
      END

      IF .T.
         ? "This will always be executed"
      ENDIF

      IF nSalary < 1000
         ? "Start looking for another job!"
      ELSEIF nSalary < 1500
```

```
      ? "That's pretty good"
   ELSEIF nSalary < 2000
      ? "You should be happy"
   ELSE
      ? "You should be very happy"
   ENDIF
RETURN
```

INIT PROCEDURE

Declares a procedure to exeecute when a program starts.

Syntax

```
INIT PROCEDURE <procName>
  [FIELD <fieldName,...> [IN <aliasName>]]
 [MEMVAR <var_Dynamic,...>]
  [LOCAL <var_Local> [:= <expression>] ,... ]

       <Statements>

[RETURN]
```

Arguments

INIT PROCEDURE <procName>

This is the symbolic name of the declared procedure. It must begin with a letter or underscore followed by digits, letters or underscores. The symbolic name can contain up to 63 characters.

FIELD <fieldName>

An optional list of field variables to use within the INIT procedure can be declared with the FIELD statement.

MEMVAR <var_Dynamic>

If dynamic memory variables, i.e. PRIVATE or PUBLIC variables, are used in the INIT procedure, they are declared with the MEMVAR statement.

LOCAL <var_Local> [:= <expression>]

Local variables are declared and optionally initialized using the LOCAL statement.

RETURN

The RETURN statement terminates an INIT procedure and branches control to the next INIT procedure. After the last INIT procedure has returned, control goes to the main, or root, procedure of an application.

Description

The INIT PROCEDURE statement declares a procedure that is automatically called when a program is started. There is one implicit INIT procedure named ErrorSys() that is invoked in all xHarbour applications. It installs the default error codeblock and is programmed in ERRORSYS.PRG.

INIT procedures do not have a list of arguments and no parameters are passed on program start. The execution order of INIT procedures is undetermined. It is only guaranteed that they are called when a program starts.

An INIT procedure cannot be called explicitely during the start sequence of a program since its symbolic name is resolved at compile time and does not exist at runtime.

Info

See also: EXIT PROCEDURE, PROCEDURE
Category: Declaration, Statements

Example

```
// See the example for EXIT PROCEDURE
```

INLINE METHOD

Declares and implements an inline method that spans across multiple lines.

Syntax

```
INLINE METHOD <MethodName>( [<params,...>] )
  <.. program code ..>
ENDMETHOD
```

Arguments

`<MethodName>`

This is the symbolic name of the method to implement. It must begin with a letter or underscore followed by digits, letters or underscores. The symbolic name can contain up to 63 characters.

`<params,...>`

This is a acomma separated list of formal parameters accepted by the method.

`ENDMETHOD`

This ends the implementation of an INLINE METHOD that spans across multiple lines.

Description

Methods are declared within the class declaration. When a method is also **implemented** between CLASS and ENDCLASS, it is called an INLINE method. The INLINE METHOD declaration declares an inline method whose implementation spans across multiple lines. The end of this implementation must be indicated with the ENDMETHOD statement.

INLINE METHOD exists for compatibility reasons. It is recommended to declare a method with METHOD (declaration) and implement it separately with METHOD (implementation).

Note: if a method can be implemeted with one line of code, the INLINE option of the METHOD (declaration) can be used.

Info

See also:	CLASS, METHOD (declaration), METHOD (implementation)
Category:	Class declaration, Declaration, xHarbour extensions
Header:	hbclass.ch
Source:	vm\classes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example implements a class whose methods are entirely implemented
// INLINE. Objects of the class extract subsequent lines of an ASCII text
// until no more text lines are available.

   #include "hbclass.ch"

   PROCEDURE Main
      LOCAL obj := LineParser():new( Memoread( "Test.prg" ) )

      DO WHILE .NOT. obj:eof()
         ? obj:line
         obj:nextLine()
      ENDDO
   RETURN
```

```
CLASS LineParser
   PROTECTED:
   DATA text

   INLINE METHOD extract
      LOCAL i := At( Chr(13)+Chr(10), ::text )

      IF i == 0
         ::line := ::text
         ::text := ""
      ELSE
         ::line := SubStr( ::text, 1, i-1 )
         ::text := SubStr( ::text, i+2    )
      ENDIF

      RETURN self
   ENDMETHOD


   EXPORTED:
   DATA line  INIT  ""  READONLY

   METHOD init( cText ) INLINE ( ::text := cText, ::extract(), self )
   METHOD eof           INLINE ( Len( ::text + ::line ) == 0 )

   INLINE METHOD nextLine
      ::extract()
      RETURN ::line
   ENDMETHOD

ENDCLASS
```

LOCAL

Declares and optionally initializes a local memory variable.

Syntax

```
LOCAL <varName> [:= <xValue> ]
```

Arguments

`LOCAL <varName>`

<varName> is the symbolic name of the local variable to declare.

`<xValue>`

<xValue> is an optional value to assign to the LOCAL variable after being declared. To assign a value, the inline assignment operator (:=) must be used. The simple assignment operator (=) cannot be used.

Description

The LOCAL statement declares a lexical memory variable that has LOCAL scope. Local variables are resolved by the compiler, i.e. their symbolic name cannot be retrieved during runtime. This makes access to LOCAL variables much faster than to dynamic memory variables of PRIVATE or PUBLIC scope, whose symbolic variable names exist at runtime.

The names of LOCAL variables cannot be included in macro-expressions since they cannot be resolved by the macro operator (&). This operator requires the symbolic name of a variable to exist at runtime.

The visibility and lifetime of LOCAL variables is restricted to the function, procedure or method that declares a LOCAL variable. Unlike PRIVATE or PUBLIC variables, LOCAL variables cannot be seen in a subroutine. To make the value of a LOCAL variable visible in a subroutine, it must be passed as a parameter to the subroutine.

When a routine executes the RETURN statement, all LOCAL variables declared in that routine are discarded and their values become subject to garbage collection.

The lines in PRG source code preceding the LOCAL statement may not call executable code. They can only contain declaration statements, i.e. only the FUNCTION, METHOD, PROCEDURE statements, and the FIELD, MEMVAR, PARAMETERS, PRIVATE, PUBLIC, STATIC variable declarations are allowed to precede the LOCAL statement.

It is possible to initialize a local variable already in the LOCAL statement. To accomplish this, the inline-assignment operator must be used. The value of any valid expression can be assigned. This includes literal values and the return values of functions.

Note: If a PRIVATE or PUBLIC variable exists that has the same symbolic name as a LOCAL variable, only the LOCAL variable is visible. PRIVATE or PUBLIC variables with the same name become visible again, when the LOCAL variable gets out of scope, i.e. when the routine that declareS the LOCAL variable returns or calls a subroutine.

Info

See also:	FUNCTION, GLOBAL, PARAMETERS, PRIVATE, PROCEDURE, PUBLIC, STATIC
Category:	Declaration, Statements

Example

```
// The example demonstrates the visibility of LOCAL and PRIVATE
// memory variables
```

```
PROCEDURE Main
   PRIVATE myVar := "Private Var"

   ? Procname(), myVar

   Test1()
RETURN


PROCEDURE Test1
   LOCAL myVar := "Local Var"

   // PRIVATE myVar is unvisible
   ? Procname(), myVar

   Test2()
RETURN


PROCEDURE Test2
   // PRIVATE myVar is visible
   ? Procname(), myVar
RETURN
```

MEMVAR

Declares PRIVATE or PUBLIC variables.

Syntax

```
MEMVAR <varName,...>
```

Arguments

`MEMVAR <varName,...>`

This is a comma separated list of symbolic names identifying field variables.

Description

The MEMVAR statement declares symbolic names of dynamic memory variables of PRIVATE or PUBLIC scope. This instructs the compiler to resolve unaliased variable names to memory variables, not field. All variables listed in *<varName,...>* that appear in program code without alias name and alias operator, are treated as if they are listed with the M-> alias name.

The scope of the MEMVAR statement depends on the place of declaration:

1. When the MEMVAR declaration appears at the top of a PRG file before any other executable statement, the declaration has file wide scope, i.e. it is valid throughout the entire PRG file.
2. When the MEMVAR declaration follows a FUNCTION, METHOD or PROCEDURE declaration, the variable is treated as memory variable only in the routine that declares the MEMVAR.

The lines in PRG source code preceding the MEMVAR declaration may not call executable code. They can only contain declaration statements, i.e. only the FUNCTION, METHOD, PROCEDURE statements, and the FIELD, LOCAL, PARAMETERS, or STATIC variable declarations are allowed to precede the MEMVAR statement.

Info

See also: FIELD, GLOBAL, LOCAL, PRIVATE, PUBLIC, STATIC
Category: Declaration, Statements

Example

```
// The example demonstrates the difference of a declared and
// undeclared memory variable of PRIVATE scope.

   MEMVAR myVar

   PROCEDURE Main
      x := "myVar"              // compiler warning
      myVar := "Hello World"    // no compiler warning

      ? x
      ? &x
   RETURN
```

MESSAGE

Declares a message name for a method.

Syntax

```
MESSAGE <MessageName> METHOD <MethodName>
MESSAGE <MessageName> IN     <SuperClass>
MESSAGE <MessageName> IS     <MethodName> IN <SuperClass>

MESSAGE <MessageName> TO     <ContainedObject>
MESSAGE <MessageName> IS     <MethodName> TO <ContainedObject>
```

Arguments

`<MessageName>`

This is the symbolic name of a message to declare for a class.

`METHOD <MethodName>`

This is the symbolic name of the method to execute when an object receives *<MessageName>* as a message.

`IN <SuperClass>`

Optionally, the name of the super class can be specified to which the message should be sent. This requires the class to inherit one or more other classes.

`IS <MethodName>`

When a message should be directed to a super class, the method to invoke in the super class can be specified with *<MethodName>*.

`TO <ContainedObject>`

This is the symbolic name of an instance variable holding the object that receives the message.

Description

The MESSAGE statement can only be used within the class declaration between CLASS and ENDCLASS. It declares a message that invokes a method of a different name in the declared class or its super class.

MESSAGE is used to resolve ambiguities when a class inherits from one or more super classes which have the same method names. The statement can also be used to invoke a method via an alternative name.

Info

See also:	ACCESS, ASSIGN, CLASS, DATA, DELEGATE, METHOD (declaration), VAR
Category:	Class declaration, Declaration, xHarbour extensions
Header:	hbclass.ch
Source:	vm\classes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// In the example a Subclass uses the method implementation
// of a Super class but redefines the message to send to
// an object. Message :print() invokes method :show().

   #include "Hbclass.ch"
```

```
PROCEDURE Main
   LOCAL obj := TextFile():new( "Text.prg" )

   obj:open()
   obj:print()
   obj:close()
RETURN


CLASS Text
   EXPORTED:
   DATA string

   METHOD init(c)  INLINE ( ::string := c, self )
   METHOD show     INLINE QOut( ::string )
ENDCLASS



CLASS TextFile FROM Text
   EXPORTED:
   DATA fileName

   METHOD init
   METHOD open
   METHOD close

   MESSAGE print IS show IN Text
ENDCLASS

METHOD init( cFile, cText ) CLASS TextFile
   ::fileName := cFile

   IF Valtype( cText ) == "C"
      ::text:init( cText )
   ELSE
      ::text:init( Memoread(::fileName ) )
   ENDIF
RETURN self

METHOD open CLASS TextFile
   SET ALTERNATE ON
   SET ALTERNATE TO (::fileName)
RETURN

METHOD close CLASS TextFile
   SET ALTERNATE TO
   SET ALTERNATE OFF
RETURN
```

METHOD (declaration)

Declares the symbolic name of a method.

Syntax

```
METHOD <MethodName>[(<params,...>] )] [INLINE <expression>] [SYNC]
```

Arguments

<MethodName>

This is the symbolic name of a single method to declare. It must begin with a letter or underscore followed by digits, letters or underscores. A symbolic name can contain up to 63 characters.

(<params,...>)

A declaration of formal parameters enclosed in parentheses is required when the method is declared as INLINE. If INLINE is not used, the method can be declared without parameters.

INLINE <expression>

This is the expression which is executed when an INLINE method is invoked. *<expression>* must be one line of code. The code cannot contain commands but only function and method calls.

SYNC

The SYNC option declares a "synchronized" method. This is only required in multi-threaded applications when a method must be protected against simultaneous access by multiple-threads. Methods declared with the SYNC attribute can be executed only by one thread at a time. Refer to function StartThread() for more information on multiple threads.

Description

When the METHOD statement occurs in the class declaration between CLASS and ENDCLASS, it declares the symbolic name of a method. This name must be sent to an object to invoke the method.

The program code for a declared method must be implemented with a separate METHOD declaration that follows the ENDCLASS statement. If, however, the method code can be programmed in one line, method declaration and implementation can be coded within the class declaration using the INLINE option. INLINE methods are not implemented separately from the class declaration.

When a method is invoked, the object executing the method is accessible in the code with the reserved variable name *self*. This can be abbreviated with two colons (::).

Info

See also: ACCESS, ASSIGN, CLASS, CLASSMETHOD, DATA, DELEGATE, EXPORTED:, HIDDEN:, INLINE METHOD, MESSAGE, METHOD (implementation), METHOD...OPERATOR, METHOD...VIRTUAL, PROTECTED:, StartThread(), VAR
Category: Class declaration, Declaration, xHarbour extensions
Header: hbclass.ch
Source: vm\classes.c
LIB: xhb.lib
DLL: xhbdll.dll

Examples

METHOD declaration

```
// The example declares a simple class with a regular and an
// INLINE method. Note that the INLINE method consists of two
```

```
// expressions. The first is an assignment and the second is the
// return value of the method. Both expressions are comma separated
// and enclosed in parentheses.

   #include "Hbclass.ch"

   PROCEDURE Main
      LOCAL obj

      CLS
      obj := Test():new( "xharbour" )

      obj:print()
   RETURN


   CLASS Test
      HIDDEN:
      VAR name

      EXPORTED:
      METHOD init( cName ) INLINE ( ::name := cName, self )
      METHOD print
   ENDCLASS

   METHOD print CLASS Test
      ? ::name
   RETURN self
```

SYNC method

```
// The example implements a Queue class that adds a value to a queue
// with method :put(). Method :get() retrieves the last value added to
// the queue. The queue itself is an array. Both methods are declared with
// the SYNC attribute to protect the queue array from simultaneous access
// by multiple threads.
//
// The Main thread puts values into the queue, while four other threads
// retrieve values from the queue.

   #include "hbclass.ch"

   PROCEDURE Main
      LOCAL i, aT[4], oQueue := Queue():new()
      CLS

      FOR i:=1 TO 4
         aT[i] := StartThread( @Test(), oQueue )
      NEXT

      FOR i:=1 TO 4000                 // Write values
         oQueue:put(i)                 // to queue
      NEXT

      WaitForThreads()                 // Wait until all threads
   RETURN                              // have terminated


   ** This code runs simultaneously in 4 threads
   PROCEDURE Test( oQueue )
      LOCAL i, nRow, nCol
```

```
   nRow := GetThreadID() + 10
   nCol := 0
                                  // Read values
   FOR i:=1 TO 1000               // from queue
      DispOutAt( nRow, nCol+20, oQueue:nCount )

      DispOutAt( nRow, nCol+40, oQueue:get()  )
   NEXT
RETURN


** class with SYNC methods protecting the :aQueue array
CLASS Queue
   PROTECTED:
   VAR aQueue INIT {}

   EXPORTED:
   VAR nCount INIT 0 READONLY

   METHOD put() SYNC
   METHOD get() SYNC
ENDCLASS


METHOD put( xValue )  CLASS Queue
   AAdd( ::aQueue, xValue )
   ::nCount ++
RETURN self


METHOD get() CLASS Queue
   LOCAL xReturn

   IF ::nCount > 0
      ::nCount --
      xReturn := ::aQueue[1]

      ADel ( ::aQueue, 1 )
      ASize( ::aQueue, ::nCount )
   ENDIF
RETURN xReturn
```

METHOD (implementation)

Declares the implementation code of a method.

Syntax

```
METHOD <MethodName>[( <params,...> )] CLASS <ClassName>

  <...program code...>

RETURN <retVal>
```

Arguments

`<MethodName>`

This is the symbolic name of the method to implement. It must begin with a letter or underscore followed by digits, letters or underscores. The symbolic name can contain up to 63 characters.

`(<params,...>)`

If a method accepts parameters, they must be declared as a comma separated list within parameters.

`CLASS <ClassName>`

This is the symbolic name of the class where the method is declared.

`RETURN <retVal>`

The RETURN statement terminates a method and branches control back to the calling routine, returning the value *<retVal>* to it.

Description

When the METHOD statement occurs outside the class declaration after ENDCLASS, it declares the implementation part of a method. A method is implemented like a FUNCTION, including formal parameter list, if required.

There is one important difference to functions: a special variable *self* exists in all methods. *self* is the object executing the method. Via *self* instance variables of an object can be accessed and other methods of the class can be invoked.

The name of the class where the method is declared must be specified with the method implementation. Without the class name, it would not be possible to implement multiple classes having the same method names in one PRG module.

When a method does not require a particular return value, it is quite common to return the *self* object. This way, method calls can be "chained" in one line of code.

Info

See also:	CLASS, FUNCTION, INLINE METHOD, METHOD (declaration), PROCEDURE
Category:	Class declaration, Declaration, xHarbour extensions
Header:	hbclass.ch
Source:	vm\classes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example implements a class with nine methods that can be
// used for database access which is independent of the current
// workarea.
```

```
#include "HbClass.ch"

PROCEDURE Main
   LOCAL oWorkArea
                         // method "chaining"
   oWorkArea := WorkArea():new( "Customer.dbf" ):open()

   ? Alias()             // result: Customer

   SELECT 0
   ? Alias()             // result: empty string ("")

                         // method "chaining"
   oWorkArea:goBottom():skip(-5)

   DO WHILE .NOT. oWorkArea:eof()
      ? Customer->LastName
      oWorkArea:skip()
   ENDDO

   oWorkArea:close()
   ? Alias()             // result: empty string ("")

   oWorkArea:open()
   ? Alias()             // result: Customer

   oWorkArea:close()
RETURN

CLASS WorkArea
   PROTECTED:
   VAR fileName

   EXPORTED:
   VAR area  READONLY

   METHOD init
   METHOD open
   METHOD close
   METHOD bof
   METHOD eof
   METHOD found
   METHOD skip
   METHOD goTop
   METHOD goBottom
ENDCLASS

METHOD init( cFileName ) CLASS WorkArea
   ::fileName := cFileName
   ::area     := 0
RETURN self

METHOD open() CLASS WorkArea
   IF ::area == 0
      USE ( ::fileName ) NEW SHARED
      ::area := Select()
   ELSE
      DbSelectArea( ::area )
   ENDIF
RETURN self
```

```
METHOD close() CLASS WorkArea
   IF ::area <> 0
      (::area)->(DbCloseArea())
      ::area := 0
   ENDIF
RETURN self

METHOD bof CLASS WorkArea
RETURN IIf( ::area == 0, .T., (::area)->(Bof()) )

METHOD eof CLASS WorkArea
RETURN IIf( ::area == 0, .T., (::area)->(Eof()) )

METHOD found CLASS WorkArea
RETURN IIf( ::area == 0, .F., (::area)->(Found()) )

METHOD skip( nSkip ) CLASS WorkArea
   IF PCount() == 0
      nSkip := 1
   ENDIF

   IF ::area > 0
     (::area)->(DbSkip(nSkip))
   ENDIF
RETURN self

METHOD goTop CLASS WorkArea
   IF ::area > 0
     (::area)->(DbGotop())
   ENDIF
RETURN self

METHOD goBottom CLASS WorkArea
   IF ::area > 0
     (::area)->(DbGoBottom())
   ENDIF
RETURN self
```

METHOD...OPERATOR

Declares a method to be executed with an operator.

Syntax

```
METHOD <MethodName> OPERATOR <xOperator>
```

Arguments

`<MethodName>`

This is the symbolic name of the method to declare. It must begin with a letter or underscore followed by digits, letters or underscores. The symbolic name can contain up to 63 characters.

`OPERATOR <xOperator>`

This is the operator which invokes the method.

Description

The OPERATOR option of the METHOD statement defines an operator that can be used with an object. When the object is an operand of *<xOperator>*, the method *<MethodName>* is invoked. Unary operators invoke the method without an argument while binary operators pass the second operand as argument to the method.

Note: only regular operators can be used for *<xOperator>*. Special operators are not supported. When *<xOperator>* is a binary operator, the object must be the left operand for invoking the method *<MethodName>*.

Info

See also:	CLASS, EXTEND CLASS...WITH METHOD, METHOD (implementation), OPERATOR
Category:	Class declaration, Declaration, xHarbour extensions
Header:	hbclass.ch
Source:	vm\classes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example demonstrates how basic numeric operations can
// be implemented with objects. The example support two unary
// and two binary operators.

   #include "Hbclass.ch"

   PROCEDURE Main
      LOCAL oNumA := Number():new( 10 )
      LOCAL oNumB := NUmber():new( 2 )

      ? oNumA + 100           // result: 110

      ? oNumA + oNumB ++      // result: 13

      ? oNumA - oNumB         // result: 7

      ? -- oNumA              // result: 9
   RETURN
```

```
CLASS Number
   PROTECTED:
   DATA    nNumber    INIT 0

   EXPORTED:
   METHOD  init(n)    INLINE (IIf(Valtype(n)=="N",::nNumber:=n,) , self)

   METHOD  add        OPERATOR +
   METHOD  subtract   OPERATOR -

   METHOD  increment  OPERATOR ++
   METHOD  decrement  OPERATOR --

   ACCESS  value      INLINE ::nNumber
   ASSIGN  value(n)   INLINE ::nNumber := n
ENDCLASS

METHOD add(x) CLASS Number
   IF HB_IsObject(x)
      RETURN ::nNumber + x:value
   ENDIF
RETURN ::nNumber + x

METHOD subtract(x) CLASS Number
   IF HB_IsObject(x)
      RETURN ::nNumber - x:value
   ENDIF
RETURN ::nNumber - x

METHOD increment CLASS Number
RETURN ++ ::nNumber

METHOD decrement CLASS Number
RETURN -- ::nNumber
```

METHOD...VIRTUAL

Declares a method as virtual.

Syntax

```
METHOD <MethodName> VIRTUAL|DEFERRED
```

Arguments

<MethodName>

This is the symbolic name of the method to declare. It must begin with a letter or underscore followed by digits, letters or underscores. The symbolic name can contain up to 63 characters.

DEFERRED

This is a synonym for VIRTUAL.

Description

The METHOD...VIRTUAL statement can only be used in the class declaration between CLASS and ENDCLASS. It declares the symbolic name of a virtual method.

A virtual method is a method that is declared in a class but not implemented. The implementation of such method is deferred to a sub-class. Consequently, virtual methods are only used with inheritance where sub-classes are derived from super classes. By means of virual methods a programmer can define the "message profile" within a class hierarchy at a very early point of the development cycle. An object having virtual methods understands the corresponding message but does not execute code.

Info

See also:	CLASS, DELEGATE
Category:	Class declaration, Declaration, xHarbour extensions
Header:	hbclass.ch
Source:	vm\classes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

OPERATOR

Overloads an operator and declares a method to be invoked for it.

Syntax

```
OPERATOR <cOperator> [ARG <operand>] INLINE <expression>
```

Arguments

`<cOperator>`

This is the operator which executes *<expression>*. The operator must be enclosed in quotes.

`ARG <operand>`

When a binary operator is overloaded, *<operand>* is the name of a variable which receives the value of the second operand of the operator. Unary operators do not require *<operand>* be declared.

`INLINE <expression>`

This is the expression which is executed when the object is an operand of *<operator>*. *<expression>* must be one line of code. The code cannot contain commands but only function and method calls.

Description

The OPERATOR statement can only be used in the class declaration between CLASS and ENDCLASS. It is the inline version of the METHOD...OPERATOR statement.

Note: only regular operators can be used for *<cOperator>*. Special operators are not supported. when *<cOperator>* is a binary operator, the object must be the left operand for *<expression>* being executed.

Info

See also:	METHOD...OPERATOR
Category:	Class declaration, Declaration, xHarbour extensions
Header:	hbclass.ch
Source:	vm\classes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example declares a String class and displays the
// results of operations with a String object.

   #include "HbClass.ch"

   PROCEDURE Main
      LOCAL obj := String():New( "Hello  " )

      ? obj =  "Hello"            // result: .T.
      ? obj == "Hello"            // result: .F.
      ? obj != "Hello"            // result: .F.
      ? obj <> "Hello"            // result: .F.
      ? obj #  "Hello"            // result: .F.
      ? obj $  "Hello"            // result: .F.

      ? obj <  "hello"            // result: .T.
      ? obj <= "hello"            // result: .T.
```

```
   ? obj >  "hello"            // result: .F.
   ? obj >= "hello"            // result: .F.

   ? obj +  "World"            // result: Hello  World
   ? obj -  "World"            // result: HelloWorld
RETURN


CLASS String
   EXPORTED:
   DATA value       INIT ""

   METHOD init( c ) INLINE ( IIf(valtype(c)=="C", ::value := c, ), self )

   OPERATOR "="   ARG c INLINE ::value =  c
   OPERATOR "=="  ARG c INLINE ::value == c
   OPERATOR "!="  ARG c INLINE ::value != c
   OPERATOR "<"   ARG c INLINE ::value <  c
   OPERATOR "<="  ARG c INLINE ::value <= c
   OPERATOR ">"   ARG c INLINE ::value >  c
   OPERATOR ">="  ARG c INLINE ::value >= c
   OPERATOR "+"   ARG c INLINE ::value +  c
   OPERATOR "-"   ARG c INLINE ::value -  c
   OPERATOR "$"   ARG c INLINE ::value $  c
ENDCLASS
```

OVERRIDE METHOD

Replaces a method in an existing class.

Syntax

```
OVERRIDE METHOD <MethodName> ;
       IN CLASS <ClassName> WITH <FunctionName>
```

Arguments

`<MethodName>`

This is the symbolic name of the method to replace.

`IN CLASS <ClassName>`

This is the symbolic name of the class in which the method is replaced.

`WITH <FunctionName>`

This is the symbolic name of the function implementing the code for the replaced method.

Description

The OVERRIDE METHOD statement replaces the implementation of a method in an existing class. When an object receives the message *<MethodName>*, the code implemented in *<FunctionName>* is executed and the previous implementation is no longer accessible. *<FunctionName>* is usually implemented as a STATIC FUNCTION where the *self* object is retrieved with HB_QSelf().

OVERRIDE METHOD can be used to replace the method implementation in a higher level of a class tree. This is especially useful when the source code of the classes is not available. However, great care must be taken in order not to break existing class inheritance.

An alternative is the declaration of a sub-class using the FROM option in the class declaration and implement the new method in the sub-class.

Info

See also:	ASSOCIATE CLASS, DELEGATE, EXTEND CLASS...WITH METHOD, METHOD (implementation), OPERATOR
Category:	Class declaration, Declaration, xHarbour extensions
Header:	hbclass.ch
Source:	vm\classes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

Example

```
// The example outlines how a method can be replaced in existing
// classes and how the self object is retrieved in the implementation
// of a replacement method.

   #include "hbclass.ch"

   PROCEDURE Main()
      LOCAL cVar := "Hello", GetList := {}

      CLS

      OVERRIDE METHOD Display IN CLASS GET WITH MyGetDisplay
      OVERRIDE METHOD AsString IN CLASS ARRAY WITH MyArrayString
```

```
   @ 10,10 GET cVar

   ? GetList:AsString()
RETURN

STATIC FUNCTION MyGetDisplay
   LOCAl Self := HB_QSelf()
   Alert( ::VarGet() )
RETURN Self

STATIC FUNCTION MyArrayString()
   LOCAl Self := HB_QSelf()
RETURN "{ ... }"
```

PARAMETERS

Declares PRIVATE function parameters.

Syntax

```
PARAMETERS <params,...>
```

Arguments

`<params,...>`

This is a comma separated list of symbolic names for the parameters to declare.

Description

The PARAMETERS statement declares the formal list of parameters for functions and procedures. The parameters are created as PRIVATE variables when the function or procedure is invoked.

Note: It is recommended to list formal parameters within parentheses in the FUNCTION or PROCEDURE declaration. This results in the formal parameters being created as LOCAL variables. Access to LOCAL variables is faster than to PRIVATE variables.

Info

See also: FUNCTION, LOCAL, PCount(), PRIVATE, PROCEDURE, PUBLIC, STATIC
Category: Declaration, Statements

Example

```
// The example demonstrates two possible forms of declaring
// parameters. The declaration shown with Sum2() is recommended.

   PROCEDURE Main
      ? Sum1( 10, 20 )
      ? Sum2( 100, 200 )
   RETURN

   FUNCTION Sum1
   PARAMETERS n1, n2
   RETURN n1 + n2

   FUNCTION Sum2( n1, n2 )
   RETURN n1 + n2
```

pragma pack()

Defines the byte alignment for the next structure declaration.

Syntax

```
pragma pack( [<nAlign>] )
```

Arguments

<nAlign>

This is a numeric value specifying the byte alignment for the next typedef struct declaration. If *<nAlign>* is omitted, the byte alignment is set to 8 bytes.

Description

xHarbour supports C structures with its typedef struct statement. Structure data is held in a contiguous block of memory which stores the values of all structure members. For efficient memory management, the values of structure members are aligned at an *<nAlign>* byte boundary. The default alignment is an 8 byte boundary.

Windows API structures

All structures used with 32-bit Windows API functions must be aligned on a 4 byte boundary. That is, *pragma pack(4)* must be called before *typedef struct*. Otherwise, structure data is not correctly processed.

Info

See also: C Structure class, (struct), typedef struct
Category: C Structure support, xHarbour extensions
Header: cstruct.ch
LIB: xhb.lib
DLL: xhbdll.dll

PRIVATE

Creates and optionally initializes a PRIVATE memory variable.

Syntax

```
PRIVATE <varName> [:= <xValue> ]
```

Arguments

`PRIVATE <varName>`

<varName> is the symbolic name of the PRIVATE variable to createe.

`<xValue>`

<xValue> is an optional value to assign to the PRIVATE variable after being created. To assign a value, the inline assignment operator (:=) must be used. The simple assignment operator (=) cannot be used.

Description

The PRIVATE statement creates a dynamic memory variable that has PRIVATE scope. Private variables are resolved at runtime, i.e. their symbolic name exist while a program is being executed. This makes PRIVATE variables accessible for the Macro operator (&) at the expense of execution speed. The access to LOCAL variables is faster than to PRIVATE variables.

Private variables are visible from the time of creation until the declaring routine returns, or until they are explicitely released. Visibility of PRIVATEs extends to all subroutines called after variable creation. A private variable can become temporarily hidden when a subroutine declares a variable having the same symbolic name. This can lead to subtle programming error, so that the use of PRIVATEs is discouraged unless there is good reason to use a PRIVATE variable. This is the case, for example, when it must be resolved within a macro expression.

It is possible to initialize a private variable already in the PRIVATE statement. To accomplish this, the inline-assignment operator must be used. The value of any valid expression can be assigned. This includes literal values and the return values of functions.

Note: PIVATE variables are *declared* with the MEMVAR statement and *created* with the PRIVATE statement. That means, PRIVATE is an executable statement that must follow all declaration statements.

All undeclared variables that are assigned a value with the inline assignemnt operator, are created as PRIVATE variables.

Info

See also: FIELD, GLOBAL, LOCAL, MEMVAR, PUBLIC, RELEASE, STATIC
Category: Declaration, Statements

Example

```
// These examples show different initializations of PRIVATE variables.

   PROCDEURE Main

      PRIVATE nNumber := 5        // initialized as numeric
      PRIVATE aUsers[10]          // array with 10 empty elements
      PRIVATE nAge, cName         // uninitialized

      dDate := Date()             // undeclared but initialized
   RETURN
```

PROCEDURE

Declares a procedure along with its formal parameters.

Syntax

```
[STATIC] [UTILITY] PROCEDURE <procName>( [<params,...>] )
           [FIELD <fieldName,...> [IN <aliasName>]]
          [MEMVAR <var_Dynamic,...>]
           [LOCAL <var_Local>  [:= <expression>] ,... ]
          [STATIC <var_Static> [:= <expression>] ,... ]

            <Statements>

           RETURN
```

Arguments

`PROCEDURE <procName>( [<params,...>] )`

This is the symbolic name of the declared procedure. It must begin with a letter or underscore followed by digits, letters or underscores. The symbolic name can contain up to 63 characters.

Optionally, the names of parameters *<params,...>* accepted by the procedure can be specified as a comma separated list. These parameters have LOCAL scope within the procedure.

When the procedure is declared as STATIC PROCEDURE, it is only visible within the PRG file that contains the procedure declaration and cannot be invoked from elsewhere.

`UTILITY`

The UTILITY attribute indicates that the declared procedure should not be used as startup code even if it is the first declared procedure in the PRG source file.

`FIELD <fieldName>`

An optional list of field variables to use within the PROCEDURE can be declared with the FIELD statement.

`MEMVAR <var_Dynamic>`

If dynamic memory variables, i.e. PRIVATE or PUBLIC variables, are used in the procedure, they are declared with the MEMVAR statement.

`LOCAL <var_Local> [:= <expression>]`

Local variables are declared and optionally initialized using the LOCAL statement.

`STATIC <var_Static> [:= <expression>]`

Static variables are declared and optionally initialized using the STATIC statement.

`RETURN`

The RETURN statement terminates a procedure and branches control back to the calling routine. A procedure has no return value.

Description

The PROCEDURE statement declares a procedure along with an optional list of parameters accepted by the function. Statements programmed in the procedure body form a self-contained part of a program that is executed when a procedure is called. Thus, tasks of a program can be split into several procedures, each of which performs a sub-task when invoked.

The body of a procedure ends with the next FUNCTION, PROCEDURE or CLASS declaration, or at the end of file, which implies that procedure declarations cannot be nested.

The execution of a procedure ends when a RETURN statement is encountered in the procedure body. A procedure does not return a value to the calling routine, only functions have a return value. The RETURN value is the only difference between functions and procedures.

When a procedure is declared with the STATIC modifier, its visibility is restricted to the PRG file that contains the STATIC PROCEDURE declaration. The names of STATIC procedures are resolved by the compiler and do not exist at runtime of a program. The names of non-STATIC procedures, also referred to as public procedures, are resolved by the linker and do exist at runtime. Thus, public procedures can be accessed by the Macro operator (&) while STATIC procedures cannot.

It is possible to declare STATIC procedures with the same symbolic name in different PRG files. A name conflict to a public procedure with the same name declared in another PRG file does not arise. However, the symbolic names of public functions, procedures or classes must always be unique.

When a procedure is invoked with values being passed to it, they are assigned to the formal parameters declared with *<params,...>*. All variables declared in this list are LOCAL variables and their visibility is restricted to the statements programmed in the procedure body.

The number of values passed to a procedure does not need to match the number of parameters declared. When fewer values are passed, the corresponding parameters are initialized with NIL. When more values are passed, the additional values are not asssigned to parameters but can be retrieved using function HB_AParams().

Info

See also: FIELD, FUNCTION, HB_AParams(), LOCAL, MEMVAR, METHOD (declaration), PARAMETERS, PCount(), PRIVATE, PUBLIC, RETURN, STATIC
Category: Declaration, Statements

Example

```
// The PROCEDURE statement is used in may other examples of
// this documentation. See there.
```

PROTECTED:

Declares the PROTECTED attribute for a group of member variables and methods.

Syntax

```
PROTECTED:
```

Description

Protected member variables and methods are accessible in the class(es) declared in the PRG module and all subclasses. Subclasses are all classes that inherit directly or indirectly from the declared class. A protected member is accessible within the context of a METHOD implemented in the declaring PRG module or in the PRG module of a subclass. Protected members are not accessible within the context of FUNCTION and PROCEDURE.

Info

See also:	CLASS, DATA, EXPORTED:, HIDDEN:, METHOD (declaration)
Category:	Class declaration, Declaration, xHarbour extensions
Header:	hbclass.ch
Source:	vm\classes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

PUBLIC

Creates and optionally initializes a PUBLIC memory variable.

Syntax

```
PUBLIC <varName> [:= <xValue> ]
```

Arguments

`PUBLIC <varName>`

<varName> is the symbolic name of the PUBLIC variable to createe.

`<xValue>`

<xValue> is an optional value to assign to the PUBLIC variable after being created. To assign a value, the inline assignment operator (:=) must be used. The simple assignment operator (=) cannot be used.

When a PUBLIC variable is not assigned a value upon creation, it is initialized with .F. (false). This is different to all other variable types, which are initialized with NIL, by default.

Description

The PUBLIC statement creates a dynamic memory variable that has PUBLIC scope. Public variables are resolved at runtime, i.e. their symbolic name exist while a program is being executed. This makes PUBLIC variables accessible for the Macro operator (&) at the expense of execution speed. The access to GLOBAL variables is faster than to PUBLIC variables.

Public variables are visible from the time of creation until they are explicitely released. Visibility of PUBLICs extends to the entire program, even if a PUBLIC variable is created in a sub-routine which has returned. A public variable can become temporarily hidden when a subroutine declares a variable having the same symbolic name.

It is possible to initialize a public variable already in the PUBLIC statement. To accomplish this, the inline-assignment operator must be used. The value of any valid expression can be assigned. This includes literal values and the return values of functions.

Note: PUBLIC variables are *declared* with the MEMVAR statement and created with the PUBLIC statement. That means, PUBLIC is an executable statement that must follow all declaration statements.

Info

See also: FIELD, GLOBAL, LOCAL, MEMVAR, PRIVATE, RELEASE, STATIC
Category: Declaration, Statements

Example

```
// This example creates PUBLIC variables in a sub-routine called from
// the Main routine

   PROCEDURE Main
      ? Type( "dataPath" )      // result: U
      ? Type( "tempPath" )      // result: U

      MakeConfig()

      ? dataPath                // result: C:\xhb\apps\data
      ? tempPath                // result: C:\xhb\apps\temp
   RETURN

   PROCEDURE MakeConfig()
```

```
   PUBLIC dataPath := "C:\xhb\apps\data"
   PUBLIC tempPath := "C:\xhb\apps\temp"
RETURN
```

REQUEST

Declares the symbolic name of an external function or procedure for the linker.

Syntax

```
REQUEST <name1> [,<nameN>]
```

Arguments

```
REQUEST <name>
```

This is the symbolic name of a function or procedure to declare for the linker. When multiple names are declared, they must be separated with commas.

Description

The REQUEST statement declares a symbolic name of a function or procedure for the linker. This is usually required when there is no direct call of a function or procedure in PRG code, for example when a function is only called within a macro-expression using the macro-operator. By requesting the symbolic name of a function, the linker is forced to link the corresponding function to the executable file.

Info

See also: #include, EXTERNAL
Category: Declaration

Example

```
// This example demonstrates a possible header file with some REQUEST
// statements. The file can the be #included in PRG files that may
// need the requested functions.

   ** File: Requests.ch
   REQUEST _ADS
   REQUEST HBObject
   // EOF
```

RETURN

Branches program control to the calling routine.

Syntax

```
RETURN [<retVal>]
```

Arguments

<retVal>

<retVal> is an expression of arbitrary data type that is passed from methods or functions to the calling routine. Procedures do not have a return value.

Description

The RETURN statement branches program control to the calling routine, eventually passing a value back. Before the called routine returns, all LOCAL and PRIVATE variables declared in this routine are discarded.

When the RETURN statement is executed by the main, or root, routine of an application, control goes back to the operating system.

Note: The RETURN statement is not the end of a FUNCTION or PROCEDURE declaration, it terminates execution of a called routine. It is possible to have multiple RETURN statements in the body of a FUNCTION or PROCEDURE declaration.

Info

See also: BEGIN SEQUENCE, ErrorLevel(), FUNCTION, LOCAL, PRIVATE, PROCEDURE, PUBLIC, QUIT

Category: Control structures, Statements

STATIC

Declares and optionally initializes a STATIC memory variable.

Syntax

```
STATIC <varName> [:= <xValue> ]
```

Arguments

`<varName>`

<varName> is the symbolic name of the static variable to declare.

`<xValue>`

<xValue> is an optional value to assign to the STATIC variable after being declared. To assign a value, the inline assignment operator (:=) must be used. The simple assignment operator (=) cannot be used. Only literal values are allowed for *<xValue>* when declaring STATIC variables.

Description

The STATIC statement declares a memory variable that has STATIC scope. Static variables are resolved by the compiler, i.e. their symbolic name cannot be retrieved during runtime. This makes acces to STATIC variables much faster than to memory variables of PRIVATE or PUBLIC scope, whose symbolic variable names exist at runtime.

The names of STATIC variables cannot be included in macro-expressions since they cannot be resolved by the macro operator (&). This operator requires the symbolic name of a variable to exist at runtime.

The lifetime of STATIC variables is identical with the lifetime of a program. Unlike LOCAL variables, whose values are discarded when the declaring routine executes the RETURN statement, STATIC variables keep their values during the entire execution cycle of a program. The value assigned to a STATIC variable is stored in this variable until it gets assigned a new value.

The visibility of STATIC variables depends on the place of declaration:

1. When the STATIC declaration appears at the top of a PRG file before any other executable statement, the STATIC variable has file wide scope, i.e. it can be seen by all routines programmed in that PRG file.
2. When the STATIC declaration follows a FUNCTION, METHOD or PROCEDURE declaration, the variable can be seen only in the routine that declares the STATIC variable.
3. STATIC variables cannot be seen outside the PRG file that contains the STATIC declaration.

When a routine executes the RETURN statement, all STATIC variables declared in that routine keep their values.

The lines in PRG source code preceding the STATIC statement may not call executable code. They can only contain declaration statements, i.e. only the FUNCTION, METHOD, PROCEDURE statements, and the FIELD, LOCAL, MEMVAR or PARAMETERS variable declarations are allowed to precede the STATIC statement.

It is possible to initialize a static variable already in the STATIC statement. To accomplish this, the inline-assignment operator must be used. The values assigned to STATIC variables must be programmed as literal values, i.e. the compiler must be able to resolve values assigned to STATIC variables. It is not possible, for example, to initialize a STATIC variable with the RETURN value of a function, since this can only be resolved at runtime of a program.

Note: symbolic name as a STATIC variable, only the STATIC variable is visible. PRIVATE or PUBLIC variables with the same name become visible again, when the STATIC variable gets out of scope, i.e. when the routine that declares the STATIC variable returns or calls a subroutine.

Info

See also: FUNCTION, GLOBAL, LOCAL, PARAMETERS, PRIVATE, PROCEDURE, PUBLIC
Category: Declaration, Statements

Example

```
// The example uses one PRIVATE variable and two STATIC variables to
// demonstrate lifetime and visibility of STATIC variables.

   STATIC snFileWide := 100

   PROCEDURE Main
      PRIVATE myVar := 10

      ? ProcName(), snFileWide, myVar        // output: MAIN   100   10

      Test1()                                // output: TEST1  100    3
      Test2()                                // output: TEST2  110   12

      ? snFileWide                           // output: 120
      ? myVar                                // output: 12
   RETURN


   PROCEDURE Test1()
      STATIC myVar := 1

      ? ProcName(), snFileWide, myVar += 2   // output: TEST1  100    3

      snFileWide += 10
   RETURN


   PROCEDURE Test2()
      ? ProcName(), snFileWide, myVar += 2   // output: TEST2  110   12

      snFileWide += 10
   RETURN
```

SWITCH

Executes one or more blocks of statements.

Syntax

```
SWITCH <Expression>
  CASE <Constant1>
    <statements>
    [EXIT]
 [CASE <ConstantN>
    <statements>
    [EXIT] ]
 [DEFAULT
    <statements>]
END
```

Arguments

`SWITCH <Expression>`

The value of <*Expression*> must be of data type Character or Numeric. When it is of type Character, it must be a single character, not a character string. Numeric values must be integer values.

`CASE <Constant1> .. <ConstantN>`

<*Constant*> is a constant value of the same data type as <*expression*>. It must be a singlecharacter or an integer numeric value.

Description

The SWITCH statement compares a constant value against a series of constant values. It is similar to the DO CASE statement but outperforms it to a great extent due to the restrictions introduced with values permitted for comparison. As a general rule, only literal values in form of single characters or integer constants may follow the CASE clauses.

The SWITCH statement evaluates <*Expression*> and then searches for a first match between the resulting value and <*Constant*>. When a match is found, the statements following the corresponding CASE clause are executed down to the END statement. To suppress execution of statements of the next CASE clause, the EXIT statement must be explicitely used. This is a major difference to the DO CASE statement where subsequent CASE clauses are skipped once a first match is found.

If no initial match is found with the CASE clauses, the statements following DEFAULT are executed, if present.

Info

See also: BEGIN SEQUENCE, DO CASE, FOR, FOR EACH, IF, TRY...CATCH
Category: Control structures, Statements, xHarbour extensions

Examples

```
// The example demonstrates the SWITCH control structure with
// EXIT statement

   PROCEDURE Main
      LOCAL nMonth := Month( Date() )

      SWITCH nMonth
      CASE 1
      CASE 2
      CASE 3
```

```
         ? "First Quarter"
         EXIT
      CASE 4
      CASE 5
      CASE 6
         ? "Second Quarter"
         EXIT
      CASE 7
      CASE 8
      CASE 9
         ? "Third Quarter"
         EXIT
      DEFAULT
         ? "Fourth quarter"
      END
   RETURN

// The example demonstrates the SWITCH control structure without
// EXIT statement

   PROCEDURE Main
      LOCAL x := "D"

      SWITCH x
      CASE "A"
        ? "A"

      CASE "B"
        ? "B"

      CASE "D"
        ? "DEF"

      CASE "H"
        ? "HIJ"

      DEFAULT
        ? "Default"
      END
   RETURN

   ** Output:
   // DEF
   // HIJ
   // Default
```

TRY...CATCH

Declares a control structure for error handling.

Syntax

```
TRY
 <statements>
 [THROW( <oErrorObject> )]
 [CATCH [<thrownError>]
   <errorHandling>
 [FINALLY
  <guaranteed> ]
END
```

Arguments

`<statements>`

These are the regular programming statements to execute in the TRY...CATCH control structure.

`THROW <oErrorObject>`

When an unexpected situation is detected in *<statements>*, a new Error object can be created explicitly and passed to the Throw() function. If *<oErrorObject>* is omitted and an error is detected by the xHarbour runtime, the Error object is created automatically.

`CATCH <thrownError>`

This is an optionally declared variable that receives the error object passed to the Throw() function in the TRY section of the control structure.

The statements following the CATCH clause are used for error handling.

`FINALLY`

The finally section is guaranteed to be executed, no matter if an error was handled or not.

Description

Any code that might throw an exception is placed inside of the try block. If an exception is thrown, the catch block is entered and the program can perform the appropriate operation to recover or alert the user.

If FINALLY is specified, code within the FINALLY section is guranteed to be executed after the TRY section has been executed and the CATCH section is activated, unless the CATCH section throws an UNHANDLED Error. This means that the FINALLY section will be executed even if the CATCH section re-throws the error or attempts to RETURN. In such cases, the requested operation which forces out of the TRY section will be deferred until after the FINALLY section has been completed.

Important: although CATCH and FINALLY are both marked as optional, at least one of these options **must** be used within a TRY...END control block.

Info

See also: BEGIN SEQUENCE, Error(), ErrorBlock(), ErrorNew(), Throw()
Category: Control structures, Statements, xHarbour extensions

Example

```
// The example demonstrates program flow for a TRY..CATCH sequence

   PROCEDURE Main()
      LOCAL oErr
```

```
   TRY
     ? "Trying"
     ? "Throwing"
     Throw( ErrorNew( "Finalize Test", 0, 0, "Forced Error" ) )

   CATCH oErr
     ? "Caught:", oErr:Operation
     ? "Throwing to outer, should be deferred"
     Throw( oErr )
   FINALLY
      ? "Finalized"
   END

   ? "Oops, should have Re-Thrown, after the FINALLY."
RETURN
```

typedef struct

Declares a new structure in C syntax.

Syntax

```
typedef struct [<tagName>] { ;
    <C Data type1>  <MemberName1>  ;
   [<C Data typeN>  <MemberNameN>] ;
} <StructureName> [,<rest>]
```

Arguments

<tagName>

This is an optional tag name of a C structure which is normally required by a C compiler. It can be omitted with the xHarbour compiler, because *<tagName>* is ignored.

<C Data type>

The structure members are declared following the opening curly brace. A member declaration begins with the data type of the structure member. All data types listed in the #include file Wintypes.ch are recognized as C data types for member declarations.

<MemberName>

This is the symbolic name of a structure member. It must follow the same naming rule like a memory variable. That is, it must begin with an underscore or an alphabetic character, followed by an underscore or alphanumeric characters.

<StructureName>

This is the symbolic name of the structure to declare.

<rest>

This is only used by a C compiler and ignored by the xHarbour compiler.

Description

The *typedef struct* statement declares a new C structure class and implements it. C structures are supported in xHarbour for complete integration of non-xHarbour DLLs into an xHarbour application. Functions contained in external DLLs often require data being provided in form of structures, which is a data type that does not exist in xHarbour. The Hash() data type of xHarbour is similar to a C structure, but not identical.

Structures organize multiple data in a contiguous block of memory. Individual data is held in so called "structure members" which are comparable to instance variables of an xHarbour object. As a matter of fact, the *typedef struct* statement declares an xHarbour class that has the instance variables *<MemberName1>* to *<MemberNameN>*. These instance variables hold the values of structure members. A new structure object is created by specifying *<StructureName>* with the (struct) statement.

The *typedef struct* statement is intentionally modelled in C syntax to provide an xHarbour programmer with the easiest form of structure declaration: Copy & Paste. C structures are only required when an xHarbour application needs to call a function in an external DLL via DllCall(). This applies to API functions of the operating system or Third Party DLLs. If such function expects a structure as parameter, the structure declaration is provided in include files (*.H file) and can be copied into the PRG code of the xHarbour application.

Structure declaration via Copy & Paste

When a C structure declaration is copied from a *.H file into a PRG file, it must be modified so that the xHarbour preprocessor is capable of translating the *typedef struct* declaration. At minimum, one

semicolon (;) must be added after the opening curly brace ({). This makes sure that the preprocessor recognizes the structure declaration as one line of PRG code.

After the semicolon is added, the PRG should be compiled to see if the preprocessor recognizes all *<C Data type>* declarations. Common C data types are listed in the #include file Wintypes.ch. Any C data type not listed in this file must be mapped to the basic C data types listed in CStruct.ch.

If a *<C Data type>* is not translated by the preprocessor, it is assumed to be the name of a structure that must be declared. If it is not declared, a runtime error is generated.

C strings

C data types like CHAR, TCHAR or WCHAR are declared in C syntax like an array in xHarbour. For example:

```
typedef struct { ;
   CHAR name[64] :
} TESTSTRUCT
```

The structure member *name* is declared to hold 64 bytes. The C language treats this as a "byte array", while xHarbour programmers would use the data type "Character string" to represent this structure member. To reflect both "points of view", xHarbour structures provide a special object maintaining byte arrays. They have a method *:asString()* which returns the contents of a C byte array as an xHarbour Character string.

Info

See also: C Structure class, DllCall(), pragma pack(), (struct)
Category: C Structure support, xHarbour extensions
Header: cstruct.ch, winapi.ch, wintypes.ch
Source: rtl\cstruct.prg
LIB: xhb.lib
DLL: xhbdll.dll

Example

```
// The example demonstrates how an xHarbour structure object can
// be passed to a Windows Api function.
// Note that the TIME_ZONE_INFORMATION structure holds two
// SYSTEMTIME structures in its members. In addition, the
// StandardName and DaylightName members contain Byte arrays.

   #include "CStruct.ch"              // required for "typedef struct"
   #include "wintypes.ch"             // includes Windows C data types

   pragma pack(4)                     // all Windows structures are
                                      // aligned at a 4 byte boundary

                                      // structures are declared via
                                      // copy&paste from Windows SDK
   typedef struct _SYSTEMTIME { ;
     WORD wYear;           //    ^ this ";" must be added
     WORD wMonth;
     WORD wDayOfWeek;
     WORD wDay;
     WORD wHour;
     WORD wMinute;
     WORD wSecond;
     WORD wMilliseconds;
   } SYSTEMTIME, *PSYSTEMTIME;
```

```
typedef struct _TIME_ZONE_INFORMATION { ;
  LONG Bias;                       //   ^ this ";" must be added
  WCHAR StandardName[32];          // member contains a byte array
  SYSTEMTIME StandardDate;         // member contains a structure
  LONG StandardBias;
  WCHAR DaylightName[32];
  SYSTEMTIME DaylightDate;
  LONG DaylightBias;
} TIME_ZONE_INFORMATION, *PTIME_ZONE_INFORMATION;


#define DC_CALL_STD       0x0020   // calling convention for DllCall()


PROCEDURE Main                     // creation of structure object
   LOCAL oTimeZone := (struct TIME_ZONE_INFORMATION)

   // pass structure object by reference (it is an OUT parameter)
   DllCall( "Kernel32.dll", DC_CALL_STD, ;
            "GetTimeZoneInformation", @oTimeZone )

   ? "Date :"                      // display structure data
   ? "Year :", oTimeZone:standardDate:wYear
   ? "Month:", oTimeZone:standardDate:wMonth
   ? "Day  :", oTimeZone:standardDate:wDay
   ?
   ? "Regular time:"
   ? "Hour :", oTimeZone:standardDate:wHour
   ? "Min. :", oTimeZone:standardDate:wMinute
   ? "Sec. :", oTimeZone:standardDate:wSecond
   ?
   ? "Daylight saving time:"
   ? "Hour :", oTimeZone:dayLightDate:wHour
   ? "Min. :", oTimeZone:dayLightDate:wMinute
   ? "Sec. :", oTimeZone:dayLightDate:wSecond
   ?
   ? "Time zone:"
   ? "Std. :", oTimeZone:standardName:asString()
   ? "Sav. ;", oTimeZone:daylightName:asString()
RETURN
```

VAR

Declares an instance variable of a class.

Syntax

```
VAR <MemberName> [ INIT <expression> ] ;
                 [ READONLY ] [ PERSISTENT ]

VAR <MessageName> IN     <SuperClass>
VAR <MessageName> IS     <MemberName> IN <SuperClass>

VAR <MessageName> TO     <ContainedObject>
VAR <MessageName> IS     <MemberName> TO <ContainedObject>
```

Arguments

<MemberName>

This is the symbolic name of the instance variable to declare. It must begin with a letter or underscore followed by digits, letters or underscores. The symbolic name can contain up to 63 characters.

INIT <expression>

This is the expression which is executed when an object is created. The return value of *<expression>* is assigned to *<MemberName>*.

READONLY

This option declares an instance variable as read only. I.e. when the object exists outside the declaring PRG module, no value can be assigned to *<MemberName>*. The value of *<MemberName>* can be changed within methods of the class.

PERSISTENT

This option specifies that the value assigned to *<MemberName>* is persistent. When an object is serialized with HB_Serialize() and later restored with HB_Deserialize(), the value of *<MemberName>* is also restored. All instance variables not declared as PERSISTENT are initialized with NIL when a serialized object is restored from its character representation.

<MessageName>

This is the symbolic name of an alternative message to declare for the class.

IN <SuperClass>

Optionally, the name of the super class can be specified to which the message should be sent. This requires the class to inherit one or more other classes.

IS <MemberName>

When a message should be directed to a super class, the member variable to access in the super class can be specified with *<MemberName>*.

TO <ContainedObject>

This is the symbolic name of an instance variable holding the object that whose instance variable is accessed.

Description

The VAR statement can only be used in the class declaration between CLASS and ENDCLASS. It declares the symbolic name of an instance variable. This name must be sent to an object to access the instance variable.

VAR is more versatile than the DATA declaration, since it allows for declaring message delegation to contained objects or super classes. Refer to DATA for more information on instance variable declaration.

Info

See also:	CLASS, DATA, EXPORTED:, HBObject()
Category:	Class declaration, Declaration, xHarbour extensions
Header:	hbclass.ch
Source:	vm\classes.c
LIB:	xhb.lib
DLL:	xhbdll.dll

WITH OBJECT

Identifies an object to receive multiple messages.

Syntax

```
WITH OBJECT <object>
 :<message1>
 [<statements>]
 [:<messageN>]
END
```

Arguments

`WITH OBJECT <object>`

<*object*> is the symbolic name of a variable that holds a reference to an object. The object is to receive messages in the WITH OBJECT control structure.

`:<message>`

All expressions that begin with the send operator in the OBJECT WITH block are sent as messages to <*object*>.

Description

The WITH OBJECT control structure delimits a block of statements where the object variable <*object*> receives multiple messages in abbreviated form. The name of the object variable can be omitted from a message sending expression, so that only the send operator (:) followed by the message to send must be typed.

WITH OBJECT basically relieves a programmer from typing. The name of the object variable is typed only once at the beginning of the WITH OBJECT block, and all subsequent messages sent to the object start with the send operator, omitting the object's variable name.

Info

See also: CLASS, METHOD (implementation), HB_QWith(), HB_SetWith()
Category: Control structures, Statements, xHarbour extensions

Example

```
// The example builds a simple database browser using a TBrowse object.
// The columns added to the object and the cursor navigation logic is
// programmed using the WITH OBJECT control structure.

   #include "inkey.ch"

   PROCEDURE Main
      LOCAL oTBrowse, aFields, cField, nKey

      USE Customer

      aFields := Array( FCount() )

      AEval( aFields, {|x,i| aFields[i] := FieldName(i) } )

      oTBrowse := TBrowseDB()

      WITH OBJECT oTBrowse
         FOR EACH cField IN aFields
            :addColumn( TBColumnNew( cField, FieldBlock( cField ) ) )
```

```
      NEXT
   END

   nKey := 0
   DO WHILE nKey <> K_ESC

      WITH OBJECT oTBrowse
         DO WHILE .NOT. :stabilize()
         ENDDO

         nKey := Inkey(0)

         SWITCH nKey
         CASE K_UP
           :up()       ; EXIT
         CASE K_DOWN
           :down()     ; EXIT
         CASE K_LEFT
           :left()     ; EXIT
         CASE K_RIGHT
           :right()    ; EXIT
         CASE K_PGUP
           :pageUp()   ; EXIT
         CASE K_PGDN
           :pageDown() ; EXIT
         CASE K_HOME
           :home()     ; EXIT
         CASE K_END
           :end()      ; EXIT
         END
      END
   ENDDO

   CLOSE Customer
RETURN
```

Categories

Array functions

AAdd()	Adds one element to the end of an array.
AChoice()	Displays a list of items to select one from.
AClone()	Creates a deep copy of an array.
ACopy()	Copy elements from a source array into a target array.
ADel()	Deletes an element from an array.
ADir()	Fills pre-allocated arrays with file and/or directory information.
AEval()	Evaluates a code block for each array element.
AFields()	Fills pre-allocated arrays with structure information of the current work area.
AFill()	Fills an array with a constant value.
AIns()	Inserts an element into an array.
ALenAlloc()	Determines for how much array elements memory is pre-allocated.
Array()	Creates an uninitialized array of specified length.
AScan()	Searches a value in an array beginning with the first element.
ASize()	Changes the number of elements in an array.
ASizeAlloc()	Pre-allocates memory for an array.
ASort()	Sorts an array.
ATail()	Returns the last element of an array.
HB_ArrayBlock()	Creates a set/get code block for an array.
HB_ArrayId()	Returns a unique identifier for an array or an object variable.
HB_ATokens()	Splits a string into tokens based on a delimiter.
HB_ThisArray()	Retrieves an array or object from its pointer.
Len()	Returns the number of items contained in an array, hash or string
RAscan()	Searches a value in an array beginning with the last element.

Assignment operators

:=	Inline assignment to a variable.
= (assignment)	Simple assignment operator (binary): assigns a value to a variable.
= (compound assignment)	Compound assignment (binary): inline operation with assignment.

Associative arrays

HaaDelAt()	Removes a key/value pair from an associative array.
HaaGetKeyAt()	Retrieves the key from an associative array by its ordinal position.
HaaGetPos()	Retrieves the ordinal position of a key in an associative array.
HaaGetRealPos()	Retrieves the sort order of a key in an associative array.
HaaGetValueAt()	Retrieves the value from an associative array by its ordinal position.
HaaSetValueAt()	Changes the value in an associative array by its ordinal position.
HGetAACompatibility()	Checks if a hash is compatible with an associative array.
HGetVaaPos()	Retrieves the sort order of all keys in an associative array.
HSetAACompatibility()	Enables or disables associative array compatibility for an empty hash.

Background processing

HB_BackGroundActive()	Queries and/or changes the activity of a single background task.
HB_BackGroundAdd()	Adds a new background task.
HB_BackGroundDel()	Removes a background task from the internal task list.

HB_BackGroundReset()	Resets the internal counter of background tasks.
HB_BackGroundRun()	Enforces execution of one or all background tasks.
HB_BackGroundTime()	Queries or changes the wait interval in milliseconds after which the task is executed.
HB_IdleAdd()	Adds a background task for being executed during idle states.
HB_IdleDel()	Removes a task from the list of idle tasks.
HB_IdleReset()	Resets the internal counter of idle tasks.
HB_IdleSleep()	Halts idle task processing for a number of seconds.
HB_IdleSleepMSec()	Queries or changes the default time interval for idle task processing.
HB_IdleState()	Signals an idle state.
HB_IdleWaitNoCPU()	Toggles the mode for CPU usage in Idle wait states.
SET BACKGROUND TASKS	Enables or disables the activity of background tasks.
SET BACKGROUNDTICK	Defines the processing interval for background tasks.

Binary functions

Bin2I()	Converts a signed short binary integer (2 bytes) into a numeric value.
Bin2L()	Converts a signed long binary integer (4 bytes) into a numeric value.
Bin2U()	Converts an unsigned long binary integer (4 bytes) into a numeric value.
Bin2W()	Converts an unsigned short binary integer (2 bytes) into a numeric value.
CtoF()	Converts an 8 byte string to a floating point number.
I2Bin()	Converts a numeric value to a signed short binary integer (2 bytes).
L2bin()	Converts a numeric value to a signed long binary integer (4 bytes).
U2bin()	Converts a numeric value to an unsigned long binary integer (4 bytes).
W2bin()	Converts a numeric value to an unsigned short binary integer (2 bytes).

Bitwise functions

BitToC()	Translates bits of an integer to a character string.
CharAND()	Binary ANDs the ASCII codes of characters in two strings.
CharNOT()	Binary NOTs the ASCII codes of characters in two strings.
CharOR()	Binary ORs the ASCII codes of characters in two strings.
CharRLL()	Rotates bits in a character string to the left.
CharRLR()	Rotates bits in a character string to the right.
CharSHL()	Shifts bits in a character string to the left.
CharSHR()	Shifts bits in a character string to the right.
CharXOR()	Binary XORs the ASCII codes of characters in two strings.
HB_BitAnd()	Performs a bitwise AND operation with numeric integer values.
HB_BitIsSet()	Checks if a bit is set in a numeric integer value.
HB_BitNot()	Performs a bitwise NOT operation with a numeric integer value.
HB_BitOr()	Performs a bitwise OR operation with numeric integer values.
HB_BitReset()	Sets a bit in a numeric integer value to 0.
HB_BitSet()	Sets a bit in a numeric integer value to 1.
HB_BitShift()	Shifts bits in a numeric integer value.
HB_BitXOr()	Performs a bitwise XOR operation with numeric integer values.
NumAND()	Performs bitwise AND operations for a list of integer values.
NumAndX()	Performs bitwise AND operations for a list of integer values.
NumNOT()	Performs a bitwise NOT operation with a numeric integer value.
NumNotX()	Performs a bitwise NOT operation with a numeric integer value.
NumOR()	Performs a bitwise OR for a list of integer values.
NumOrX()	Performs a bitwise OR for a list of integer values.
NumRoL()	Rotates bits of a numeric 16-bit integer value to the left.
NumRolX()	Rotates bits of a numeric integer value to the left.
NumXOR()	Performs a bitwise XOR operation with two numeric 32-bit integer values.
NumXorX()	Performs a bitwise XOR operation with two numeric integer values.
SetBit()	Sets one or more bits of a numeric integer value to 1.

Bitwise operators

& (bitwise AND)	Bitwise AND operator (binary): performs a logical AND operation.
<<	Left-shift operator (binary): shifts bits to the left.
>>	Right-shift operator (binary): shifts bits to the right.
^^	Bitwise XOR operator (binary): performs a logical XOR operation.
| (bitwise OR)	Bitwise OR operator (binary): performs a logical OR operation.

Blob functions

BlobDirectExport()	Exports the contents of a binary large object (BLOB) to a file.
BlobDirectGet()	Loads data of a binary large object (BLOB) into memory.
BlobDirectImport()	Imports a file into a binary large object (BLOB).
BlobDirectPut()	Assigns data to a binary large object (BLOB).
BlobExport()	Exports the contents of a memo field holding a binary large object (BLOB) to a file.
BlobGet()	Reads the contents of a memo field holding a binary large object (BLOB).
BlobImport()	Imports a file into a memo field.
BlobRootDelete()	Deleted the root area of a BLOB file.
BlobRootGet()	Retrieves data from the root area of a BLOB file.
BlobRootLock()	Places a lock on the root area of a BLOB file.
BlobRootPut()	Stores data in the root area of a BLOB file.
BlobRootUnlock()	Releases the lock on the root area of a BLOB file.

C Structure support

(struct)	Creates a new structure object
C Structure class	Abstract class for C structure support.
HB_ArrayToStructure()	Converts an array to a binary C structure.
HB_SizeofCStructure()	Calculates the amount of memory required to store a C structure.
HB_StructureToArray()	Converts values contained in a binary C structure string to an array.
pragma pack()	Defines the byte alignment for the next structure declaration.
typedef struct	Declares a new structure in C syntax.

CFTS functions

CFTSAdd()	Adds a text string entry to a Full Text Search index file.
CFTSClose()	Closes a Full Text Search index file.
CFTSCrea()	Creates a new Full Text Search index file.
CFTSDelete()	Marks an index entry as deleted in a Full Text Search index file.
CFTSIfDel()	Checks if a Full Text Search index entry is marked as deleted.
CFTSNext()	Searches a Full Text Search index file for a matching index entry.
CFTSOpen()	Opens a Full Text Search index file.
CFTSRecn()	Returns the number of index entries in a Full Text Search index file.
CFTSReplac()	Changes a Full Text Search index entry.
CFTSSet()	Defines a search string for subsequent CFTSNext() calls.
CFTSUndel()	Removes the deletion mark from an index entry in a Full Text Search index file.
CFTSVeri()	Verifies a CFTSNext() match against the index key.
CFTSVers()	Returns version information for HiPer-SEEK functions.

Character functions

AddASCII()	Adds a numeric value to the ASCII code of a specified character in a string.
AfterAtNum()	Extracts the remainder of a string after the last occurrence of a search string.
AllTrim()	Removes leading and trailing blank spaces from a string.
ASCIISum()	Sums the ASCII codes of all characters in a string.
AscPos()	Determines the ASCII code of a specified character in a string.
At()	Locates the position of a substring within a character string.
AtAdjust()	Justifies a character sequence within a string.
AtNum()	Searches multiple occurrences of a substring within a string.
AtRepl()	Searches and replaces a substring within a string.
AtSkipStrings()	Locates the position of a substring within a character string.
AtToken()	Returns the position of the n-th token in a string.
BeforAtNum()	Extracts the remainder of a string before the last occurrance of a search string.
Center()	Returns a string for centered display.
CharAdd()	Creates a string from the sum of ASCII codes of two strings.
CharAND()	Binary ANDs the ASCII codes of characters in two strings.
CharEven()	Extracts characters at even positions from a string.
CharHist()	Creates a histogram of characters in a character string
CharList()	Removes duplicate characters from a string.
CharMirr()	Reverses the order of characters in a string.
CharMix()	Merges the characters of two strings.
CharNoList()	Returns a string containing all characters not included in a string.
CharNOT()	Binary NOTs the ASCII codes of characters in two strings.
CharOdd()	Extracts characters at odd positions from a string.
CharOne()	Removes duplicate adjacent characters from a string.
CharOnly()	Removes all characters but the specified ones from a string
CharOR()	Binary ORs the ASCII codes of characters in two strings.
CharPack()	Compresses a string.
CharRela()	Tests if two substrings in two strings have the same position.
CharRelRep()	Replaces characters if two substrings in two strings have the same position.
CharRem()	Deletes specified characters from a string.
CharRepl()	Searches a list of characters and replaces them with a corresponding list.
CharRLL()	Rotates bits in a character string to the left.
CharRLR()	Rotates bits in a character string to the right.
CharSHL()	Shifts bits in a character string to the left.
CharSHR()	Shifts bits in a character string to the right.
CharSList()	Removes duplicate characters from a string and sorts the result.
CharSort()	Sorts character (sequences) within a string.
CharSpread()	Formats a character string for block paragraphs.
CharSub()	Creates a string by subtracting ASCII codes of two strings.
CharSwap()	Exchanges adjacent characters in a string.
CharUnpack()	Uncompresses a CharPack() compressed string.
CharXOR()	Binary XORs the ASCII codes of characters in two strings.
CheckSum()	Calculates the checksum for a character string.
CountLeft()	Counts a specified character from the left side of a string.
CountRight()	Counts a specified character from the right side of a string.
Crypt()	Encrypts or decrypts a character string.
CSetAtMuPa()	Queries or changes the multi-pass mode for At***() functions.
CSetRef()	Queries or changes the pass-by-reference mode for several string functions.
Expand()	Inserts characters between all characters in a string.
HardCR()	Replaces soft carriage returns with hard CRs in a character string.

HB_ATokens()	Splits a string into tokens based on a delimiter.
HB_AtX()	Locates a substring within a character string based on a regular expression.
HB_Crypt()	Encrypts a character string.
HB_Decrypt()	Decrypts an encrypted character string.
HB_FNameMerge()	Composes a file name from individual components.
HB_FNameSplit()	Splits a file name into individual components.
HB_ReadLine()	Scans a formatted character string or memo field for text lines.
HB_RegEx()	Searches a string using a regular expression
HB_RegExAll()	Parses a string and fills an array with parsing information.
HB_RegExAtX()	Parses a string and fills an array with parsing information.
HB_RegExComp()	Compiles a regular expression into a binary search pattern.
HB_RegExMatch()	Tests if a string contains a substring using a regular expression
HB_RegExReplace()	Searches and replaces characters within a character string using a regular expression.
HB_RegExSplit()	Parses a string using a regular expression and fills an array.
HexToStr()	Converts a Hex encoded character string to an ASCII string.
IsAlNum()	Checks if the first character of a string is alpha-numeric.
IsAlpha()	Checks if the first character of a string is a letter.
IsAscii()	Checks if the first character of a string is a 7-bit ASCII character.
IsCntrl()	Checks if the first character of a string is a control character.
IsDigit()	Checks if the first character of a string is a digit.
IsGraph()	Checks if the first character of a string is a 7-bit graphical ASCII character.
IsLower()	Checks if the first character of a string is a lowercase letter.
IsPrint()	Checks if the first character of a string is a printable 7-bit ASCII character.
IsPunct()	Checks if the first character of a string is a punctuation character.
IsSpace()	Checks if the first character of a string is a white-space character.
IsUpper()	Checks if the first character of a string is an uppercase letter.
IsXDigit()	Checks if the first character of a string is a hexadecimal digit.
JustLeft()	Left justifies characters in a character string.
JustRight()	Right justifies characters in a character string.
Left()	Extracts characters from the left side of a string
Len()	Returns the number of items contained in an array, hash or string
Lower()	Converts a character string to lowercase.
LTrim()	Removes leading white-space characters from a character string.
MaxLine()	Returns the longest line in an ASCII formatted character string.
MemoEdit()	Displays and/or edits character strings and memo fields in text mode.
MemoLine()	Extracts a line of text from a formatted character string or memo field.
MemoRead()	Reads an entire file from disk into memory.
MemoTran()	Replaces "carriage return/line feed" pairs in a character string.
MemoWrit()	Writes a character string or a memo field to a file.
MLCount()	Counts the number of lines in a character string or memo field
MLCToPos()	Determines the position of a single character in a formatted text string or memo field.
MlPos()	Determines the starting position of a line in a formatted character string or memo field.
MPosToLC()	Calculates row and column position of a character in a formatted string or memo field.
NumAt()	Counts multiple occurrences of a substring within a string.
NumLine()	Returns the number of lines in an ASCII formatted character string.
NumToken()	Returns the number of tokens in a string.
Occurs()	Counts the occurrance of a substring in a character string.
PadC() | PadL() | PadR()	Pads values of data type Character, Date and Numeric with a fill character.
PadLeft()	Pads a character string on the left.
PadRight()	Pads a character string on the right.
PosAlpha()	Returns the position of the first alphabetic characters in a character string.
PosChar()	Replaces a single character at a specified position in a string.
PosDel()	Deletes character(s) at a specified position in a string.
PosDiff()	Searches the first position where two character strings differ.

PosEqual()	Searches the first position where two character strings are equal.
PosIns()	Inserts character(s) at a specified position in a string.
PosLower()	Returns the position of the first lower case letter in a character string.
PosRange()	Retrieves the position of the first character out of a range found in a string.
PosRepl()	Replaces characters in a string beginning at a specified position.
PosUpper()	Returns the position of the first upper case letter in a character string.
RangeRem()	Deletes character(s) within a specified range of characters.
RangeRepl()	Repleaces character(s) within a specified range of characters.
RAt()	Locates the position of a substring within a character string.
RemAll()	Deletes a specified character from both sides of a string.
RemLeft()	Deletes a specified character from the left side of a string.
RemRight()	Deletes a specified character from the right side of a string.
ReplAll()	Replaces a specified character on both sides of a string.
Replicate()	Creates a character string by replicating an input string.
ReplLeft()	Replaces a specified character on the left side of a string.
ReplRight()	Replaces a specified character on the right side of a string.
RestToken()	Restores the global environment of the incremental tokenizer.
Right()	Extracts characters from the right side of a string
RTrim()	Removes trailing blank spaces from a character string.
SaveToken()	Saves the global environment of the incremental tokenizer.
SetAtLike()	Sets the search mode for At***() functions
SoundEx()	Converts a character string using the Soundex algorithm.
Space()	Returns a string consisting of blank spaces.
Str()	Converts a numeric value to a character string.
Strdel()	Deletes characters from a string based on a mask string.
StrDiff()	Calculates the similarity of two strings.
StrSwap()	Exchanges characters between two strings.
StrToHex()	Converts a character string to a Hex string.
StrTran()	Searches and replaces characters within a character string or memo field.
StrZero()	Converts a numeric value to a character string, padded with zeros.
Stuff()	Deletes and/or inserts characters in a string.
SubStr()	Extracts a substring from a character string.
SX_Decrypt()	Decrypts an encrypted character string.
SX_Encrypt()	Encrypts a character string.
TabExpand()	Replaces a tab with a specified number of another character.
TabPack()	Inserts a Tab (Chr(9)) for multiple occurrances of a character.
Token()	Retrieves the n-th token from a string.
TokenAt()	Returns the start and end position of a token.
TokenEnd()	Tests if tokens can still be found with TokenNext().
TokenExit()	Releases memory resources of the global tokenizer environment.
TokenInit()	Initializes the environment for the incremental tokenizer.
TokenLower()	Changes the first character of tokens to lower case.
TokenNext()	Retrieves the next token from a string.
TokenNum()	Returns the number of tokens in a tokenizer environment.
TokenSep()	Retrieves the separating characters of a token.
TokenUpper()	Changes the first character of tokens to upper case.
Trim()	Removes trailing blank spaces from a character string.
Upper()	Converts a character string to uppercase.
Val()	Convert a character string containing digits to numeric data type
ValPos()	Returns the numeric value of a digit at a specified position in a string.
Wild2RegEx()	Converts a character string including wild card characters to a regular expression.
WildMatch()	Tests if a string begins with a search pattern.
WordOne()	Removes duplicate adjacent words (2-byte sequences) from a string.
WordOnly()	Removes all words (2-byte sequence) but the specified ones from a string
WordRem()	Deletes specified words (2-byte sequence) from a string.
WordRepl()	Replaces words (2-byte sequences) in a string with a specified word.
WordSwap()	Exchanges adjacent words (2-byte sequences) in a string.
WordToChar()	Replaces words (2 byte sequence) with characters (1 byte)

Character operators

$	Substring operator (binary): search substring in string.
+	Plus operator: add values, concatenate values and unary positive.
-	Minus operator: add values, concatenate values and unary negative.
HAS	Searches a character string for a matching regular expression.
IN	Searches a value in another value.
LIKE	Compares a character string with a regular expression.
[ ] (string)	Character operator (unary): retrieves a character from a string.

Checksum functions

CheckSum()	Calculates the checksum for a character string.
HB_CheckSum()	Calculates the checksum for a stream of data using the Adler32 algorithm.
Hb_CRC32()	Calculates the checksum for a stream of data using the CRC 32 algorithm.
HB_MD5()	Calculates a message digest for a stream of data using the MD5 algorithm.
HB_MD5File()	Calculates a message digest for a file using the MD5 algorithm.

Class declaration

ACCESS	Declares an ACCESS method of a class.
ASSIGN	Declares an ASSIGN method of a class.
ASSOCIATE CLASS	Defines a scalar class for a native data type.
CLASS	Declares the class function of a user-defined class.
CLASSDATA	Declares a class variable of a class.
CLASSMETHOD	Declares the symbolic name of a class method.
DATA	Declares an instance variable of a class.
DELEGATE	Declares a message to be directed to a contained object.
DESTRUCTOR	Declares a method to be called by the garbage collector.
ENABLE TYPE CLASS	Activates scalar classes for native data tyes.
ERROR HANDLER	Declares the method for error handling within a class.
EXPORTED:	Declares the EXPORTED attribute for a group of member variables and/or methods.
EXTEND CLASS...WITH DATA	Adds a new member variable to an existing class.
EXTEND CLASS...WITH METHOD	Adds a new method to an existing class.
HIDDEN:	Declares the HIDDEN attribute for a group of member variables and/or methods.
INLINE METHOD	Declares and implements an inline method that spans across multiple lines.
MESSAGE	Declares a message name for a method.
METHOD (declaration)	Declares the symbolic name of a method.
METHOD (implementation)	Declares the implementation code of a method.
METHOD...OPERATOR	Declares a method to be executed with an operator.
METHOD...VIRTUAL	Declares a method as virtual.
OPERATOR	Overloads an operator and declares a method to be invoked for it.
OVERRIDE METHOD	Replaces a method in an existing class.
PROTECTED:	Declares the PROTECTED attribute for a group of member variables and methods.
VAR	Declares an instance variable of a class.

Code block functions

AEval()	Evaluates a code block for each array element.
DbEval()	Evaluates a code block in a work area.

Eval() Evaluates a code block.
FieldBlock() Creates a set/get code block for a field variable
FieldWBlock() Creates a set/get code block for a field variable in a particular work area.
HB_RestoreBlock() Converts binary information back to a code block.
HB_SaveBlock() Utility function for code block serialization.
HEval() Evaluates a code block with each hash element.
MemVarBlock() Creates a set/get code block for a dynamic memory variable.

Comparison operators

$ Substring operator (binary): search substring in string.
< Less than operator (binary): compares the size of two values.
<= Less than or equal operator (binary): compares the size of two values.
<> != # Not equal operator (binary): compares two values for inequality.
= (comparison) Equal operator (binary): compares two values for equality.
== Exact equal operator (binary): compares two values for identity.
> Greater than operator (binary): compares the size of two values.
>= Greater than or equal operator (binary): compares the size of two values.
HAS Searches a character string for a matching regular expression.
IN Searches a value in another value.
LIKE Compares a character string with a regular expression.

Console commands

? | ?? Displays values of expressions to the console window.
EJECT Ejects the current page from the printer.
SET ALTERNATE Records the console output in a text file.
SET CONSOLE Sets if console display is shown on the screen.
SET PRINTER Enables or disables output to the printer or redirects printer output.
TEXT Displays a block of literal text.

Control structures

BEGIN SEQUENCE Declares a control structure for error handling.
DO CASE Executes a block of statements based on one or more conditions.
DO WHILE Executes a block of statements while a condition is true.
FOR Executes a block of statements a specific number of times.
FOR EACH Iterates elements of data types that can be seen as a collection.
HB_EnumIndex() Returns the current ordinal position of a FOR EACH iteration.
IF Executes a block of statements based on one or more conditions.
RETURN Branches program control to the calling routine.
SWITCH Executes one or more blocks of statements.
TRY...CATCH Declares a control structure for error handling.
WITH OBJECT Identifies an object to receive multiple messages.

Conversion functions

AmPm() Converts a time string into am/pm format.
AnsiToHtml() Inserts HTML character entities into an ANSI text string,
Asc() Returns the ASCII code for characters.
Bin2I() Converts a signed short binary integer (2 bytes) into a numeric value.
Bin2L() Converts a signed long binary integer (4 bytes) into a numeric value.
Bin2U() Converts an unsigned long binary integer (4 bytes) into a numeric value.
Bin2W() Converts an unsigned short binary integer (2 bytes) into a numeric value.
CDoW() Returns the name of a week day from a date.

Chr()	Converts a numeric ASCII code to a character.
CMonth()	Returns the name of a month from a date.
ConvToAnsiCP()	Converts an OEM string to the ANSI character set.
ConvToOemCP()	Converts an ANSI string to the OEM character set.
CStr()	Converts a value to a character string.
CStrToVal()	Converts a character string to a value of specific data type.
CtoD()	Converts a character string into a Date value
CtoF()	Converts an 8 byte string to a floating point number.
CtoT()	Converts a character string into a DateTime value
DateTime()	Returns the current date and time from the operating system.
Day()	Extracts the numeric day number from a Date value.
Descend()	Converts a value for a descending index.
DoW()	Determines the numeric day of the week from a date.
DtoC()	Converts a Date value to a character string in SET DATE format.
DtoR()	Converts an angle from degrees to radians.
DtoS()	Converts a Date value to a character string in YYYYMMDD format.
HB_AnsiToOem()	Converts a character string from the ANSI to the OEM character set.
HB_Crypt()	Encrypts a character string.
HB_Decode()	Provides a functional equivalent for the DO CASE statement.
HB_DecodeOrEmpty()	Provides a functional equivalent for the DO CASE statement.
HB_Decrypt()	Decrypts an encrypted character string.
HB_DeSerialize()	Converts a binary string back to its original data type.
HB_OemToAnsi()	Converts a character string from the OEM to the ANSI character set.
HB_RestoreBlock()	Converts binary information back to a code block.
HB_SaveBlock()	Utility function for code block serialization.
HB_Serialize()	Converts an arbitrary value to a binary string.
HB_ValToStr()	Converts values of simple data types to character string.
HexToNum()	Converts a Hex string to a numeric value.
HexToStr()	Converts a Hex encoded character string to an ASCII string.
Hour()	Extracts the hour from a DateTime value
HtmlToAnsi()	Converts an HTML formatted text string to the ANSI character set.
HtmlToOem()	Converts an HTML formatted text string to the OEM character set
I2Bin()	Converts a numeric value to a signed short binary integer (2 bytes).
If() | IIf()	Returns the result of an expression based on a logical expression
L2bin()	Converts a numeric value to a signed long binary integer (4 bytes).
Minute()	Extracts the minute from a DateTime value
Month()	Extracts the numeric month number from a Date value.
NumToHex()	Converts a numeric value or a pointer to a Hex string.
OemToHtml()	Inserts HTML character entities into an OEM text string.
PadC() | PadL() | PadR()	Pads values of data type Character, Date and Numeric with a fill character.
PrgExpToVal()	Converts a character string obtained from ValToPrgExp() back to the original data type.
SoundEx()	Converts a character string using the Soundex algorithm.
StoD()	Converts a "yyyymmdd" formatted string to a Date value
StoT()	Converts a "YYYYMMDDhhmmss.ccc" formatted string to a DateTime value
Str()	Converts a numeric value to a character string.
StringToLiteral()	Creates a literal character string from a string.
StrToHex()	Converts a character string to a Hex string.
StrZero()	Converts a numeric value to a character string, padded with zeros.
SX_Decrypt()	Decrypts an encrypted character string.
SX_DtoP()	Converts a Date value into a 3-byte character string.
SX_Encrypt()	Encrypts a character string.
SX_PtoD()	Unpacks a packed 3-byte date value.
Transform()	Converts values to a PICTURE formatted character string.
TtoC()	Converts a DateTime value to a character string in SET DATE and SET TIME format.
TtoS()	Converts a Date value to a character string in YYYYMMDDhhmmss.ccc format.

U2bin()	Converts a numeric value to an unsigned long binary integer (4 bytes).
Val()	Convert a character string containing digits to numeric data type
ValToPrg()	Converts a value to PRG code.
ValToPrgExp()	Converts a value to a character string holding a macro-expression.
W2bin()	Converts a numeric value to an unsigned short binary integer (2 bytes).
Word()	Converts a floating point number to a 32-it integer value.
Year()	Extracts the numeric year from a Date value

CT:Database

DbfSize()	Returns the size of a database file in memory that is opened in a workarea.
FieldDeci()	Returns the decimal places of a database field.
FieldNum()	Returns the ordinal position of a field in a database.
FieldSize()	Returns the length of a database field.

CT:DateTime

AddMonth()	Adds or subtracts a number of months to/from a Date value.
BoM()	Returns the date of the first day of a month.
BoQ()	Returns the date of the first day of a quarter.
BoY()	Returns the date of the first day of a year.
CtoDoW()	Returns the number of a week day from its name.
CtoMonth()	Returns the number of a month from its name.
DaysInMonth()	Returns the number of days in a month.
DaysToMonth()	Returns the number of days from first January to the beginning of a month.
DMY()	Formats a date as "dd. Month yyyy"
DoY()	Returns the day number of a Date value in a year.
EoM()	Returns the date of the last day in a month.
EoQ()	Returns the date of the last day in a quarter.
EoY()	Returns the date for the last day of a year.
IsLeap()	Checks if a Date value belongs to a leap year.
LastDayoM()	Returns the number of days in a month.
MDY()	Formats a date as "Month dd, yy".
MilliSec()	Defines a time delay in milliseconds.
NtoCDoW()	Converts a numeric week day to its name.
NtoCMonth()	Converts a numeric month to its name.
Quarter()	Returns the quarter a date belongs to.
SecToTime()	Converts numeric seconds into a time formatted character string.
SetDate()	Changes the system date from a Date value.
SetNewDate()	Changes the system date from Numeric values.
SetNewTime()	Changes the system time from Numeric values.
SetTime()	Changes the system time from a Time string.
ShowTime()	Displays the system time continuously at a specified screen position.
TimeToSec()	Calculates the number of seconds since midnight.
TimeValid()	Checks if a character string is a valid time string.
WaitPeriod()	Defines a wait period and allows for time controlled loops.
Week()	Calculates the numeric calendar week from a date.
WoM()	Calculates the week number in a month.

CT:DiskUtil

DeleteFile()	Deletes a file with error handling.
DirMake()	Creates a directory.
DirName()	Returns the current directory.
DiskFormat()	Formats a floppy disk.

DiskFree()	Returns the free storage space of a disk drive in bytes.
DiskReady()	Test if a disk drive is ready.
DiskReadyW()	Tests if a drive can be written to.
DiskTotal()	Returns the totl storage space of a disk drive in bytes.
DiskUsed()	Returns the used storage space of a disk drive in bytes.
DriveType()	Determines the type of a drive.
FileAppend()	Concatenates two files.
FileAttr()	Returns the attributes of a file.
FileCClose()	Closes the file opened in backup mode with FileCopy().
FileCCont()	Continues file copying in backup mode of FileCopy().
FileCOpen()	Tests if the file opened in backup mode with FileCopy() is still open.
FileCopy()	Copies files normally or in backup mode.
FileDate()	Returns the date of a file.
FileDelete()	Deletes one or more files specified by a file mask and file attributes.
FileMove()	Moves a file to another directory.
FileSeek()	Seeks files specified by a file mask and file attributes.
FileSize()	Returns the size of a file.
FileStr()	Reads a string from a file beginning at a specified offset.
FileTime()	Returns the change time of a file.
FileValid()	Tests if a string contains a valid file name.
FloppyType()	Determines the type of a floppy drive.
GetVolInfo()	Retrieves the volume label of a disk.
IsDir()	Checks if a character string contains the name of an existing directory.
NumDiskL()	Returns the number of logical drives.
RenameFile()	Renames a file and handles errors.
SetFAttr()	Sets file attributes.
SetFCreate()	Sets the default file attribute(s) for creating files.
SetFDaTi()	Sets the last change date and time of a file.
StrFile()	Writes a string to a file starting at a specified position.
TrueName()	Completes a relative path to include the root directory.
VolSerial()	Returns the serial number of a disk drive.
Volume()	Sets the volume label of a disk.

CT:GetSys

CountGets()	Returns the number of Get fields in the current Getlist array.
CurrentGet()	Returns the position of the current Get field in the Getlist array.
GetFldCol()	Returns the screen column position of a Get field.
GetFldRow()	Returns the screen row position of a Get field.
GetFldVar()	Returns the name of a Get variable.
GetSecret()	Allows for hidden Get input for character strings.
RestGets()	Restores a previously saved !IGetlist!EI array.
RestSetKey()	Restores SetKey() settings and associated code blocks.
SaveGets()	Saves the current !IGetlist!EI array for nested READs.
SaveSetKey()	Saves SetKey() settings and associated code blocks.

CT:Math

ACos()	Calculates the arc cosine.
ASin()	Calculates the arc sine.
ATan()	Calculates the arc tangent.
ATn2()	Calculates the radians of an angle from sine and cosine.
Ceiling()	Rounds a decimal number to the next higher integer.
Cos()	Calculates the cosine for an angle.
CosH()	Calculates the hyperbolic cosine for an angle.
Cot()	Calculates the cotangent.
DtoR()	Converts an angle from degrees to radians.

Fact()	Calculates the factorial of a number.
Floor()	Rounds a decimal number to the next lower integer.
Fv()	Calculates the future value of constant, periodic investments.
GetPrec()	Retrieves computing precision for trigonometric functions.
Log10()	Calculates the base 10 logarithm.
Payment()	Calculates a periodical payment for loans.
Periods()	Calculates the duration for paying back a loan at fixed interest rate and payments.
Pi()	Returns Pi with highest accuracy.
Pv()	Calculates the present value of future capital, based on periodic investments.
Rate()	Calculates the interest rate for a loan.
RtoD()	Converts angles from radians to degrees.
SetPrec()	Specifies the computing precision for trigonometric functions.
Sign()	Converts the sign of a number to a numeric value.
Sin()	Calculates the sine.
SinH()	Calculates the hyperbolic sine.
Tan()	Calculates the tangent.
TanH()	Calculates the hyperbolic tangent.

CT:Miscellaneous

Blank()	Returns empty values for the data types A, C, D, L, M and N.
Complement()	Creates the complement for values of data type C, D, L, M, N
Default()	Assigns a default value to a variable.
DosParam()	Returns the command line parameters passed to an xHarbour application.
ExeName()	Returns the EXE file name of an xHarbour application.
KbdStat()	Determines the state of special keys like Ctrl or Shift keys.
KeySec()	Starts a timer to write a key code into the keyboard buffer.
KeyTime()	Writes a key code into the keyboard buffer at a specified time.
LtoC()	Converts a logical value to a character.
Nul()	Returns a null string ("").
XtoC()	Converts values of data type C, D, L, M, N to a string.

CT:Network

NetCancel()	Releases an existing network connection.
NetDisk()	Tests if a drive is a network drive.
NetPrinter()	Tests if the current printer is a network printer.
NetRedir()	Establishes a connection to a server.
NetRmtName()	Returns the remote name of a network device.
Network()	Tests for the existence of a network.
NNetwork()	Tests for the existence of a Novell network.

CT:NumBits

BitToC()	Translates bits of an integer to a character string.
Celsius()	Converts degrees Fahrenheit to Celsius.
ClearBit()	Sets one or more bits of a numeric integer value to 0.
CtoBit()	Converts a character string to an integer based on a bit pattern.
CtoF()	Converts an 8 byte string to a floating point number.
CtoN()	Converts a string of digits to an integer of the specified base.
Exponent()	Calculates the exponent of a floating point number.
Fahrenheit()	Converts degrees Celsius to Fahrenheit.
FtoC()	Converts a floating point number to an 8 byte binary string.
Infinity()	Returns the largest number.

IsBit()	Checks whether a bit at a specified position is set.
LtoN()	Converts a logical value to a numeric value.
Mantissa()	Calculates the mantissa of a floating point number.
NtoC()	Converts an integer to a string of digits for a specified number base.
NumAND()	Performs bitwise AND operations for a list of integer values.
NumAndX()	Performs bitwise AND operations for a list of integer values.
NumCount()	Installs and/or increments an internal counter value.
NumHigh()	Retrieves the high byte of a 16-bit integer.
NumLow()	Retrieves the low byte of a 16 bit integer.
NumMirr()	Mirrors 8 or 16 bits of a 16-bit integer.
NumMirrX()	Mirrors bits of a numeric integer value.
NumNOT()	Performs a bitwise NOT operation with a numeric integer value.
NumNotX()	Performs a bitwise NOT operation with a numeric integer value.
NumOR()	Performs a bitwise OR for a list of integer values.
NumOrX()	Performs a bitwise OR for a list of integer values.
NumRoL()	Rotates bits of a numeric 16-bit integer value to the left.
NumRolX()	Rotates bits of a numeric integer value to the left.
NumXOR()	Performs a bitwise XOR operation with two numeric 32-bit integer values.
NumXorX()	Performs a bitwise XOR operation with two numeric integer values.
SetBit()	Sets one or more bits of a numeric integer value to 1.

CT:Printer

PrintReady()	Tests if a printer connected to a specified port is ready.
PrintSend()	Sends a string or a single character to the printer.
PrintStat()	Returns the status of a printer.

CT:Settings

CSetCent()	Queries or changes the SET CENTURY setting.
CSetCurs()	Queries or changes the SET CURSOR setting.
CSetKey()	Retrieves the code block associated with a key.
CSetSafety()	Retrieves and/or changes the safety switch used in CA-Tools file operations.
KSetCaps()	Queries or changes the status of the Caps lock key
KSetIns()	Queries or changes the status of the Insert/Overwrite key
KSetNum()	Queries or changes the status of the Num lock key
KSetScroll()	Queries or changes the status of the Scroll lock key
SetLastKey()	Changes the return value of function LastKey()

CT:String manipulation

AddASCII()	Adds a numeric value to the ASCII code of a specified character in a string.
AfterAtNum()	Extracts the remainder of a string after the last occurrence of a search string.
ASCIISum()	Sums the ASCII codes of all characters in a string.
AscPos()	Determines the ASCII code of a specified character in a string.
AtAdjust()	Justifies a character sequence within a string.
AtNum()	Searches multiple occurrences of a substring within a string.
AtRepl()	Searches and replaces a substring within a string.
AtToken()	Returns the position of the n-th token in a string.
BeforAtNum()	Extracts the remainder of a string before the last occurrance of a search string.
Center()	Returns a string for centered display.
CharAdd()	Creates a string from the sum of ASCII codes of two strings.

CharAND()	Binary ANDs the ASCII codes of characters in two strings.
CharEven()	Extracts characters at even positions from a string.
CharHist()	Creates a histogram of characters in a character string
CharList()	Removes duplicate characters from a string.
CharMirr()	Reverses the order of characters in a string.
CharMix()	Merges the characters of two strings.
CharNoList()	Returns a string containing all characters not included in a string.
CharNOT()	Binary NOTs the ASCII codes of characters in two strings.
CharOdd()	Extracts characters at odd positions from a string.
CharOne()	Removes duplicate adjacent characters from a string.
CharOnly()	Removes all characters but the specified ones from a string
CharOR()	Binary ORs the ASCII codes of characters in two strings.
CharPack()	Compresses a string.
CharRela()	Tests if two substrings in two strings have the same position.
CharRelRep()	Replaces characters if two substrings in two strings have the same position.
CharRem()	Deletes specified characters from a string.
CharRepl()	Searches a list of characters and replaces them with a corresponding list.
CharRLL()	Rotates bits in a character string to the left.
CharRLR()	Rotates bits in a character string to the right.
CharSHL()	Shifts bits in a character string to the left.
CharSHR()	Shifts bits in a character string to the right.
CharSList()	Removes duplicate characters from a string and sorts the result.
CharSort()	Sorts character (sequences) within a string.
CharSpread()	Formats a character string for block paragraphs.
CharSub()	Creates a string by subtracting ASCII codes of two strings.
CharSwap()	Exchanges adjacent characters in a string.
CharUnpack()	Uncompresses a CharPack() compressed string.
CharXOR()	Binary XORs the ASCII codes of characters in two strings.
CheckSum()	Calculates the checksum for a character string.
CountLeft()	Counts a specified character from the left side of a string.
CountRight()	Counts a specified character from the right side of a string.
Crypt()	Encrypts or decrypts a character string.
CSetAtMuPa()	Queries or changes the multi-pass mode for At***() functions.
CSetRef()	Queries or changes the pass-by-reference mode for several string functions.
Expand()	Inserts characters between all characters in a string.
JustLeft()	Left justifies characters in a character string.
JustRight()	Right justifies characters in a character string.
MaxLine()	Returns the longest line in an ASCII formatted character string.
NumAt()	Counts multiple occurrences of a substring within a string.
NumLine()	Returns the number of lines in an ASCII formatted character string.
NumToken()	Returns the number of tokens in a string.
Occurs()	Counts the occurrance of a substring in a character string.
PadLeft()	Pads a character string on the left.
PadRight()	Pads a character string on the right.
PosAlpha()	Returns the position of the first alphabetic characters in a character string.
PosChar()	Replaces a single character at a specified position in a string.
PosDel()	Deletes character(s) at a specified position in a string.
PosDiff()	Searches the first position where two character strings differ.
PosEqual()	Searches the first position where two character strings are equal.
PosIns()	Inserts character(s) at a specified position in a string.
PosLower()	Returns the position of the first lower case letter in a character string.
PosRange()	Retrieves the position of the first character out of a range found in a string.
PosRepl()	Replaces characters in a string beginning at a specified position.
PosUpper()	Returns the position of the first upper case letter in a character string.
RangeRem()	Deletes character(s) within a specified range of characters.
RangeRepl()	Repleaces character(s) within a specified range of characters.
RemAll()	Deletes a specified character from both sides of a string.

RemLeft()	Deletes a specified character from the left side of a string.
RemRight()	Deletes a specified character from the right side of a string.
ReplAll()	Replaces a specified character on both sides of a string.
ReplLeft()	Replaces a specified character on the left side of a string.
ReplRight()	Replaces a specified character on the right side of a string.
RestToken()	Restores the global environment of the incremental tokenizer.
SaveToken()	Saves the global environment of the incremental tokenizer.
SetAtLike()	Sets the search mode for At***() functions
StrDiff()	Calculates the similarity of two strings.
StrSwap()	Exchanges characters between two strings.
TabExpand()	Replaces a tab with a specified number of another character.
TabPack()	Inserts a Tab (Chr(9)) for multiple occurrances of a character.
Token()	Retrieves the n-th token from a string.
TokenAt()	Returns the start and end position of a token.
TokenEnd()	Tests if tokens can still be found with TokenNext().
TokenExit()	Releases memory resources of the global tokenizer environment.
TokenInit()	Initializes the environment for the incremental tokenizer.
TokenLower()	Changes the first character of tokens to lower case.
TokenNext()	Retrieves the next token from a string.
TokenNum()	Returns the number of tokens in a tokenizer environment.
TokenSep()	Retrieves the separating characters of a token.
TokenUpper()	Changes the first character of tokens to upper case.
ValPos()	Returns the numeric value of a digit at a specified position in a string.
WordOne()	Removes duplicate adjacent words (2-byte sequences) from a string.
WordOnly()	Removes all words (2-byte sequence) but the specified ones from a string
WordRem()	Deletes specified words (2-byte sequence) from a string.
WordRepl()	Replaces words (2-byte sequences) in a string with a specified word.
WordSwap()	Exchanges adjacent words (2-byte sequences) in a string.
WordToChar()	Replaces words (2 byte sequence) with characters (1 byte)

CT:Video

CharWin()	Replaces characters in a specified screen area.
ClearEol()	Clears a row on the screen beginning at a specified position.
ClearSlow()	Clears a screen area incrementally with a delayed imploding effect.
ClearWin()	Clears all or parts of the screen.
ClEol()	Clears characters and colors in a row on the screen.
ClWin()	Clears characters and colors on the screen.
ColorRepl()	Replaces color attributes on the screen.
ColorToN()	Converts a color value to a numeric color attribute.
ColorWin()	Replaces a color attribute in a screen region.
Enhanced()	Selects the enhanced color of SetColor().
FileScreen()	Reads the contents of a screen from a file.
GetClearA()	Returns the default color attribute for clearing the screen.
GetClearB()	Returns the default character for clearing the screen.
InvertAttr()	Exchanges the foreground and background color.
InvertWin()	Exchanges the foreground and background color on the screen.
NtoColor()	Converts a numeric color attribute to a color string.
RestCursor()	Restores shape and position of the screen cursor.
SaveCursor()	Saves the current cursor shape and position.
SayDown()	Outputs a string vertically to the bottom of the screen.
SayMoveIn()	Outputs a string on the screen using a "move in" effect.
SayScreen()	Displays a string on screen keeping existing color attributes.
SaySpread()	Outputs a string on the screen using a "spread" effect.
ScreenAttr()	Returns the numeric color attribute for a specified coordinate on the screen.
ScreenFile()	Writes the contents of the current screen to a file.
ScreenMark()	Searches strings on the screen and changes their color.

ScreenMix()	Mixes a character string with color attributes.
ScreenStr()	Returns the screen contents beginning at a specified position.
SetClearA()	Sets the default color attribute for clearing the screen.
SetClearB()	Sets the default character attribute for clearing the screen.
Standard()	Selects the standard color of SetColor().
StrScreen()	Displays a screen string at the specified position.
Unselected()	Selects the unselected color of SetColor().
UntextWin()	Removes all text characters from the screen.

CT:Window

WAClose()	Closes all windows.
WBoard()	Determines the area that can be used for displaying windows.
WBox()	Draws a frame around the current window.
WCenter()	Centers a window on the screen or makes it entirely visible.
WClose()	Closes the current window and selects the next window as current window.
WCol()	Returns the left column position of the selected window.
WfCol()	Returns the left column position of the usable area in a formatted window.
WfLastCol()	Returns the right column position of the usable area in a formatted window.
WfLastRow()	Returns the bottom row position of the usable area in a formatted window.
WFormat()	Defines the usable display area inside the current window.
WfRow()	Returns the top row position of the usable area in a formatted window.
WInfo()	Returns all coordinates of the current window.
WLastCol()	Returns the right column position of the selected window.
WLastRow()	Returns the bottom row position of the selected window.
WMode()	Determines on which side windows are allowed to be moved off the screen.
WMove()	Changes the upper left coordinate for the current window.
WMSetPos()	Moves the mouse cursor to a new position in a window.
WNum()	Returns the largest window ID of all windows.
WOpen()	Creates a new window.
WRow()	Returns the top row position of the selected window.
WSelect()	Returns and/or selects an open window as the current window.
WSetMouse()	Determines the visibility and/or position of the mouse cursor.
WSetMove()	Toggles the setting for moving windows interactively.
WSetShadow()	Defines the shadow color for windows.
WStack()	Returns window IDs of all open windows.
WStep()	Sets the increments for horizontal and vertical window movement.

Database commands

APPEND BLANK	Creates a new record in the current work area.
APPEND FROM	Imports records from a database file or an ASCII text file.
AVERAGE	Calculates the average of numeric expressions in the current work area.
CLOSE	Closes one or more specified files.
COMMIT	Writes memory buffers of all used work areas to disk.
CONTINUE	Resumes a pending LOCATE command.
COPY STRUCTURE	Copies the current database structure to a new database file.
COPY STRUCTURE EXTENDED	Copies field information to a new database file.
COPY TO	Exports records from the current work area to a database or an ASCII text file.
COUNT	Counts records in the current work area.
CREATE	Creates and opens an empty structure extended database file.
CREATE FROM	Creates a new database file from a structure extended file.
DELETE	Marks one or more records for deletion.

DELETE TAG	Deletes a tag from an index.
FIND	Searches an index for a specified key value.
GO	Moves the record pointer to a specified position.
INDEX	Creates an index and/or index file.
JOIN	Merges records of two work areas into a new database.
LIST	Lists records of a work area to the console, file or printer.
LOCATE	Scans the current work area for a record matching a condition.
PACK	Removes records marked for deletion physically from a database file.
RECALL	Removes a deletion mark from one or more records.
REINDEX	Rebuilds all open indexes in the current work area.
REPLACE	Assigns values to field variables.
SEEK	Searches a value in an index.
SELECT	Changes the current work area.
SET AUTOPEN	Toggles automatic opening of a structural index file.
SET AUTORDER	Defines the default controlling index for automatically opened index files.
SET AUTOSHARE	Defines network detection for shared file access.
SET DBFLOCKSCHEME	Selects the locking scheme for shared database access.
SET DELETED	Specifies visibility of records marked for deletion.
SET EXCLUSIVE	Sets the global EXCLUSIVE open mode for databases.
SET FILTER	Defines a condition for filtering records in the current work area.
SET HARDCOMMIT	Toggles immediate committing of changes to record buffers.
SET INDEX	Opens one or more index files in the current work area.
SET MEMOBLOCK	Defines the default block size for memo files.
SET OPTIMIZE	Toggles filter optimization with indexed databases.
SET ORDER	Selects the controlling index.
SET RELATION	Synchronizes the record pointers in one or more work areas.
SET SCOPE	Changes the top and/or bottom scope for navigating in the controlling index.
SET SCOPEBOTTOM	Changes the bottom scope for navigating in the controlling index.
SET SCOPETOP	Changes the top scope for navigating in the controlling index.
SET STRICTREAD	Toggles read optimization for database access.
SKIP	Moves the record pointer to a new position.
SORT	Creates a new, physically sorted database.
SUM	Calculates the sum of numeric expressions in the current work area.
TOTAL	Creates a new database summarizing numeric fields by an expression.
UNLOCK	Releases file and/or record locks in the current or all work areas
UPDATE	Updates records in the current work area from a second work area.
USE	Opens a database and its associated files in a work area.
ZAP	Delete all records from the current database file

Database drivers

DbSetDriver()	Retrieves and/or selects the default replaceable database driver.
RddInfo()	Queries and/or changes configuration data of RDDs.
RddList()	Retrieves information about available Replaceable Database Drivers (RDDs).
RddName()	Retrieves the name of an RDD used to open files in a work area
RddRegister()	Registers a user defined Replaceable Database Driver (RDD).
RddSetDefault()	Retrieves or changes the default replaceable database driver.
UsrRdd_AreaData()	Queries or attaches user-defined data to a work area of a user-defined RDD.
UsrRdd_AreaResult()	Queries the result of the last operation of a user-defined RDD.
UsrRdd_ID()	Queries the ID of a user-defined RDD.
UsrRdd_RddData()	Queries or attaches user-defined data a user-defined RDD.
UsrRdd_SetBof()	Sets the BoF() flag in a work area of a user-defined RDD.
UsrRdd_SetBottom()	Defines the bottom scope value for a work area of a user-defined RDD.
UsrRdd_SetEof()	Sets the Eof() flag in a work area of a user-defined RDD.
UsrRdd_SetFound()	Sets the Found() flag in a work area of a user-defined RDD.

UsrRdd_SetTop()	Defines the top value for a work area of a user-defined RDD.

Database functions

Alias()	Returns the alias name of a work area.
BlobDirectExport()	Exports the contents of a binary large object (BLOB) to a file.
BlobDirectGet()	Loads data of a binary large object (BLOB) into memory.
BlobDirectImport()	Imports a file into a binary large object (BLOB).
BlobDirectPut()	Assigns data to a binary large object (BLOB).
BlobExport()	Exports the contents of a memo field holding a binary large object (BLOB) to a file.
BlobGet()	Reads the contents of a memo field holding a binary large object (BLOB).
BlobImport()	Imports a file into a memo field.
BlobRootDelete()	Deleted the root area of a BLOB file.
BlobRootGet()	Retrieves data from the root area of a BLOB file.
BlobRootLock()	Places a lock on the root area of a BLOB file.
BlobRootPut()	Stores data in the root area of a BLOB file.
BlobRootUnlock()	Releases the lock on the root area of a BLOB file.
BoF()	Determines if the record pointer reached the begin-of-file boundary.
Browse()	Browse a database file
DbAppend()	Appends a new record to a database open in a work area.
DbClearFilter()	Clears the current filter condition in a work area.
DbClearIndex()	Closes all indexes open in the current work area.
DbClearRelation()	Clears all active relations in a work area.
DbCloseAll()	Close all open files in all work areas.
DbCloseArea()	Closes a database file in a work area.
DbCommit()	Writes database and index buffers to disk.
DbCommitAll()	Writes database and index buffers of all used work areas to disk.
DbCopyExtStruct()	Creates a structure extended database file.
DbCopyStruct()	Creates a new database based on the current database structure.
DbCreate()	Creates an empty database from a structure definition array.
DbCreateIndex()	Creates a new index and/or index file.
DbDelete()	Marks records for deletion.
DbEdit()	Browse records in a table.
DbEval()	Evaluates a code block in a work area.
Dbf()	Retrieves the alias name of the current work area.
DbFieldInfo()	Retrieves information about a database field
DbFileGet()	Copies data from a field into an external file.
DbFilePut()	Copies the contents of an external file into a field.
DbFilter()	Returns the filter expression of a work area-
DbfSize()	Returns the size of a database file in memory that is opened in a workarea.
DbGoBottom()	Moves the record pointer to last record.
DbGoto()	Positions the record pointer on a particular record.
DbGoTop()	Moves the record pointer to first record.
DbInfo()	Queries and/or changes information about a database file open in a work area.
DbJoin()	Merges records of two work areas into a new database.
Dblist()	Displays records of a work area to the console, printer or file.
DbOrderInfo()	Queries and/or changes information about indexes open in a work area.
DbRecall()	Recalls a record previousy marked for deletion.
DbRecordInfo()	Queries and/or changes information about a record of an open database file.
DbReindex()	Rebuilds indexes open in a work area.
DbRelation()	Returns the linking expression of a specified relation.
DbRLock()	Locks a record for write access.
DbRLockList()	Returns a list of locked records.
DbRSelect()	Returns the child work area number of a relation.
DbRUnlock()	Unlocks a record based on its indentifier.

DbSeek()	Searches a value in the controlling index.
DbSelectArea()	Selects the current work area.
DbSetDriver()	Retrieves and/or selects the default replaceable database driver.
DbSetFilter()	Defines a filter condition for a work area.
DbSetIndex()	Opens an index file.
DbSetOrder()	Selects the controlling index.
DbSetRelation()	Relates a child work area with a parent work area.
DbSkip()	Moves the record pointer in a work area.
DbSkipper()	Helper function for browse objects to skip a database
DbSort()	Creates a new, physically sorted database.
DbStruct()	Loads structural information of a database into an array.
DbTableExt()	Retrieves the default database file extension of the current RDD.
DbTotal()	Creates a new database summarizing numeric fields by an expression.
DbUnlock()	Releases file and all record locks in a work area.
DbUnlockAll()	Unlocks all records and releases all file locks in all work areas.
DbUpdate()	Updates records in the current work area from a second work area.
DbUseArea()	Opens a database file in a work area.
Deleted()	Queries the Deleted flag of the current record
Eof()	Determines when the end-of-file is reached.
FieldBlock()	Creates a set/get code block for a field variable
FieldGet()	Retrieves the value of a field variable by its ordinal position.
FieldName()	Retrieves the name of a field variable by its ordinal position.
FieldPos()	Retrieves the ordinal position of a field variable by its name.
FieldPut()	Assigns a value to a field variable by its ordinal position.
FieldWBlock()	Creates a set/get code block for a field variable in a particular work area.
FLock()	Applies a file lock to an open, shared database.
Found()	Checks if the last database search operation was successful
Header()	Returns the size of the database file header.
HS_Add()	Adds a text string entry to a HiPer-SEEK index file.
HS_Close()	Closes a HiPer-SEEK index file.
HS_Create()	Creates a new HiPer-SEEK index file.
HS_Delete()	Marks an index entry as deleted in a HiPer-SEEK index file.
HS_Filter()	Uses a HiPer-SEEK index file as filter for a work area
HS_IfDel()	Checks if a HiPer-SEEK index entry is marked as deleted
HS_Index()	Creates a new HiPer-SEEK index file and fills it with index entries.
HS_KeyCount()	Returns the number of index entries in a HiPer-SEEK index file.
HS_Next()	Searches a HiPer-SEEK index file for a matching index entry.
HS_Open()	Opens a HiPer-SEEK index file.
HS_Replace()	Changes a HiPer-SEEK index entry.
HS_Set()	Defines a search string for subsequent HS_Next() calls.
HS_Undelete()	Removes the deletion mark from an index entry in a HiPer-SEEK index file.
HS_Verify()	Verifies a HS_Next() match against the index key.
HS_Version()	Returns version information for HiPer-SEEK functions.
IndexExt()	Retrieves the default index file extension in a work area.
IndexKey()	Returns the index expression of an open index.
IndexOrd()	Returns the ordinal position of the controlling index in a work area.
LastRec()	Returns the number of records available in a work area.
LUpdate()	Returns the last modification date of a database open in a work area.
NetAppend()	Appends a new record to a database open in shared mode in a work area.
NetCommitAll()	Writes database and index buffers of all used work areas to disk.
NetDbUse()	Opens a database file for shared access in a work area.
NetDelete()	Marks records for deletion.
NetErr()	Determines if a database command has failed in network operation.
NetError()	Determines if a Net*() function has failed.
NetFileLock()	Applies a file lock to an open, shared database.
NetFunc()	Evaluates a code block until a timeout period expires.
NetLock()	Applies locks to a database file or record with timeout.
NetOpenFiles()	Opens databases and associated index files.

NetRecall()	Recalls a record previousy marked for deletion.
NetRecLock()	Locks the current record for write access.
OrdBagExt()	Retrieves the default extension for index files.
OrdBagName()	Retrieves the file name of an open index.
OrdCondSet()	Sets the conditions for index creation.
OrdCount()	Determines the number of indexes open in a work area.
OrdCreate()	Creates a new index.
OrdCustom()	Determines if an index is a custom index.
OrdDescend()	Determines the navigational order of a work area.
OrdDestroy()	Deletes an index from an index file.
OrdFindRec()	Searches a record number in the controlling index.
OrdFor()	Retrieves the FOR expression of an index.
OrdIsUnique()	Retrieves the UNIQUE flag of an index.
OrdKey()	Returns the key expression of an index.
OrdKeyAdd()	Adds an index key to a custom built index.
OrdKeyCount()	Retrieves the total number of keys included in an index.
OrdKeyDel()	Removes an index key from a custom built index.
OrdKeyGoTo()	Moves the record pointer to a logical record number.
OrdKeyNo()	Returns the logical record number of the current record.
OrdKeyRelPos()	Returns or sets the relative position of the current record.
OrdKeyVal()	Retrieves the index value of the current record from the controlling index.
OrdListAdd()	Opens and optionally activates a new index in a work area.
OrdListClear()	Closes all indexes open in a work area.
OrdListRebuild()	Rebuilds all indexes open in the current work area
OrdName()	Returns the symbolic name of an open index by its ordinal position.
OrdNumber()	Returns the ordinal position of an open index by its symbolic name.
OrdScope()	Defines the top and/or bottom scope for navigating in the controlling index.
OrdSetFocus()	Sets focus to the controlling index in a work area
OrdSetRelation()	Defines a scoped relation between child work area and parent work area.
OrdSkipRaw()	Moves the record pointer via the controlling index.
OrdSkipUnique()	Navigates to the next or previous record that has another index value.
OrdWildSeek()	Searches a value in the controlling index using wild card characters.
RecCount()	Returns the number of records available in a work area.
RecNo()	Retrieves the number of the current record in a work area.
RecSize()	Retrieves the number of bytes required to store a database record.
RLock()	Locks the current record for concurrent write access in a work area.
Select()	Retrieves the work area number by alias name.
SetNetDelay()	Queries or changes the timeout period for Net*() functions.
Used()	Determines if a database file is open in a work area.

Datagram functions

INetDGram()	Creates an unbound datagram oriented socket.
INetDGramBind()	Creates a bound datagram oriented socket.
INetDGramRecv()	Reads data from a datagram socket.
INetDGramSend()	Sends data to a datagram socket.

Date and time

AddMonth()	Adds or subtracts a number of months to/from a Date value.
BoM()	Returns the date of the first day of a month.
BoQ()	Returns the date of the first day of a quarter.
BoY()	Returns the date of the first day of a year.
CDoW()	Returns the name of a week day from a date.
CMonth()	Returns the name of a month from a date.
CtoD()	Converts a character string into a Date value

CtoDoW()	Returns the number of a week day from its name.
CtoMonth()	Returns the number of a month from its name.
CtoT()	Converts a character string into a DateTime value
Date()	Returns the current date from the operating system.
DateTime()	Returns the current date and time from the operating system.
Day()	Extracts the numeric day number from a Date value.
Days()	Calculates the number of days from elapsed seconds.
DaysInMonth()	Returns the number of days in a month.
DaysToMonth()	Returns the number of days from first January to the beginning of a month.
DMY()	Formats a date as "dd. Month yyyy"
DoW()	Determines the numeric day of the week from a date.
DoY()	Returns the day number of a Date value in a year.
DtoC()	Converts a Date value to a character string in SET DATE format.
DtoS()	Converts a Date value to a character string in YYYYMMDD format.
ElapTime()	Calculates the time elapsed between a start and an end time.
EoM()	Returns the date of the last day in a month.
EoQ()	Returns the date of the last day in a quarter.
EoY()	Returns the date for the last day of a year.
HB_Clocks2Secs()	Calculates seconds from CPU ticks.
Hour()	Extracts the hour from a DateTime value
IsLeap()	Checks if a Date value belongs to a leap year.
LastDayoM()	Returns the number of days in a month.
Max()	Returns the larger value of two Numerics or Dates.
MDY()	Formats a date as "Month dd, yy".
MilliSec()	Defines a time delay in milliseconds.
Min()	Returns the smallerr value of two Numerics or Dates.
Minute()	Extracts the minute from a DateTime value
Month()	Extracts the numeric month number from a Date value.
NtoCDoW()	Converts a numeric week day to its name.
NtoCMonth()	Converts a numeric month to its name.
Quarter()	Returns the quarter a date belongs to.
Seconds()	Returns the number of seconds elapsed since midnight
SecondsCpu()	Returns the CPU time used by the current process.
Secs()	Calculates the number of seconds from a time string.
SecToTime()	Converts numeric seconds into a time formatted character string.
SetDate()	Changes the system date from a Date value.
SetNewDate()	Changes the system date from Numeric values.
SetNewTime()	Changes the system time from Numeric values.
SetTime()	Changes the system time from a Time string.
ShowTime()	Displays the system time continuously at a specified screen position.
StoD()	Converts a "yyyymmdd" formatted string to a Date value
StoT()	Converts a "YYYYMMDDhhmmss.ccc" formatted string to a DateTime value
SX_DtoP()	Converts a Date value into a 3-byte character string.
SX_PtoD()	Unpacks a packed 3-byte date value.
Time()	Retrieves the system time as a formatted character string.
TimeToSec()	Calculates the number of seconds since midnight.
TimeValid()	Checks if a character string is a valid time string.
TString()	Converts numeric seconds into a time formatted character string.
TtoC()	Converts a DateTime value to a character string in SET DATE and SET TIME format.
TtoS()	Converts a Date value to a character string in YYYYMMDDhhmmss.ccc format.
WaitPeriod()	Defines a wait period and allows for time controlled loops.
Week()	Calculates the numeric calendar week from a date.
WoM()	Calculates the week number in a month.
Year()	Extracts the numeric year from a Date value

Debug functions

AltD()	Activates and/or invokes the "classic" debugger.
CStr()	Converts a value to a character string.
ErrorBlock()	Sets or retrieves the error handling code block.
ErrorLevel()	Sets the exit code of the xHarbour application.
HB_AParams()	Collects values of all parameters passed to a function, method or procedure.
HB_ArgC()	Returns the number of command line arguments.
HB_ArgCheck()	Checks if an internal switch is set on the command line.
HB_ArgString()	Retrieves the vale of an internal switch set on the command line.
HB_ArgV()	Retrieves the value of a command line argument.
HB_ArrayId()	Returns a unique identifier for an array or an object variable.
HB_BldLogMsg()	Converts a list of parameters into a text string.
HB_CmdArgArgV()	Returns the first command line argument (EXE file name).
HB_DumpVar()	Converts a value to a character string holding human readable text.
HB_GCAll()	Scans the memory and releases all garbage memory blocks.
HB_GCStep()	Invokes the garbage collector for one collection cycle.
HB_IsArray()	Tests if the value returned by an expression is an array.
HB_IsBlock()	Tests if the value returned by an expression is a Code block.
HB_IsByRef	Tests if a parameter is passed by reference.
HB_IsDate()	Tests if the value returned by an expression is a Date.
HB_IsDateTime()	Tests if the value returned by an expression is a DateTime value.
HB_IsHash()	Tests if the value returned by an expression is a Hash.
HB_IsLogical()	Tests if the value returned by an expression is a logical value.
HB_IsMemo()	Tests if the value returned by an expression is a Memo value.
HB_IsNIL()	Tests if the value returned by an expression is NIL.
HB_IsNull()	Tests if the value returned by an expression is empty.
HB_IsNumeric()	Tests if the value returned by an expression is a numeric value.
HB_IsObject()	Tests if the value returned by an expression is an object.
HB_IsPointer()	Tests if the value returned by an expression is a pointer.
HB_IsString()	Tests if the value returned by an expression is a character string.
HB_LogDateStamp()	Returns a character string holding a date stamp.
HB_ThisArray()	Retrieves an array or object from its pointer.
HB_XmlErrorDesc()	Retrieves a textual error description for XML file parsing errors.
Memory()	Returns memory statistics.
PCount()	Returns the number of passed parameters.
ProcFile()	Determines the current PRG source code file.
ProcLine()	Determines the current line number executed in a PRG source code file.
ProcName()	Determines the symbolic name of the currently executed function, method or procedure.
PValue()	Retrieves the value of a parameter passed to a function, method or procedure.
Throw()	Throws an exception.
TraceLog()	Traces and logs the contents of one or more variables.
Type()	Determines the data type of a macro expression.
Valtype()	Determines the data type of the value returned by an expression.

Declaration

ACCESS	Declares an ACCESS method of a class.
ANNOUNCE	Declaration of a module identifier name.
ASSIGN	Declares an ASSIGN method of a class.
ASSOCIATE CLASS	Defines a scalar class for a native data type.
CLASS	Declares the class function of a user-defined class.
CLASSDATA	Declares a class variable of a class.

CLASSMETHOD	Declares the symbolic name of a class method.
DATA	Declares an instance variable of a class.
DELEGATE	Declares a message to be directed to a contained object.
DESTRUCTOR	Declares a method to be called by the garbage collector.
ENABLE TYPE CLASS	Activates scalar classes for native data tyes.
ERROR HANDLER	Declares the method for error handling within a class.
EXIT PROCEDURE	Declares a procedure to execeute when a program terminates.
EXPORTED:	Declares the EXPORTED attribute for a group of member variables and/or methods.
EXTEND CLASS...WITH DATA	Adds a new member variable to an existing class.
EXTEND CLASS...WITH METHOD	Adds a new method to an existing class.
EXTERNAL	Declares the symbolic name of an external function or procedure for the linker.
FIELD	Declares a field variable
FUNCTION	Declares a function along with its formal parameters.
GLOBAL	Declares and optionally initializes a GLOBAL memory variable.
HIDDEN:	Declares the HIDDEN attribute for a group of member variables and/or methods.
INIT PROCEDURE	Declares a procedure to execeute when a program starts.
INLINE METHOD	Declares and implements an inline method that spans across multiple lines.
LOCAL	Declares and optionally initializes a local memory variable.
MEMVAR	Declares PRIVATE or PUBLIC variables.
MESSAGE	Declares a message name for a method.
METHOD (declaration)	Declares the symbolic name of a method.
METHOD (implementation)	Declares the implementation code of a method.
METHOD...OPERATOR	Declares a method to be executed with an operator.
METHOD...VIRTUAL	Declares a method as virtual.
OPERATOR	Overloads an operator and declares a method to be invoked for it.
OVERRIDE METHOD	Replaces a method in an existing class.
PARAMETERS	Declares PRIVATE function parameters.
PRIVATE	Creates and optionally initializes a PRIVATE memory variable.
PROCEDURE	Declares a procedure along with its formal parameters.
PROTECTED:	Declares the PROTECTED attribute for a group of member variables and methods.
PUBLIC	Creates and optionally initializes a PUBLIC memory variable.
REQUEST	Declares the symbolic name of an external function or procedure for the linker.
STATIC	Declares and optionally initializes a STATIC memory variable.
VAR	Declares an instance variable of a class.

Directory functions

CurDir()	Returns the current directory of a drive
CurDirX()	Returns the current directory of a drive including directory separators.
DefPath()	Returns the SET DEFAULT directory.
DeleteFile()	Deletes a file with error handling.
DirChange()	Changes the current directory.
Directory()	Loads file and directory information into a two-dimensional array.
DirectoryRecurse()	Loads file information recursively into a two-dimensional array.
DirMake()	Creates a directory.
DirName()	Returns the current directory.
DirRemove()	Removes a directory.
DiskChange()	Changes the current disk drive.
DiskName()	Returns the current disk drive.
IsDir()	Checks if a character string contains the name of an existing directory.
IsDirectory()	Checks if a character string contains the name of an existing directory.
MakeDir()	Creates a new directory.

TrueName()	Completes a relative path to include the root directory.

Disks and Drives

CurDrive()	Determines or changes the current disk drive
DeleteFile()	Deletes a file with error handling.
DirMake()	Creates a directory.
DirName()	Returns the current directory.
DiskFormat()	Formats a floppy disk.
DiskFree()	Returns the free storage space of a disk drive in bytes.
DiskReady()	Test if a disk drive is ready.
DiskReadyW()	Tests if a drive can be written to.
DiskSpace()	Returns the free storage space for a disk drive.
DiskTotal()	Returns the totl storage space of a disk drive in bytes.
DiskUsed()	Returns the used storage space of a disk drive in bytes.
DriveType()	Determines the type of a drive.
FloppyType()	Determines the type of a floppy drive.
GetVolInfo()	Retrieves the volume label of a disk.
HB_DiskSpace()	Returns the free storage space for a disk drive by drive letter.
HB_OsDriveSeparator()	Returns the operating specific drive separator character.
IsDir()	Checks if a character string contains the name of an existing directory.
IsDisk()	Verify if a drive is ready
NumDiskL()	Returns the number of logical drives.
VolSerial()	Returns the serial number of a disk drive.
Volume()	Sets the volume label of a disk.

DLL functions

CallDll()	Executes a function located in a dynamically loaded external library.
DllCall()	Executes a function located in a dynamically loaded external library.
DllExecuteCall()	Executes a DLL function via call template.
DllLoad()	Loads a DLL file into memory.
DllPrepareCall()	Creates a call template for an external DLL function.
DllUnload()	Releases a dynamically loaded external DLL from memory.
FreeLibrary()	Releases a dynamically loaded external DLL from memory.
GetLastError()	Retrieves the error code of the last dynamically called DLL function.
GetProcAddress()	Retrieves the memory address of a function in a dynamically loaded DLL.
HB_LibDo()	Executes a function located in a dynamically loaded xHarbour DLL.
LibFree()	Releases a dynamically loaded xHarbour DLL from memory.
LibLoad()	Loads an xHarbour DLL file into memory.
LoadLibrary()	Loads an external DLL file into memory.
SetLastError()	Sets a numeric value as last error code.

Encoding/Decoding

HB_Base64Decode()	Decodes a base 64 encoded character string.
HB_Base64DecodeFile()	Decodes a base 64 encoded file.
HB_Base64Encode()	Encodes a character string base 64.
HB_Base64EncodeFile()	Encodes a file base 64.
HB_UUDecode()	Decodes a UUEncoded character string.
HB_UUDecodeFile()	Decodes a UUEncoded file.
HB_UUEncode()	UUEncodes a character string.
HB_UUEncodeFile()	UUEncodes a file.

Environment commands

CANCEL	Terminates program execution unconditionally.
CLOSE	Closes one or more specified files.
SET BELL	Toggles automatic sounding of the bell in the GET system.
SET DIRCASE	Specifies how directories are accessed on disk.
SET DIRSEPARATOR	Specifies the default separator for directories.
SET EOL	Defines the end-of-line character(s) for ASCII text files.
SET ERRORLOG	Defines the default error log file.
SET ERRORLOOP	Defines the maximum recursion depth for error handling.
SET FILECASE	Specifies how files are accessed on disk.
SET FIXED	Toggles fixed formatting for displaying numbers in text-mode.
SET TRACE	Toggles output of function TraceLog().

Environment functions

CSetCent()	Queries or changes the SET CENTURY setting.
CSetCurs()	Queries or changes the SET CURSOR setting.
CSetKey()	Retrieves the code block associated with a key.
DosParam()	Returns the command line parameters passed to an xHarbour application.
ExeName()	Returns the EXE file name of an xHarbour application.
GetE()	Retrieves an operating system environment variable.
GetEnv()	Retrieves an operating system environment variable.
HB_AParams()	Collects values of all parameters passed to a function, method or procedure.
HB_ArgC()	Returns the number of command line arguments.
HB_ArgCheck()	Checks if an internal switch is set on the command line.
HB_ArgString()	Retrieves the vale of an internal switch set on the command line.
HB_ArgV()	Retrieves the value of a command line argument.
HB_BuildDate()	Retrieves the formatted build date of the xHarbour compiler
HB_BuildInfo()	Retrieves build information of the xHarbour compiler.
HB_CmdArgArgV()	Returns the first command line argument (EXE file name).
HB_Compiler()	Retrieves the version of the C compiler shipped with xHarbour.
HB_EnumIndex()	Returns the current ordinal position of a FOR EACH iteration.
HB_GCAll()	Scans the memory and releases all garbage memory blocks.
HB_GCStep()	Invokes the garbage collector for one collection cycle.
HB_OsDriveSeparator()	Returns the operating specific drive separator character.
HB_OsNewLine()	Returns the end-of-line character(s) to use with the current operating system.
HB_OsPathDelimiters()	Returns the operating specific characters for paths.
HB_OsPathListSeparator()	Returns the operating specific separator character for a path list.
HB_OsPathSeparator()	Returns the operating specific separator character for a path.
HB_PCodeVer()	Retrieves the PCode version of the current xHarbour build.
HB_SetKeyArray()	Associates a code block with multiple keys.
HB_SetKeyCheck()	Evaluates a code block associated with a key.
HB_SetKeyGet()	Retrieves code blocks associated with a key.
HB_SetKeySave()	Queries or changes all SetKey() code blocks.
HB_VMMode()	Indicates the creation mode of the xHarbour virtual machine.
Memory()	Returns memory statistics.
NetName()	Retrieves the current user name or the computer name.
Os()	Returns the name of the operating system.
Os_IsWin2000()	Checks if the application is running on Windows 2000.
Os_IsWin2000_Or_Later()	Checks if the application is running on Windows version 2000 or later
Os_IsWin2003()	Checks if the application is running on Windows 2003.
Os_IsWin95()	Checks if the application is running on Windows 95.
Os_IsWin98()	Checks if the application is running on Windows 98.

Os_IsWin9X()	Checks if the application is running on a Windows 9x platform.
Os_IsWinME()	Checks if the application is running on Windows ME.
Os_IsWinNT()	Checks if the application is running on a Windows NT platform.
Os_IsWinNT351()	Checks if the application is running on Windows NT 3.51.
Os_IsWinNT4()	Checks if the application is running on Windows NT 4.0.
Os_IsWinVista()	Checks if the application is running on Windows Vista.
Os_IsWinXP()	Checks if the application is running on Windows XP.
Os_IsWtsClient()	Checks if the application is running on a Windows Terminal Server client.
Os_VersionInfo()	Retrieves specific version information about the operating system.
PCol()	Returns the column position of the print head.
PCount()	Returns the number of passed parameters.
PRow()	Returns the row position of the print head.
PValue()	Retrieves the value of a parameter passed to a function, method or procedure.
Set()	Retrieves or changes a system setting.
SetCancel()	Determines if Alt+C and Ctrl+Break terminate an application
SetDate()	Changes the system date from a Date value.
SetKey()	Associates a code block with a key.
SetPrc()	Changes the PRow() and PCol() values.
Tone()	Sounds a speaker tone with specific tone frequency and duration.
Used()	Determines if a database file is open in a work area.
Version()	Retrieves xHarbour version information.

Error functions

Break()	Exits from a BEGIN SEQUENCE block.
DosError()	Sets or retrieves the last DOS error code.
ErrorBlock()	Sets or retrieves the error handling code block.
ErrorLevel()	Sets the exit code of the xHarbour application.
ErrorNew()	Creates a new Error object and optionally fills it with data.
ErrorSys()	Installs the default error code block.
HB_LangErrMsg()	Returns language specific error messages.
HB_OsError()	Returns the operating specific error code of the last low-level file operation.
SetErrorMode()	Queries or changes the behavior with operating system errors.
Throw()	Throws an exception.

Field functions

AFields()	Fills pre-allocated arrays with structure information of the current work area.
DbFieldInfo()	Retrieves information about a database field
DbFileGet()	Copies data from a field into an external file.
DbFilePut()	Copies the contents of an external file into a field.
FCount()	Returns the number of fields in the current work area.
FieldDec()	Retrieves the number of decimal places of a field variable.
FieldDeci()	Returns the decimal places of a database field.
FieldGet()	Retrieves the value of a field variable by its ordinal position.
FieldLen()	Retrieves the number of bytes occupied by a field variable.
FieldName()	Retrieves the name of a field variable by its ordinal position.
FieldNum()	Returns the ordinal position of a field in a database.
FieldPos()	Retrieves the ordinal position of a field variable by its name.
FieldPut()	Assigns a value to a field variable by its ordinal position.
FieldSize()	Returns the length of a database field.
FieldType()	Retrieves the data type of a field variable.

File commands

COPY FILE	Copies a file to a new file or to an output device.
DELETE FILE	Deletes a file from disk.
ERASE	Deletes a file from disk.
RENAME	Changes the name of a file.

File functions

ADir()	Fills pre-allocated arrays with file and/or directory information.
CSetSafety()	Retrieves and/or changes the safety switch used in CA-Tools file operations.
CurDir()	Returns the current directory of a drive
CurDirX()	Returns the current directory of a drive including directory separators.
CurDrive()	Determines or changes the current disk drive
DefPath()	Returns the SET DEFAULT directory.
DirChange()	Changes the current directory.
Directory()	Loads file and directory information into a two-dimensional array.
DirectoryRecurse()	Loads file information recursively into a two-dimensional array.
DirRemove()	Removes a directory.
DisableWaitLocks()	Toggles the exclusive file opening mode.
DiskChange()	Changes the current disk drive.
DiskName()	Returns the current disk drive.
DiskSpace()	Returns the free storage space for a disk drive.
FCharCount()	Counts the number of characters in a text file ignoring white space characters.
FClose()	Closes a binary file.
FCount()	Returns the number of fields in the current work area.
FCreate()	Creates an empty binary file.
FErase()	Deletes a file from disk.
FError()	Retrieves the error code of the last low-level file operation.
File()	Checks for the existence of a file in the default directory or path
FileAppend()	Concatenates two files.
FileAttr()	Returns the attributes of a file.
FileCClose()	Closes the file opened in backup mode with FileCopy().
FileCCont()	Continues file copying in backup mode of FileCopy().
FileCOpen()	Tests if the file opened in backup mode with FileCopy() is still open.
FileCopy()	Copies files normally or in backup mode.
FileDate()	Returns the date of a file.
FileDelete()	Deletes one or more files specified by a file mask and file attributes.
FileMove()	Moves a file to another directory.
FileSeek()	Seeks files specified by a file mask and file attributes.
FileSize()	Returns the size of a file.
FileStats()	Retrieves file information for a single file.
FileStr()	Reads a string from a file beginning at a specified offset.
FileTime()	Returns the change time of a file.
FileValid()	Tests if a string contains a valid file name.
FLineCount()	Counts the lines in an ASCII text file.
FOpen()	Opens a file on the operating system level.
FParse()	Parses a delimited text file and loads it into an array.
FParseEx()	Parses a delimited text file and loads it into an array (optimized).
FParseLine()	Parses one line of a delimited text and loads it into an array.
FRead()	Reads characters from a binary file into a memory variable.
FReadStr()	Reads characters up to Chr(0) from a binary file.
FRename()	Renames a file.
FSeek()	Changes the position of the file pointer.

FWordCount()	Counts the words in a text file.
FWrite()	Writes data to an open binary file.
HB_FCommit()	Forces a disk write.
HB_FCreate()	Creates and/or opens a binary file.
HB_FEof()	Tests if the end-of-file is reached in the currently selected text file.
HB_FGoBottom()	Moves the file pointer to the last line in a text file.
HB_FGoto()	Moves the record pointer to a specific line in the currently selected text file.
HB_FGoTop()	Moves the record pointer to the begin-of-file.
HB_FInfo()	Retrieves status information about the currently selected text file.
HB_FLastRec()	Returns the number of lines in the currently selected text file.
HB_FNameMerge()	Composes a file name from individual components.
HB_FNameSplit()	Splits a file name into individual components.
HB_FReadAndSkip()	Reads the current line and moves the record pointer.
HB_FreadLN()	Reads the current line and without moving the record pointer.
HB_FRecno()	Returns the current line number of the currently selected text file.
HB_FSelect()	Queries or changes the currently selected text file area.
HB_FSkip()	Moves the record pointer in the currently selected text file.
HB_FTempCreate()	Creates and opens a temporary file.
HB_FUse()	Opens or closes a text file in a text file area.
HB_F_Eof()	Tests for End-of-file with binary files.
HB_ReadIni()	Reads an INI file from disk.
HB_SetIniComment()	Defines the delimiting characters for comments and inline comments.
HB_WriteIni()	Creates an INI file and writes hash data to it.
IsDirectory()	Checks if a character string contains the name of an existing directory.
IsDisk()	Verify if a drive is ready
MakeDir()	Creates a new directory.
RenameFile()	Renames a file and handles errors.
SetFAttr()	Sets file attributes.
SetFCreate()	Sets the default file attribute(s) for creating files.
SetFDaTi()	Sets the last change date and time of a file.
StrFile()	Writes a string to a file starting at a specified position.
SX_FCompress()	Compresses a file.
SX_FDecompress()	Decompresses a compressed file.

Financial functions

Fv()	Calculates the future value of constant, periodic investments.
Payment()	Calculates a periodical payment for loans.
Periods()	Calculates the duration for paying back a loan at fixed interest rate and payments.
Pv()	Calculates the present value of future capital, based on periodic investments.
Rate()	Calculates the interest rate for a loan.

Get system

@...GET	Creates a Get object (entry field) and displays it to the screen
@...GET CHECKBOX	Creates a Get object as check box and displays it to the screen.
@...GET LISTBOX	Creates a Get object as list box and displays it to the screen.
@...GET PUSHBUTTON	Creates a Get object as push button and displays it to the screen.
@...GET RADIOGROUP	Creates a Get object as radio button group and displays it to the screen.
@...GET TBROWSE	Creates a Get object as browser and displays it to the screen
CLEAR GETS	Releases all Get objects in the current GetList array.
CountGets()	Returns the number of Get fields in the current Getlist array.
CurrentGet()	Returns the position of the current Get field in the Getlist array.
Get()	Creates a new Get object.

GetActive()	Returns or assigns the current Get object during READ.
GetApplyKey()	Sends an Inkey() code to the currently active Get object.
GetDoSetKey()	Evaluates a SetKey() code block during READ.
GetFldCol()	Returns the screen column position of a Get field.
GetFldRow()	Returns the screen row position of a Get field.
GetFldVar()	Returns the name of a Get variable.
GetPostValidate()	Validates data after editing.
GetPreValidate()	Validates data before editing.
GetSecret()	Allows for hidden Get input for character strings.
GuiApplyKey()	Sends an Inkey() code to the associated control of the currently active Get object.
GuiGetPostValidate()	Validates data after editing.
GuiGetPrevalidate()	Validates data before editing.
GuiReader()	Processes user input.
HbCheckBox()	Creates a new HbCheckBox object.
HbListBox()	Creates a new HbListBox object
HbPushButton()	Creates a new HbPushButton object.
HbRadioButton()	Creates a new HRadioButton object.
HbRadioGroup()	Creates a new HbRadioGroup object.
HbScrollBar()	Creates a new HbScrollBar object.
IsDefColor()	Checks if the default color is set.
MenuItem()	Creates a new MenuItem object.
MenuModal()	Activates the menu system represented by a TopBarMenu object.
Popup()	Creates a new Popup menu object.
READ	Activates editing of @...GET entry fields in text mode.
ReadExit()	Enables or disables up/down arrow keys as exit keys for READ
ReadInsert()	Queries or changes the insert/overstrike mode during READ.
ReadKey()	Returns the key code of the key that has terminated READ.
ReadKill()	Queries or changes the READ termination flag.
ReadModal()	Activates editing of @...GET entry fields in text mode.
ReadUpdated()	Queries or changes the updated flag of the READ command.
ReadVar()	Returns name of the current GET or MENU TO variable.
RestGets()	Restores a previously saved !IGetlist!EI array.
RestSetKey()	Restores SetKey() settings and associated code blocks.
SaveGets()	Saves the current !IGetlist!EI array for nested READs.
SaveSetKey()	Saves SetKey() settings and associated code blocks.
TopBarMenu()	Creates a new TopBarMenu object.
Updated()	Queries the updated flag of the READ command.

Hash functions

HaaDelAt()	Removes a key/value pair from an associative array.
HaaGetKeyAt()	Retrieves the key from an associative array by its ordinal position.
HaaGetPos()	Retrieves the ordinal position of a key in an associative array.
HaaGetRealPos()	Retrieves the sort order of a key in an associative array.
HaaGetValueAt()	Retrieves the value from an associative array by its ordinal position.
HaaSetValueAt()	Changes the value in an associative array by its ordinal position.
HAllocate()	Pre-allocates memory for a large hash.
Hash()	Creates a new hash.
HClone()	Creates an entire copy of a hash.
HCopy()	Copies key/value pairs from a hash into another hash.
HDel()	Removes a key/value pair from the hash by its key.
HDelAt()	Removes a key/value pair from the hash by its ordinal position.
HEval()	Evaluates a code block with each hash element.
HFill()	Copies the same value into all key/value pairs.
HGet()	Retrieves the value associated with a specified key.
HGetAACompatibility()	Checks if a hash is compatible with an associative array.
HGetAutoAdd()	Retrieves the AutoAdd attribute of a hash.

HGetCaseMatch()	Retrieves the case sensitivity attribute of a hash.
HGetKeyAt()	Retrieves the key from a hash by its ordinal position.
HGetKeys()	Collects all keys from a hash in an array.
HGetPairAt()	Retrieves a key/value pair from a hash by its ordinal position.
HGetPartition()	Checks if a hash is partitioned.
HGetPos()	Retrieves the ordinal position of a key in a hash.
HGetVaaPos()	Retrieves the sort order of all keys in an associative array.
HGetValueAt()	Retrieves the value from a hash by its ordinal position.
HGetValues()	Collects all values from a hash in an array.
HHasKey()	Determines if a key is present in a hash.
HMerge()	Merges the contents of an entire hash into another hash.
HScan()	Searches a value in a hash.
HSet()	Associates a value with a key in a hash.
HSetAACompatibility()	Enables or disables associative array compatibility for an empty hash.
HSetAutoAdd()	Changes the AutoAdd attribute of a hash.
HSetCaseMatch()	Changes the case sensitivity attribute of a hash.
HSetPartition()	Partitions a linear hash for improved performance.
HSetValueAt()	Changes the value in a hash by its ordinal position.
Len()	Returns the number of items contained in an array, hash or string

HiPer-SEEK functions

HS_Add()	Adds a text string entry to a HiPer-SEEK index file.
HS_Close()	Closes a HiPer-SEEK index file.
HS_Create()	Creates a new HiPer-SEEK index file.
HS_Delete()	Marks an index entry as deleted in a HiPer-SEEK index file.
HS_Filter()	Uses a HiPer-SEEK index file as filter for a work area
HS_IfDel()	Checks if a HiPer-SEEK index entry is marked as deleted
HS_Index()	Creates a new HiPer-SEEK index file and fills it with index entries.
HS_KeyCount()	Returns the number of index entries in a HiPer-SEEK index file.
HS_Next()	Searches a HiPer-SEEK index file for a matching index entry.
HS_Open()	Opens a HiPer-SEEK index file.
HS_Replace()	Changes a HiPer-SEEK index entry.
HS_Set()	Defines a search string for subsequent HS_Next() calls.
HS_Undelete()	Removes the deletion mark from an index entry in a HiPer-SEEK index file.
HS_Verify()	Verifies a HS_Next() match against the index key.
HS_Version()	Returns version information for HiPer-SEEK functions.

HTML functions

AnsiToHtml()	Inserts HTML character entities into an ANSI text string,
HtmlToAnsi()	Converts an HTML formatted text string to the ANSI character set.
HtmlToOem()	Converts an HTML formatted text string to the OEM character set
OemToHtml()	Inserts HTML character entities into an OEM text string.
THtmlCleanup()	Releases memory tables required for HTML classes.
THtmlDocument()	Creates a new THtmlDocument object.
THtmlInit()	Initializes memory tables required for HTML classes.
THtmlIsValid()	Validates a HTML tag name and attribute.
THtmlIterator()	Creates a new THtmlIterator object.
THtmlIteratorRegEx()	Creates a new THtmlIteratorRegEx object.
THtmlIteratorScan()	Creates a new THtmlIteratorScan object.
THtmlNode()	Creates a new THtmlNode object.
TIpClientHttp()	Creates a new TIpClientHttp object.

Index commands

DELETE TAG	Deletes a tag from an index.
INDEX	Creates an index and/or index file.
REINDEX	Rebuilds all open indexes in the current work area.
SEEK	Searches a value in an index.
SET DESCENDING	Changes the descending flag of the controlling index at runtime.
SET INDEX	Opens one or more index files in the current work area.
SET ORDER	Selects the controlling index.
SET SCOPE	Changes the top and/or bottom scope for navigating in the controlling index.
SET SCOPEBOTTOM	Changes the bottom scope for navigating in the controlling index.
SET SCOPETOP	Changes the top scope for navigating in the controlling index.
SET SOFTSEEK	Enables or disables relative seeking.
SET UNIQUE	Includes or excludes non-unique keys to/from an index.

Index functions

DbClearIndex()	Closes all indexes open in the current work area.
DbCreateIndex()	Creates a new index and/or index file.
DbReindex()	Rebuilds indexes open in a work area.
DbSeek()	Searches a value in the controlling index.
DbSetIndex()	Opens an index file.
DbSetOrder()	Selects the controlling index.
Found()	Checks if the last database search operation was successful
HS_Add()	Adds a text string entry to a HiPer-SEEK index file.
HS_Close()	Closes a HiPer-SEEK index file.
HS_Create()	Creates a new HiPer-SEEK index file.
HS_Delete()	Marks an index entry as deleted in a HiPer-SEEK index file.
HS_Filter()	Uses a HiPer-SEEK index file as filter for a work area
HS_IfDel()	Checks if a HiPer-SEEK index entry is marked as deleted
HS_Index()	Creates a new HiPer-SEEK index file and fills it with index entries.
HS_KeyCount()	Returns the number of index entries in a HiPer-SEEK index file.
HS_Next()	Searches a HiPer-SEEK index file for a matching index entry.
HS_Open()	Opens a HiPer-SEEK index file.
HS_Replace()	Changes a HiPer-SEEK index entry.
HS_Set()	Defines a search string for subsequent HS_Next() calls.
HS_Undelete()	Removes the deletion mark from an index entry in a HiPer-SEEK index file.
HS_Verify()	Verifies a HS_Next() match against the index key.
HS_Version()	Returns version information for HiPer-SEEK functions.
IndexExt()	Retrieves the default index file extension in a work area.
IndexKey()	Returns the index expression of an open index.
IndexOrd()	Returns the ordinal position of the controlling index in a work area.
OrdBagExt()	Retrieves the default extension for index files.
OrdBagName()	Retrieves the file name of an open index.
OrdCondSet()	Sets the conditions for index creation.
OrdCount()	Determines the number of indexes open in a work area.
OrdCreate()	Creates a new index.
OrdCustom()	Determines if an index is a custom index.
OrdDescend()	Determines the navigational order of a work area.
OrdDestroy()	Deletes an index from an index file.
OrdFindRec()	Searches a record number in the controlling index.
OrdFor()	Retrieves the FOR expression of an index.
OrdIsUnique()	Retrieves the UNIQUE flag of an index.
OrdKey()	Returns the key expression of an index.

OrdKeyAdd() Adds an index key to a custom built index.
OrdKeyCount() Retrieves the total number of keys included in an index.
OrdKeyDel() Removes an index key from a custom built index.
OrdKeyGoTo() Moves the record pointer to a logical record number.
OrdKeyNo() Returns the logical record number of the current record.
OrdKeyRelPos() Returns or sets the relative position of the current record.
OrdKeyVal() Retrieves the index value of the current record from the controlling index.
OrdListAdd() Opens and optionally activates a new index in a work area.
OrdListClear() Closes all indexes open in a work area.
OrdListRebuild() Rebuilds all indexes open in the current work area
OrdName() Returns the symbolic name of an open index by its ordinal position.
OrdNumber() Returns the ordinal position of an open index by its symbolic name.
OrdScope() Defines the top and/or bottom scope for navigating in the controlling index.
OrdSetFocus() Sets focus to the controlling index in a work area
OrdSetRelation() Defines a scoped relation between child work area and parent work area.
OrdSkipRaw() Moves the record pointer via the controlling index.
OrdSkipUnique() Navigates to the next or previous record that has another index value.
OrdWildSeek() Searches a value in the controlling index using wild card characters.

Indirect execution

& (macro operator) Macro operator (unary): compiles a character string at runtime.
<|| > Extended literal code block.
@() Function-reference operator (unary): obtains a function pointer.
Eval() Evaluates a code block.
HB_AExpressions() Parses a character string into an array of macro expressions.
HB_Exec() Executes a function, procedure or method from its pointer.
HB_ExecFromArray() Executes a function, procedure or method indirectly.
HB_FuncPtr() Obtains the pointer to a function or procedure.
HB_MacroCompile() Compiles a macro string into a PCode sequence.
HB_ObjMsgPtr() Retrieves the pointer to a method.
HB_SetMacro() Enables or disables runtime behavior of the macro compiler.
HB_VMExecute() Executes a PCode string.
{|| } Literal code block.

Info functions

DbInfo() Queries and/or changes information about a database file open in a work area.
DbOrderInfo() Queries and/or changes information about indexes open in a work area.
DbRecordInfo() Queries and/or changes information about a record of an open database file.
RddInfo() Queries and/or changes configuration data of RDDs.

Input commands

@...GET Creates a Get object (entry field) and displays it to the screen
@...GET CHECKBOX Creates a Get object as check box and displays it to the screen.
@...GET LISTBOX Creates a Get object as list box and displays it to the screen.
@...GET PUSHBUTTON Creates a Get object as push button and displays it to the screen.
@...GET RADIOGROUP Creates a Get object as radio button group and displays it to the screen.
@...GET TBROWSE Creates a Get object as browser and displays it to the screen
@...PROMPT Displays a menu item for a text mode menu.
ACCEPT Basic user input routine.
CLEAR TYPEAHEAD Empties the keyboard buffer.

INPUT	Assigns the result of an input expression to a variable.
KEYBOARD	Writes a string or numeric key codes into the keyboard buffer.
MENU TO	Activates a text-mode menu defined with @...PROMPT commands.
READ	Activates editing of @...GET entry fields in text mode.
SET CONFIRM	Determines how a GET entry field is exited.
SET EVENTMASK	Sets which events should be returned by the Inkey() function.
SET FUNCTION	Associates a character string with a function key.
WAIT	Suspend program execution until a key is pressed.

Internet functions

INetAccept()	Waits for an incoming connection on a server side socket.
INetAddress()	Determines the internet address of a remote station.
INetCleanup()	Releases memory resources for sockets.
INetClearError()	Resets the last error code of a socket.
INetClearPeriodCallback()	Voids a callback function for a socket.
INetClearTimeout()	Voids a timout value for a socket.
INetClose()	Closes a connection on a socket.
INetConnect()	Establishes a sockets connection to a server.
INetConnectIP()	Establishes a sockets connection to a server using the IP address.
INetCount()	Returns the number of bytes transferred in the last sockets operation
INetCreate()	Creates a raw, unconnected socket.
INetCRLF()	Returns new line characters used in internet communications.
INetDataReady()	Tests if incoming data is available to be read.
INetErrorCode()	Returns the last sockets error code.
INetErrorDesc()	Returns a descriptive error message
INetGetAlias()	Retrieves alias names from a server.
INetGetHosts()	Queries IP addresses associated with a name.
INetGetPeriodCallback()	Queries a callback associated with a socket.
INetGetTimeout()	Queries a timout value for a socket.
INetInit()	Initializes the sockets subsystem.
INetPort()	Determines the port number a socket is connected to.
INetRecv()	Reads data from a socket.
INetRecvAll()	Reads all data from a socket.
INetRecvEndblock()	Reads one block of data until an end-of-block marker is detected.
INetRecvLine()	Reads one line of data until CRLF is detected.
INetSend()	Sends data to a socket.
INetSendAll()	Sends all data to a socket.
INetServer()	Creates a server side socket.
INetSetPeriodCallback()	Associates callback information with a socket.
INetSetTimeout()	Sets a timout value in milliseconds for a socket.
TIpClient()	Abstract class for internet communication.
TIpClientFtp()	Creates a new TIpClientFtp object.
TIpClientHttp()	Creates a new TIpClientHttp object.
TIpClientPop()	Creates a new TIpClientPop object.
TIpClientSmtp()	Creates a new TIpClientSmtp object.
TIpMail()	Creates a new TIpMail object.
TUrl()	Creates a new TUrl object.

Keyboard functions

FkLabel()	Returns a function key name.
FkMax()	Returns the number of available function keys.
HB_KeyPut()	Puts an inkey code into the keyboard buffer.
HB_SetKeyArray()	Associates a code block with multiple keys.
HB_SetKeyCheck()	Evaluates a code block associated with a key.
HB_SetKeyGet()	Retrieves code blocks associated with a key.

HB_SetKeySave()	Queries or changes all SetKey() code blocks.
Inkey()	Retrieves a character from the keyboard buffer or a mouse event.
KbdStat()	Determines the state of special keys like Ctrl or Shift keys.
KeySec()	Starts a timer to write a key code into the keyboard buffer.
KeyTime()	Writes a key code into the keyboard buffer at a specified time.
KSetCaps()	Queries or changes the status of the Caps lock key
KSetIns()	Queries or changes the status of the Insert/Overwrite key
KSetNum()	Queries or changes the status of the Num lock key
KSetScroll()	Queries or changes the status of the Scroll lock key
LastKey()	Returns the last Inkey() code retrieved.
NextKey()	Returns the next pending key or mouse event.
SetKey()	Associates a code block with a key.
SetLastKey()	Changes the return value of function LastKey()

Language specific

HB_LangErrMsg()	Returns language specific error messages.
HB_LangMessage()	Returns language specific messages or text strings.
HB_LangName()	Returns the currently selected language.
HB_LangSelect()	Queries or changes the current national language .
HB_SetCodePage()	Queries or changes the current code page.
HB_Translate()	Converts a character string from one code page to another one.
IsAffirm()	Converts "Yes" in a national language to a logical value.
IsNegative()	Converts "No" in a national language to a logical value.
NationMsg()	Returns application specific messages.

Log commands

CLOSE LOG	Closes all open log channels.
INIT LOG	Initializes the Log system and opens requested log channels.
LOG	Sends a message to open log channels.
SET LOG STYLE	Selects a style for logging data to log channels.

Log functions

HB_BldLogMsg()	Converts a list of parameters into a text string.
HB_LogDateStamp()	Returns a character string holding a date stamp.
HB_SysLogClose()	Closes the log file of the operating system.
HB_SysLogMessage()	Adds a new entry to the log file of the operating system.
HB_SysLogOpen()	Opens the log file of the operating system
TraceLog()	Traces and logs the contents of one or more variables.

Logical functions

Empty()	Checks if the passed argument is empty.
HB_IsArray()	Tests if the value returned by an expression is an array.
HB_IsBlock()	Tests if the value returned by an expression is a Code block.
HB_IsByRef	Tests if a parameter is passed by reference.
HB_IsDate()	Tests if the value returned by an expression is a Date.
HB_IsDateTime()	Tests if the value returned by an expression is a DateTime value.
HB_IsHash()	Tests if the value returned by an expression is a Hash.
HB_IsLogical()	Tests if the value returned by an expression is a logical value.
HB_IsMemo()	Tests if the value returned by an expression is a Memo value.
HB_IsNIL()	Tests if the value returned by an expression is NIL.
HB_IsNull()	Tests if the value returned by an expression is empty.

HB_IsNumeric()	Tests if the value returned by an expression is a numeric value.
HB_IsObject()	Tests if the value returned by an expression is an object.
HB_IsPointer()	Tests if the value returned by an expression is a pointer.
HB_IsString()	Tests if the value returned by an expression is a character string.
If() | IIf()	Returns the result of an expression based on a logical expression

Logical operators

.AND.	Logical AND operator (binary).
.NOT.	Logical NOT operator (unary).
.OR.	Logical OR operator (binary).

Low level file functions

DisableWaitLocks()	Toggles the exclusive file opening mode.
FClose()	Closes a binary file.
FCreate()	Creates an empty binary file.
FErase()	Deletes a file from disk.
FError()	Retrieves the error code of the last low-level file operation.
FileAppend()	Concatenates two files.
FileAttr()	Returns the attributes of a file.
FileCClose()	Closes the file opened in backup mode with FileCopy().
FileCCont()	Continues file copying in backup mode of FileCopy().
FileCOpen()	Tests if the file opened in backup mode with FileCopy() is still open.
FileCopy()	Copies files normally or in backup mode.
FileDate()	Returns the date of a file.
FileDelete()	Deletes one or more files specified by a file mask and file attributes.
FileMove()	Moves a file to another directory.
FileSeek()	Seeks files specified by a file mask and file attributes.
FileSize()	Returns the size of a file.
FileStats()	Retrieves file information for a single file.
FileStr()	Reads a string from a file beginning at a specified offset.
FileTime()	Returns the change time of a file.
FileValid()	Tests if a string contains a valid file name.
FOpen()	Opens a file on the operating system level.
FRead()	Reads characters from a binary file into a memory variable.
FReadStr()	Reads characters up to Chr(0) from a binary file.
FRename()	Renames a file.
FSeek()	Changes the position of the file pointer.
FWrite()	Writes data to an open binary file.
HB_FCommit()	Forces a disk write.
HB_FCreate()	Creates and/or opens a binary file.
HB_FReadLine()	Extracts the next line from a text file.
HB_FSize()	Returns the size of a file in bytes.
HB_FTempCreate()	Creates and opens a temporary file.
HB_F_Eof()	Tests for End-of-file with binary files.
HB_OsError()	Returns the operating specific error code of the last low-level file operation.
SetFAttr()	Sets file attributes.
SetFCreate()	Sets the default file attribute(s) for creating files.
SetFDaTi()	Sets the last change date and time of a file.
StrFile()	Writes a string to a file starting at a specified position.

Mathematical functions

ACos()	Calculates the arc cosine.

ASin()	Calculates the arc sine.
ATan()	Calculates the arc tangent.
ATn2()	Calculates the radians of an angle from sine and cosine.
Ceiling()	Rounds a decimal number to the next higher integer.
Cos()	Calculates the cosine for an angle.
CosH()	Calculates the hyperbolic cosine for an angle.
Cot()	Calculates the cotangent.
DtoR()	Converts an angle from degrees to radians.
Exp()	Calculates the value of e raised by an exponent.
Fact()	Calculates the factorial of a number.
Floor()	Rounds a decimal number to the next lower integer.
Fv()	Calculates the future value of constant, periodic investments.
GetPrec()	Retrieves computing precision for trigonometric functions.
Log()	Calculates the natural logarithm of a numeric value.
Log10()	Calculates the base 10 logarithm.
Payment()	Calculates a periodical payment for loans.
Periods()	Calculates the duration for paying back a loan at fixed interest rate and payments.
Pi()	Returns Pi with highest accuracy.
Pv()	Calculates the present value of future capital, based on periodic investments.
Rate()	Calculates the interest rate for a loan.
RtoD()	Converts angles from radians to degrees.
SetPrec()	Specifies the computing precision for trigonometric functions.
Sign()	Converts the sign of a number to a numeric value.
Sin()	Calculates the sine.
SinH()	Calculates the hyperbolic sine.
Sqrt()	Calculates the square root of a positive number
Tan()	Calculates the tangent.
TanH()	Calculates the hyperbolic tangent.

Mathematical operators

%	Modulus operator (binary): calculates the remainder of a division.
*	Multiplication operator (binary): multiplys numeric values.
**	Exponentiation (binary): raises a number to the power of an exponent.
+	Plus operator: add values, concatenate values and unary positive.
++	Increment operator (unary): prefix / postfix increment.
-	Minus operator: add values, concatenate values and unary negative.
--	Decrement operator (unary): Prefix / postfix decrement
/	Division operator (binary): divides numeric values.
= (compound assignment)	Compound assignment (binary): inline operation with assignment.

Memo functions

HardCR()	Replaces soft carriage returns with hard CRs in a character string.
HB_ReadLine()	Scans a formatted character string or memo field for text lines.
MemoEdit()	Displays and/or edits character strings and memo fields in text mode.
MemoLine()	Extracts a line of text from a formatted character string or memo field.
MemoRead()	Reads an entire file from disk into memory.
MemoTran()	Replaces "carriage return/line feed" pairs in a character string.
MemoWrit()	Writes a character string or a memo field to a file.
MLCount()	Counts the number of lines in a character string or memo field
MLCToPos()	Determines the position of a single character in a formatted text string or memo field.
MlPos()	Determines the starting position of a line in a formatted character string or memo field.

MPosToLC()	Calculates row and column position of a character in a formatted string or memo field.

Memory commands

CLEAR ALL	Closes files in all work areas and releases all dynamic memory variables.
CLEAR MEMORY	Releases all dynamic memory variables.
DEFAULT TO	Assigns a default value to a NIL argument.
QUIT	Terminates program execution unconditionally.
RELEASE	Deletes or resets dynamic memory variables.
RESTORE	Loads dynamic memory variables from disk into memory.
SAVE	Saves dynamic memory variables to a memory file.
STORE	Assigns a value to one ore more variables.

Miscellaneous functions

Blank()	Returns empty values for the data types A, C, D, L, M and N.
Complement()	Creates the complement for values of data type C, D, L, M, N
Default()	Assigns a default value to a variable.
KbdStat()	Determines the state of special keys like Ctrl or Shift keys.
KeySec()	Starts a timer to write a key code into the keyboard buffer.
KeyTime()	Writes a key code into the keyboard buffer at a specified time.
LtoC()	Converts a logical value to a character.
Nul()	Returns a null string ("").
StrPeek()	Determines the ASCII code of a specified character in a string.
StrPoke()	Changes the ASCII code of a specified character in a string.
XtoC()	Converts values of data type C, D, L, M, N to a string.

Mouse functions

Inkey()	Retrieves a character from the keyboard buffer or a mouse event.
LastKey()	Returns the last Inkey() code retrieved.
MCol()	Determines the screen column position of the mouse cursor.
MDblClk()	Determines the double-click interval for the mouse.
MHide()	Hides the mouse cursor.
MLeftDown()	Determines the status of the left mouse button.
MPresent()	Determines if a mouse is available.
MRestState()	Restores a previously saved state of the mouse.
MRightDown()	Determines the status of the left mouse button.
MRow()	Determines the screen row position of the mouse cursor.
MSaveState()	Saves the current state of the mouse.
MSetBounds()	Sets restricting boundaries for the mouse cursor.
MSetCursor()	Determines the visibility of the mouse cursor.
MSetPos()	Moves the mouse cursor to a new position on screen.
MShow()	Displays the mouse cursor.
NextKey()	Returns the next pending key or mouse event.
NumButtons()	Returns the number of mouse buttons.
SetMouse()	Determines the visibility of the mouse cursor.
TBMouse()	Moves the browse cursor to the mouse pointer.
WMSetPos()	Moves the mouse cursor to a new position in a window.
WSetMouse()	Determines the visibility and/or position of the mouse cursor.

Multi-threading functions

GetCurrentThread()	Retrieves the handle of the current thread.

GetSystemThreadID()	Retrieves the numeric system Thread ID of a thread.
GetThreadID()	Retrieves the numeric application Thread ID of a thread.
HbConsoleLock()	Locks the console for the current thread
HbConsoleUnlock()	Releases the console lock.
HB_MultiThread()	Checks if an application is created for multi-threading.
HB_MutexCreate()	Creates a Mutex.
HB_MutexLock()	Obtains a permanent lock on a Mutex.
HB_MutexTimeoutLock()	Tries to obtain a lock on a Mutex with timeout.
HB_MutexTryLock()	Tries to obtain a permanent lock on a Mutex.
HB_MutexUnlock()	Unlocks a Mutex.
IsSameThread()	Compares two thread handles.
IsValidThread()	Checks if an expression is the thread handle of a running thread.
JoinThread()	Suspends the current thread until a second thread has terminated.
KillAllThreads()	Kills all running threads except for the main thread.
KillThread()	Kills a running thread.
Notify()	Resumes a single thread blocked by a particular Mutex.
NotifyAll()	Resumes all threads blocked by a particular Mutex.
SecondsSleep()	Suspends thread execution for a number of seconds.
StartThread()	Starts a new thread.
StopThread()	Stops a thread from outside.
Subscribe()	Subscribes for notifications on a Mutex.
SubscribeNow()	Subscribes for notifications on a Mutex and discards pending notifications.
ThreadSleep()	Puts a thread to sleep.
WaitForThreads()	Suspends the current thread until all other threads have terminated.

Mutex functions

HB_MutexCreate()	Creates a Mutex.
HB_MutexLock()	Obtains a permanent lock on a Mutex.
HB_MutexTimeoutLock()	Tries to obtain a lock on a Mutex with timeout.
HB_MutexTryLock()	Tries to obtain a permanent lock on a Mutex.
HB_MutexUnlock()	Unlocks a Mutex.
Notify()	Resumes a single thread blocked by a particular Mutex.
NotifyAll()	Resumes all threads blocked by a particular Mutex.
Subscribe()	Subscribes for notifications on a Mutex.
SubscribeNow()	Subscribes for notifications on a Mutex and discards pending notifications.

Network functions

DbRLock()	Locks a record for write access.
DbRLockList()	Returns a list of locked records.
DbRUnlock()	Unlocks a record based on its indentifier.
DbUnlock()	Releases file and all record locks in a work area.
DbUnlockAll()	Unlocks all records and releases all file locks in all work areas.
FLock()	Applies a file lock to an open, shared database.
IsLocked()	Checks if a record is locked.
NetAppend()	Appends a new record to a database open in shared mode in a work area.
NetCancel()	Releases an existing network connection.
NetCommitAll()	Writes database and index buffers of all used work areas to disk.
NetDbUse()	Opens a database file for shared access in a work area.
NetDelete()	Marks records for deletion.
NetDisk()	Tests if a drive is a network drive.
NetErr()	Determines if a database command has failed in network operation.
NetError()	Determines if a Net*() function has failed.
NetFileLock()	Applies a file lock to an open, shared database.
NetFunc()	Evaluates a code block until a timeout period expires.
NetLock()	Applies locks to a database file or record with timeout.

NetName()	Retrieves the current user name or the computer name.
NetOpenFiles()	Opens databases and associated index files.
NetPrinter()	Tests if the current printer is a network printer.
NetRecall()	Recalls a record previousy marked for deletion.
NetRecLock()	Locks the current record for write access.
NetRedir()	Establishes a connection to a server.
NetRmtName()	Returns the remote name of a network device.
Network()	Tests for the existence of a network.
NNetwork()	Tests for the existence of a Novell network.
Os_NetRegOk()	Checks for correct network registry settings on Windows platforms.
OS_NetVRedirOk()	Checks for the correct VREDIR.VXD file.
RLock()	Locks the current record for concurrent write access in a work area.
SetNetDelay()	Queries or changes the timeout period for Net*() functions.
SetNetMsgColor()	Queries or changes the color for displaying failure messages of Net*() functions.

New topics in this help file

@...GET CHECKBOX	Creates a Get object as check box and displays it to the screen.
@...GET LISTBOX	Creates a Get object as list box and displays it to the screen.
@...GET PUSHBUTTON	Creates a Get object as push button and displays it to the screen.
@...GET RADIOGROUP	Creates a Get object as radio button group and displays it to the screen.
@...GET TBROWSE	Creates a Get object as browser and displays it to the screen
ACos()	Calculates the arc cosine.
AddASCII()	Adds a numeric value to the ASCII code of a specified character in a string.
AddMonth()	Adds or subtracts a number of months to/from a Date value.
AfterAtNum()	Extracts the remainder of a string after the last occurrence of a search string.
AnsiToHtml()	Inserts HTML character entities into an ANSI text string,
ASCIISum()	Sums the ASCII codes of all characters in a string.
AscPos()	Determines the ASCII code of a specified character in a string.
ASin()	Calculates the arc sine.
AtAdjust()	Justifies a character sequence within a string.
ATan()	Calculates the arc tangent.
ATn2()	Calculates the radians of an angle from sine and cosine.
AtNum()	Searches multiple occurrences of a substring within a string.
AtRepl()	Searches and replaces a substring within a string.
AtSkipStrings()	Locates the position of a substring within a character string.
AtToken()	Returns the position of the n-th token in a string.
BeforAtNum()	Extracts the remainder of a string before the last occurrance of a search string.
BitToC()	Translates bits of an integer to a character string.
Blank()	Returns empty values for the data types A, C, D, L, M and N.
BoM()	Returns the date of the first day of a month.
BoQ()	Returns the date of the first day of a quarter.
BoY()	Returns the date of the first day of a year.
CallDll()	Executes a function located in a dynamically loaded external library.
Ceiling()	Rounds a decimal number to the next higher integer.
Celsius()	Converts degrees Fahrenheit to Celsius.
Center()	Returns a string for centered display.
CFTSAdd()	Adds a text string entry to a Full Text Search index file.
CFTSClose()	Closes a Full Text Search index file.
CFTSCrea()	Creates a new Full Text Search index file.
CFTSDelete()	Marks an index entry as deleted in a Full Text Search index file.
CFTSIfDel()	Checks if a Full Text Search index entry is marked as deleted.
CFTSNext()	Searches a Full Text Search index file for a matching index entry.
CFTSOpen()	Opens a Full Text Search index file.

CFTSRecn()	Returns the number of index entries in a Full Text Search index file.
CFTSReplac()	Changes a Full Text Search index entry.
CFTSSet()	Defines a search string for subsequent CFTSNext() calls.
CFTSUndel()	Removes the deletion mark from an index entry in a Full Text Search index file.
CFTSVeri()	Verifies a CFTSNext() match against the index key.
CFTSVers()	Returns version information for HiPer-SEEK functions.
CharAdd()	Creates a string from the sum of ASCII codes of two strings.
CharAND()	Binary ANDs the ASCII codes of characters in two strings.
CharEven()	Extracts characters at even positions from a string.
CharHist()	Creates a histogram of characters in a character string
CharList()	Removes duplicate characters from a string.
CharMirr()	Reverses the order of characters in a string.
CharMix()	Merges the characters of two strings.
CharNoList()	Returns a string containing all characters not included in a string.
CharNOT()	Binary NOTs the ASCII codes of characters in two strings.
CharOdd()	Extracts characters at odd positions from a string.
CharOne()	Removes duplicate adjacent characters from a string.
CharOnly()	Removes all characters but the specified ones from a string
CharOR()	Binary ORs the ASCII codes of characters in two strings.
CharPack()	Compresses a string.
CharRela()	Tests if two substrings in two strings have the same position.
CharRelRep()	Replaces characters if two substrings in two strings have the same position.
CharRem()	Deletes specified characters from a string.
CharRepl()	Searches a list of characters and replaces them with a corresponding list.
CharRLL()	Rotates bits in a character string to the left.
CharRLR()	Rotates bits in a character string to the right.
CharSHL()	Shifts bits in a character string to the left.
CharSHR()	Shifts bits in a character string to the right.
CharSList()	Removes duplicate characters from a string and sorts the result.
CharSort()	Sorts character (sequences) within a string.
CharSpread()	Formats a character string for block paragraphs.
CharSub()	Creates a string by subtracting ASCII codes of two strings.
CharSwap()	Exchanges adjacent characters in a string.
CharUnpack()	Uncompresses a CharPack() compressed string.
CharWin()	Replaces characters in a specified screen area.
CharXOR()	Binary XORs the ASCII codes of characters in two strings.
CheckSum()	Calculates the checksum for a character string.
ClearBit()	Sets one or more bits of a numeric integer value to 0.
ClearEol()	Clears a row on the screen beginning at a specified position.
ClearSlow()	Clears a screen area incrementally with a delayed imploding effect.
ClearWin()	Clears all or parts of the screen.
ClEol()	Clears characters and colors in a row on the screen.
CLOSE LOG	Closes all open log channels.
ClWin()	Clears characters and colors on the screen.
ColorRepl()	Replaces color attributes on the screen.
ColorToN()	Converts a color value to a numeric color attribute.
ColorWin()	Replaces a color attribute in a screen region.
Complement()	Creates the complement for values of data type C, D, L, M, N
Cos()	Calculates the cosine for an angle.
CosH()	Calculates the hyperbolic cosine for an angle.
Cot()	Calculates the cotangent.
CountGets()	Returns the number of Get fields in the current Getlist array.
CountLeft()	Counts a specified character from the left side of a string.
CountRight()	Counts a specified character from the right side of a string.
CreateObject()	Instantiates a new OLE Automation object.
Crypt()	Encrypts or decrypts a character string.
CSetAtMuPa()	Queries or changes the multi-pass mode for At***() functions.

CSetCent()	Queries or changes the SET CENTURY setting.
CSetCurs()	Queries or changes the SET CURSOR setting.
CSetKey()	Retrieves the code block associated with a key.
CSetRef()	Queries or changes the pass-by-reference mode for several string functions.
CSetSafety()	Retrieves and/or changes the safety switch used in CA-Tools file operations.
CStrToVal()	Converts a character string to a value of specific data type.
CtoBit()	Converts a character string to an integer based on a bit pattern.
CtoDoW()	Returns the number of a week day from its name.
CtoF()	Converts an 8 byte string to a floating point number.
CtoMonth()	Returns the number of a month from its name.
CtoN()	Converts a string of digits to an integer of the specified base.
CtoT()	Converts a character string into a DateTime value
CurDirX()	Returns the current directory of a drive including directory separators.
CurrentGet()	Returns the position of the current Get field in the Getlist array.
DateTime()	Returns the current date and time from the operating system.
DaysInMonth()	Returns the number of days in a month.
DaysToMonth()	Returns the number of days from first January to the beginning of a month.
DbfSize()	Returns the size of a database file in memory that is opened in a workarea.
DbJoin()	Merges records of two work areas into a new database.
Dblist()	Displays records of a work area to the console, printer or file.
DbSort()	Creates a new, physically sorted database.
DbTotal()	Creates a new database summarizing numeric fields by an expression.
DbUpdate()	Updates records in the current work area from a second work area.
Default()	Assigns a default value to a variable.
DeleteFile()	Deletes a file with error handling.
DirMake()	Creates a directory.
DirName()	Returns the current directory.
DisableWaitLocks()	Toggles the exclusive file opening mode.
DiskFormat()	Formats a floppy disk.
DiskFree()	Returns the free storage space of a disk drive in bytes.
DiskReady()	Test if a disk drive is ready.
DiskReadyW()	Tests if a drive can be written to.
DiskTotal()	Returns the totl storage space of a disk drive in bytes.
DiskUsed()	Returns the used storage space of a disk drive in bytes.
DMY()	Formats a date as "dd. Month yyyy"
DosParam()	Returns the command line parameters passed to an xHarbour application.
DoY()	Returns the day number of a Date value in a year.
DriveType()	Determines the type of a drive.
DtoR()	Converts an angle from degrees to radians.
Enhanced()	Selects the enhanced color of SetColor().
EoM()	Returns the date of the last day in a month.
EoQ()	Returns the date of the last day in a quarter.
EoY()	Returns the date for the last day of a year.
ExeName()	Returns the EXE file name of an xHarbour application.
Expand()	Inserts characters between all characters in a string.
Exponent()	Calculates the exponent of a floating point number.
Fact()	Calculates the factorial of a number.
Fahrenheit()	Converts degrees Celsius to Fahrenheit.
FieldDeci()	Returns the decimal places of a database field.
FieldNum()	Returns the ordinal position of a field in a database.
FieldSize()	Returns the length of a database field.
FileAppend()	Concatenates two files.
FileAttr()	Returns the attributes of a file.
FileCClose()	Closes the file opened in backup mode with FileCopy().
FileCCont()	Continues file copying in backup mode of FileCopy().
FileCOpen()	Tests if the file opened in backup mode with FileCopy() is still open.

FileCopy()	Copies files normally or in backup mode.
FileDate()	Returns the date of a file.
FileDelete()	Deletes one or more files specified by a file mask and file attributes.
FileMove()	Moves a file to another directory.
FileReader()	Returns a new TStreamFileReader object.
FileScreen()	Reads the contents of a screen from a file.
FileSeek()	Seeks files specified by a file mask and file attributes.
FileSize()	Returns the size of a file.
FileStr()	Reads a string from a file beginning at a specified offset.
FileTime()	Returns the change time of a file.
FileValid()	Tests if a string contains a valid file name.
FileWriter()	Returns a new TStreamFileWriter object.
Floor()	Rounds a decimal number to the next lower integer.
FloppyType()	Determines the type of a floppy drive.
FtoC()	Converts a floating point number to an 8 byte binary string.
Fv()	Calculates the future value of constant, periodic investments.
GetActive()	Returns or assigns the current Get object during READ.
GetActiveObject()	Retturns an instantiated OLE Automation object.
GetApplyKey()	Sends an Inkey() code to the currently active Get object.
GetClearA()	Returns the default color attribute for clearing the screen.
GetClearB()	Returns the default character for clearing the screen.
GetClrBack()	Returns the background color of a color value.
GetClrFore()	Returns the foreground color of a color value.
GetClrPair()	Extracts a color value from a color string.
GetDoSetKey()	Evaluates a SetKey() code block during READ.
GetE()	Retrieves an operating system environment variable.
GetFldCol()	Returns the screen column position of a Get field.
GetFldRow()	Returns the screen row position of a Get field.
GetFldVar()	Returns the name of a Get variable.
GetPairLen()	Returns the length of a color value within a color string.
GetPairPos()	Returns the absolute position of a color value in a color string.
GetPostValidate()	Validates data after editing.
GetPrec()	Retrieves computing precision for trigonometric functions.
GetPreValidate()	Validates data before editing.
GetSecret()	Allows for hidden Get input for character strings.
GetVolInfo()	Retrieves the volume label of a disk.
GuiApplyKey()	Sends an Inkey() code to the associated control of the currently active Get object.
GuiGetPostValidate()	Validates data after editing.
GuiGetPrevalidate()	Validates data before editing.
GuiReader()	Processes user input.
HbCheckBox()	Creates a new HbCheckBox object.
HbConsoleLock()	Locks the console for the current thread
HbConsoleUnlock()	Releases the console lock.
HbListBox()	Creates a new HbListBox object
HbPushButton()	Creates a new HbPushButton object.
HbRadioButton()	Creates a new HRadioButton object.
HbRadioGroup()	Creates a new HbRadioGroup object.
HbScrollBar()	Creates a new HbScrollBar object.
HB_ArrayBlock()	Creates a set/get code block for an array.
HB_Base64Decode()	Decodes a base 64 encoded character string.
HB_Base64DecodeFile()	Decodes a base 64 encoded file.
HB_Base64Encode()	Encodes a character string base 64.
HB_Base64EncodeFile()	Encodes a file base 64.
HB_BldLogMsg()	Converts a list of parameters into a text string.
HB_Clocks2Secs()	Calculates seconds from CPU ticks.
HB_CloseProcess()	Closes a child process
HB_ClrArea()	Replaces a color attribute in a screen region.
HB_ColorToN()	Converts a color value to a numeric color attribute.

HB_Compress()	Compresses a character string (ZIP).
HB_CompressBufLen()	Calculates the buffer size required for compression.
HB_CompressError()	Returns the error code of the last (un)compression.
HB_CompressErrorDesc()	Returns an error description from a numeric error code.
HB_CreateLen8()	Converts a numeric value to an eight byte character string in network byte order.
HB_Decode()	Provides a functional equivalent for the DO CASE statement.
HB_DecodeOrEmpty()	Provides a functional equivalent for the DO CASE statement.
HB_DeserialBegin()	Initiates deserialization of a group of variables of simple or complex data types.
HB_DeserializeSimple()	Deserializes values of simple data types.
HB_DeserialNext()	Deserialization the next variable of simple or complex data types.
HB_DiskSpace()	Returns the free storage space for a disk drive by drive letter.
HB_DumpVar()	Converts a value to a character string holding human readable text.
HB_FCommit()	Forces a disk write.
HB_FCreate()	Creates and/or opens a binary file.
HB_FEof()	Tests if the end-of-file is reached in the currently selected text file.
HB_FGoBottom()	Moves the file pointer to the last line in a text file.
HB_FGoto()	Moves the record pointer to a specific line in the currently selected text file.
HB_FGoTop()	Moves the record pointer to the begin-of-file.
HB_FInfo()	Retrieves status information about the currently selected text file.
HB_FLastRec()	Returns the number of lines in the currently selected text file.
HB_FReadAndSkip()	Reads the current line and moves the record pointer.
HB_FreadLN()	Reads the current line and without moving the record pointer.
HB_FRecno()	Returns the current line number of the currently selected text file.
HB_FSelect()	Queries or changes the currently selected text file area.
HB_FSkip()	Moves the record pointer in the currently selected text file.
HB_FUse()	Opens or closes a text file in a text file area.
HB_F_Eof()	Tests for End-of-file with binary files.
HB_GetLen8()	Retrieves the length of data in a serialized binary string.
HB_IdleWaitNoCPU()	Toggles the mode for CPU usage in Idle wait states.
HB_IsDateTime()	Tests if the value returned by an expression is a DateTime value.
HB_LangErrMsg()	Returns language specific error messages.
HB_LangMessage()	Returns language specific messages or text strings.
HB_LangName()	Returns the currently selected language.
HB_LogDateStamp()	Returns a character string holding a date stamp.
HB_OpenProcess()	Opens a child process.
HB_OsDriveSeparator()	Returns the operating specific drive separator character.
HB_OsError()	Returns the operating specific error code of the last low-level file operation.
HB_OsPathDelimiters()	Returns the operating specific characters for paths.
HB_OsPathListSeparator()	Returns the operating specific separator character for a path list.
HB_OsPathSeparator()	Returns the operating specific separator character for a path.
HB_PCodeVer()	Retrieves the PCode version of the current xHarbour build.
HB_Pointer2String()	Reads bytes from a pointer into a character string.
HB_ProcessValue()	Retrieves the return value of a child process.
HB_ReadLine()	Scans a formatted character string or memo field for text lines.
HB_RegExAtX()	Parses a string and fills an array with parsing information.
HB_RegExReplace()	Searches and replaces characters within a character string using a regular expression.
HB_SerializeSimple()	Serializes values of simple data types.
HB_SerialNext()	Returns the position of the next chunk of binary data to retrieve.
HB_SetKeyArray()	Associates a code block with multiple keys.
HB_SetKeyCheck()	Evaluates a code block associated with a key.
HB_SetKeyGet()	Retrieves code blocks associated with a key.
HB_SetKeySave()	Queries or changes all SetKey() code blocks.
HB_String2Pointer()	Obtains the pointer for a character string.
HB_SysLogClose()	Closes the log file of the operating system.

HB_SysLogMessage()	Adds a new entry to the log file of the operating system.
HB_SysLogOpen()	Opens the log file of the operating system
HB_Translate()	Converts a character string from one code page to another one.
HB_Uncompress()	Uncompresses a compressed character string (ZIP).
HB_UUDecode()	Decodes a UUEncoded character string.
HB_UUDecodeFile()	Decodes a UUEncoded file.
HB_UUEncode()	UUEncodes a character string.
HB_UUEncodeFile()	UUEncodes a file.
HB_ValToStr()	Converts values of simple data types to character string.
Hour()	Extracts the hour from a DateTime value
HS_Add()	Adds a text string entry to a HiPer-SEEK index file.
HS_Close()	Closes a HiPer-SEEK index file.
HS_Create()	Creates a new HiPer-SEEK index file.
HS_Delete()	Marks an index entry as deleted in a HiPer-SEEK index file.
HS_Filter()	Uses a HiPer-SEEK index file as filter for a work area
HS_IfDel()	Checks if a HiPer-SEEK index entry is marked as deleted
HS_Index()	Creates a new HiPer-SEEK index file and fills it with index entries.
HS_KeyCount()	Returns the number of index entries in a HiPer-SEEK index file.
HS_Next()	Searches a HiPer-SEEK index file for a matching index entry.
HS_Open()	Opens a HiPer-SEEK index file.
HS_Replace()	Changes a HiPer-SEEK index entry.
HS_Set()	Defines a search string for subsequent HS_Next() calls.
HS_Undelete()	Removes the deletion mark from an index entry in a HiPer-SEEK index file.
HS_Verify()	Verifies a HS_Next() match against the index key.
HS_Version()	Returns version information for HiPer-SEEK functions.
HtmlToAnsi()	Converts an HTML formatted text string to the ANSI character set.
HtmlToOem()	Converts an HTML formatted text string to the OEM character set
INetAccept()	Waits for an incoming connection on a server side socket.
INetAddress()	Determines the internet address of a remote station.
INetCleanup()	Releases memory resources for sockets.
INetClearError()	Resets the last error code of a socket.
INetClearPeriodCallback()	Voids a callback function for a socket.
INetClearTimeout()	Voids a timout value for a socket.
INetClose()	Closes a connection on a socket.
INetConnect()	Establishes a sockets connection to a server.
INetConnectIP()	Establishes a sockets connection to a server using the IP address.
INetCount()	Returns the number of bytes transferred in the last sockets operation
INetCreate()	Creates a raw, unconnected socket.
INetCRLF()	Returns new line characters used in internet communications.
INetDataReady()	Tests if incoming data is available to be read.
INetDGram()	Creates an unbound datagram oriented socket.
INetDGramBind()	Creates a bound datagram oriented socket.
INetDGramRecv()	Reads data from a datagram socket.
INetDGramSend()	Sends data to a datagram socket.
INetErrorCode()	Returns the last sockets error code.
INetErrorDesc()	Returns a descriptive error message
INetGetAlias()	Retrieves alias names from a server.
INetGetHosts()	Queries IP addresses associated with a name.
INetGetPeriodCallback()	Queries a callback associated with a socket.
INetGetTimeout()	Queries a timout value for a socket.
INetInit()	Initializes the sockets subsystem.
INetPort()	Determines the port number a socket is connected to.
INetRecv()	Reads data from a socket.
INetRecvAll()	Reads all data from a socket.
INetRecvEndblock()	Reads one block of data until an end-of-block marker is detected.
INetRecvLine()	Reads one line of data until CRLF is detected.
INetSend()	Sends data to a socket.
INetSendAll()	Sends all data to a socket.

INetServer()	Creates a server side socket.
INetSetPeriodCallback()	Associates callback information with a socket.
INetSetTimeout()	Sets a timout value in milliseconds for a socket.
Infinity()	Returns the largest number.
INIT LOG	Initializes the Log system and opens requested log channels.
InvertAttr()	Exchanges the foreground and background color.
InvertWin()	Exchanges the foreground and background color on the screen.
IsAffirm()	Converts "Yes" in a national language to a logical value.
IsBit()	Checks whether a bit at a specified position is set.
IsDefColor()	Checks if the default color is set.
IsDir()	Checks if a character string contains the name of an existing directory.
IsLeap()	Checks if a Date value belongs to a leap year.
IsLocked()	Checks if a record is locked.
IsNegative()	Converts "No" in a national language to a logical value.
JustLeft()	Left justifies characters in a character string.
JustRight()	Right justifies characters in a character string.
KbdStat()	Determines the state of special keys like Ctrl or Shift keys.
KeySec()	Starts a timer to write a key code into the keyboard buffer.
KeyTime()	Writes a key code into the keyboard buffer at a specified time.
KSetCaps()	Queries or changes the status of the Caps lock key
KSetIns()	Queries or changes the status of the Insert/Overwrite key
KSetNum()	Queries or changes the status of the Num lock key
KSetScroll()	Queries or changes the status of the Scroll lock key
LastDayoM()	Returns the number of days in a month.
LIST	Lists records of a work area to the console, file or printer.
LOG	Sends a message to open log channels.
Log10()	Calculates the base 10 logarithm.
LtoC()	Converts a logical value to a character.
LtoN()	Converts a logical value to a numeric value.
Mantissa()	Calculates the mantissa of a floating point number.
MaxLine()	Returns the longest line in an ASCII formatted character string.
MDY()	Formats a date as "Month dd, yy".
MilliSec()	Defines a time delay in milliseconds.
Minute()	Extracts the minute from a DateTime value
NationMsg()	Returns application specific messages.
NetAppend()	Appends a new record to a database open in shared mode in a work area.
NetCancel()	Releases an existing network connection.
NetCommitAll()	Writes database and index buffers of all used work areas to disk.
NetDbUse()	Opens a database file for shared access in a work area.
NetDelete()	Marks records for deletion.
NetDisk()	Tests if a drive is a network drive.
NetError()	Determines if a Net*() function has failed.
NetFileLock()	Applies a file lock to an open, shared database.
NetFunc()	Evaluates a code block until a timeout period expires.
NetLock()	Applies locks to a database file or record with timeout.
NetOpenFiles()	Opens databases and associated index files.
NetPrinter()	Tests if the current printer is a network printer.
NetRecall()	Recalls a record previousy marked for deletion.
NetRecLock()	Locks the current record for write access.
NetRedir()	Establishes a connection to a server.
NetRmtName()	Returns the remote name of a network device.
Network()	Tests for the existence of a network.
NNetwork()	Tests for the existence of a Novell network.
NtoC()	Converts an integer to a string of digits for a specified number base.
NtoCDoW()	Converts a numeric week day to its name.
NtoCMonth()	Converts a numeric month to its name.
NtoColor()	Converts a numeric color attribute to a color string.
Nul()	Returns a null string ("").
NumAND()	Performs bitwise AND operations for a list of integer values.

NumAndX()	Performs bitwise AND operations for a list of integer values.
NumAt()	Counts multiple occurrences of a substring within a string.
NumCount()	Installs and/or increments an internal counter value.
NumDiskL()	Returns the number of logical drives.
NumHigh()	Retrieves the high byte of a 16-bit integer.
NumLine()	Returns the number of lines in an ASCII formatted character string.
NumLow()	Retrieves the low byte of a 16 bit integer.
NumMirr()	Mirrors 8 or 16 bits of a 16-bit integer.
NumMirrX()	Mirrors bits of a numeric integer value.
NumNOT()	Performs a bitwise NOT operation with a numeric integer value.
NumNotX()	Performs a bitwise NOT operation with a numeric integer value.
NumOR()	Performs a bitwise OR for a list of integer values.
NumOrX()	Performs a bitwise OR for a list of integer values.
NumRoL()	Rotates bits of a numeric 16-bit integer value to the left.
NumRolX()	Rotates bits of a numeric integer value to the left.
NumToken()	Returns the number of tokens in a string.
NumXOR()	Performs a bitwise XOR operation with two numeric 32-bit integer values.
NumXorX()	Performs a bitwise XOR operation with two numeric integer values.
Occurs()	Counts the occurrance of a substring in a character string.
OemToHtml()	Inserts HTML character entities into an OEM text string.
Ole2TxtError()	Returns the last OLE error code as character string.
OleError()	Returns the last OLE error code.
Os_NetRegOk()	Checks for correct network registry settings on Windows platforms.
OS_NetVRedirOk()	Checks for the correct VREDIR.VXD file.
PadLeft()	Pads a character string on the left.
PadRight()	Pads a character string on the right.
Payment()	Calculates a periodical payment for loans.
Periods()	Calculates the duration for paying back a loan at fixed interest rate and payments.
Pi()	Returns Pi with highest accuracy.
PosAlpha()	Returns the position of the first alphabetic characters in a character string.
PosChar()	Replaces a single character at a specified position in a string.
PosDel()	Deletes character(s) at a specified position in a string.
PosDiff()	Searches the first position where two character strings differ.
PosEqual()	Searches the first position where two character strings are equal.
PosIns()	Inserts character(s) at a specified position in a string.
PosLower()	Returns the position of the first lower case letter in a character string.
PosRange()	Retrieves the position of the first character out of a range found in a string.
PosRepl()	Replaces characters in a string beginning at a specified position.
PosUpper()	Returns the position of the first upper case letter in a character string.
PrgExpToVal()	Converts a character string obtained from ValToPrgExp() back to the original data type.
PrintReady()	Tests if a printer connected to a specified port is ready.
PrintSend()	Sends a string or a single character to the printer.
PrintStat()	Returns the status of a printer.
Pv()	Calculates the present value of future capital, based on periodic investments.
Quarter()	Returns the quarter a date belongs to.
RangeRem()	Deletes character(s) within a specified range of characters.
RangeRepl()	Repleaces character(s) within a specified range of characters.
Rate()	Calculates the interest rate for a loan.
RddRegister()	Registers a user defined Replaceable Database Driver (RDD).
ReadExit()	Enables or disables up/down arrow keys as exit keys for READ
ReadInsert()	Queries or changes the insert/overstrike mode during READ.
ReadKey()	Returns the key code of the key that has terminated READ.
ReadKill()	Queries or changes the READ termination flag.
ReadUpdated()	Queries or changes the updated flag of the READ command.
ReadVar()	Returns name of the current GET or MENU TO variable.
RemAll()	Deletes a specified character from both sides of a string.

RemLeft()	Deletes a specified character from the left side of a string.
RemRight()	Deletes a specified character from the right side of a string.
RenameFile()	Renames a file and handles errors.
ReplAll()	Replaces a specified character on both sides of a string.
ReplLeft()	Replaces a specified character on the left side of a string.
ReplRight()	Replaces a specified character on the right side of a string.
RestCursor()	Restores shape and position of the screen cursor.
RestGets()	Restores a previously saved !IGetlist!EI array.
RestSetKey()	Restores SetKey() settings and associated code blocks.
RestToken()	Restores the global environment of the incremental tokenizer.
RtoD()	Converts angles from radians to degrees.
SaveCursor()	Saves the current cursor shape and position.
SaveGets()	Saves the current !IGetlist!EI array for nested READs.
SaveSetKey()	Saves SetKey() settings and associated code blocks.
SaveToken()	Saves the global environment of the incremental tokenizer.
SayDown()	Outputs a string vertically to the bottom of the screen.
SayMoveIn()	Outputs a string on the screen using a "move in" effect.
SayScreen()	Displays a string on screen keeping existing color attributes.
SaySpread()	Outputs a string on the screen using a "spread" effect.
ScreenAttr()	Returns the numeric color attribute for a specified coordinate on the screen.
ScreenFile()	Writes the contents of the current screen to a file.
ScreenMark()	Searches strings on the screen and changes their color.
ScreenMix()	Mixes a character string with color attributes.
ScreenStr()	Returns the screen contents beginning at a specified position.
Scrollfixed()	Scrolls a screen region horizontally and/or vertically.
SecondsCpu()	Returns the CPU time used by the current process.
SecToTime()	Converts numeric seconds into a time formatted character string.
SET LOG STYLE	Selects a style for logging data to log channels.
SetAtLike()	Sets the search mode for At***() functions
SetBit()	Sets one or more bits of a numeric integer value to 1.
SetClearA()	Sets the default color attribute for clearing the screen.
SetClearB()	Sets the default character attribute for clearing the screen.
SetClrPair()	Replaces a color value in a color string.
SetDate()	Changes the system date from a Date value.
SetFAttr()	Sets file attributes.
SetFCreate()	Sets the default file attribute(s) for creating files.
SetFDaTi()	Sets the last change date and time of a file.
SetLastKey()	Changes the return value of function LastKey()
SetNetDelay()	Queries or changes the timeout period for Net*() functions.
SetNetMsgColor()	Queries or changes the color for displaying failure messages of Net*() functions.
SetNewDate()	Changes the system date from Numeric values.
SetNewTime()	Changes the system time from Numeric values.
SetPosBS()	Moves the screen cursor (text mode) one column to the right.
SetPrec()	Specifies the computing precision for trigonometric functions.
SetTime()	Changes the system time from a Time string.
ShowTime()	Displays the system time continuously at a specified screen position.
Sign()	Converts the sign of a number to a numeric value.
Sin()	Calculates the sine.
SinH()	Calculates the hyperbolic sine.
Standard()	Selects the standard color of SetColor().
StoT()	Converts a "YYYYMMDDhhmmss.ccc" formatted string to a DateTime value
Strdel()	Deletes characters from a string based on a mask string.
StrDiff()	Calculates the similarity of two strings.
StrFile()	Writes a string to a file starting at a specified position.
StringToLiteral()	Creates a literal character string from a string.
StrPeek()	Determines the ASCII code of a specified character in a string.

StrPoke()	Changes the ASCII code of a specified character in a string.
StrScreen()	Displays a screen string at the specified position.
StrSwap()	Exchanges characters between two strings.
SX_Decrypt()	Decrypts an encrypted character string.
SX_DtoP()	Converts a Date value into a 3-byte character string.
SX_Encrypt()	Encrypts a character string.
SX_FCompress()	Compresses a file.
SX_FDecompress()	Decompresses a compressed file.
SX_PtoD()	Unpacks a packed 3-byte date value.
TabExpand()	Replaces a tab with a specified number of another character.
TabPack()	Inserts a Tab (Chr(9)) for multiple occurrances of a character.
Tan()	Calculates the tangent.
TanH()	Calculates the hyperbolic tangent.
THtmlCleanup()	Releases memory tables required for HTML classes.
THtmlDocument()	Creates a new THtmlDocument object.
THtmlInit()	Initializes memory tables required for HTML classes.
THtmlIsValid()	Validates a HTML tag name and attribute.
THtmlIterator()	Creates a new THtmlIterator object.
THtmlIteratorRegEx()	Creates a new THtmlIteratorRegEx object.
THtmlIteratorScan()	Creates a new THtmlIteratorScan object.
THtmlNode()	Creates a new THtmlNode object.
TimeToSec()	Calculates the number of seconds since midnight.
TimeValid()	Checks if a character string is a valid time string.
TIpClient()	Abstract class for internet communication.
TIpClientFtp()	Creates a new TIpClientFtp object.
TIpClientHttp()	Creates a new TIpClientHttp object.
TIpClientPop()	Creates a new TIpClientPop object.
TIpClientSmtp()	Creates a new TIpClientSmtp object.
TIpMail()	Creates a new TIpMail object.
Token()	Retrieves the n-th token from a string.
TokenAt()	Returns the start and end position of a token.
TokenEnd()	Tests if tokens can still be found with TokenNext().
TokenExit()	Releases memory resources of the global tokenizer environment.
TokenInit()	Initializes the environment for the incremental tokenizer.
TokenLower()	Changes the first character of tokens to lower case.
TokenNext()	Retrieves the next token from a string.
TokenNum()	Returns the number of tokens in a tokenizer environment.
TokenSep()	Retrieves the separating characters of a token.
TokenUpper()	Changes the first character of tokens to upper case.
TrueName()	Completes a relative path to include the root directory.
TStream()	Abstract class for low-level file data streams.
TStreamFileReader()	Creates a new TStreamFileReader object.
TStreamFileWriter()	Creates a new TStreamFileWriter object.
TtoC()	Converts a DateTime value to a character string in SET DATE and SET TIME format.
TtoS()	Converts a Date value to a character string in YYYYMMDDhhmmss.ccc format.
TUrl()	Creates a new TUrl object.
Unselected()	Selects the unselected color of SetColor().
UntextWin()	Removes all text characters from the screen.
Updated()	Queries the updated flag of the READ command.
UsrRdd_AreaData()	Queries or attaches user-defined data to a work area of a user-defined RDD.
UsrRdd_AreaResult()	Queries the result of the last operation of a user-defined RDD.
UsrRdd_ID()	Queries the ID of a user-defined RDD.
UsrRdd_RddData()	Queries or attaches user-defined data a user-defined RDD.
UsrRdd_SetBof()	Sets the BoF() flag in a work area of a user-defined RDD.
UsrRdd_SetBottom()	Defines the bottom scope value for a work area of a user-defined RDD.
UsrRdd_SetEof()	Sets the Eof() flag in a work area of a user-defined RDD.

UsrRdd_SetFound()	Sets the Found() flag in a work area of a user-defined RDD.
UsrRdd_SetTop()	Defines the top value for a work area of a user-defined RDD.
ValPos()	Returns the numeric value of a digit at a specified position in a string.
VAR	Declares an instance variable of a class.
VolSerial()	Returns the serial number of a disk drive.
Volume()	Sets the volume label of a disk.
WAClose()	Closes all windows.
WaitPeriod()	Defines a wait period and allows for time controlled loops.
WBoard()	Determines the area that can be used for displaying windows.
WBox()	Draws a frame around the current window.
WCenter()	Centers a window on the screen or makes it entirely visible.
WClose()	Closes the current window and selects the next window as current window.
WCol()	Returns the left column position of the selected window.
Week()	Calculates the numeric calendar week from a date.
WfCol()	Returns the left column position of the usable area in a formatted window.
WfLastCol()	Returns the right column position of the usable area in a formatted window.
WfLastRow()	Returns the bottom row position of the usable area in a formatted window.
WFormat()	Defines the usable display area inside the current window.
WfRow()	Returns the top row position of the usable area in a formatted window.
Wild2RegEx()	Converts a character string including wild card characters to a regular expression.
Win32Bmp()	Creates a new Win32Bmp object.
Win32Prn()	Creates a new Win32Prn object.
WInfo()	Returns all coordinates of the current window.
WLastCol()	Returns the right column position of the selected window.
WLastRow()	Returns the bottom row position of the selected window.
WMode()	Determines on which side windows are allowed to be moved off the screen.
WMove()	Changes the upper left coordinate for the current window.
WMSetPos()	Moves the mouse cursor to a new position in a window.
WNum()	Returns the largest window ID of all windows.
WoM()	Calculates the week number in a month.
WOpen()	Creates a new window.
WordOne()	Removes duplicate adjacent words (2-byte sequences) from a string.
WordOnly()	Removes all words (2-byte sequence) but the specified ones from a string
WordRem()	Deletes specified words (2-byte sequence) from a string.
WordRepl()	Replaces words (2-byte sequences) in a string with a specified word.
WordSwap()	Exchanges adjacent words (2-byte sequences) in a string.
WordToChar()	Replaces words (2 byte sequence) with characters (1 byte)
WRow()	Returns the top row position of the selected window.
WSelect()	Returns and/or selects an open window as the current window.
WSetMouse()	Determines the visibility and/or position of the mouse cursor.
WSetMove()	Toggles the setting for moving windows interactively.
WSetShadow()	Defines the shadow color for windows.
WStack()	Returns window IDs of all open windows.
WStep()	Sets the increments for horizontal and vertical window movement.
XtoC()	Converts values of data type C, D, L, M, N to a string.
{^ }	Literal DateTime value.

Numbers and Bits

BitToC()	Translates bits of an integer to a character string.
Celsius()	Converts degrees Fahrenheit to Celsius.
ClearBit()	Sets one or more bits of a numeric integer value to 0.
CtoBit()	Converts a character string to an integer based on a bit pattern.
CtoF()	Converts an 8 byte string to a floating point number.

CtoN()	Converts a string of digits to an integer of the specified base.
Exponent()	Calculates the exponent of a floating point number.
Fahrenheit()	Converts degrees Celsius to Fahrenheit.
FtoC()	Converts a floating point number to an 8 byte binary string.
Infinity()	Returns the largest number.
IsBit()	Checks whether a bit at a specified position is set.
LtoN()	Converts a logical value to a numeric value.
Mantissa()	Calculates the mantissa of a floating point number.
NtoC()	Converts an integer to a string of digits for a specified number base.
NumAND()	Performs bitwise AND operations for a list of integer values.
NumAndX()	Performs bitwise AND operations for a list of integer values.
NumCount()	Installs and/or increments an internal counter value.
NumHigh()	Retrieves the high byte of a 16-bit integer.
NumLow()	Retrieves the low byte of a 16 bit integer.
NumMirr()	Mirrors 8 or 16 bits of a 16-bit integer.
NumMirrX()	Mirrors bits of a numeric integer value.
NumNOT()	Performs a bitwise NOT operation with a numeric integer value.
NumNotX()	Performs a bitwise NOT operation with a numeric integer value.
NumOR()	Performs a bitwise OR for a list of integer values.
NumOrX()	Performs a bitwise OR for a list of integer values.
NumRoL()	Rotates bits of a numeric 16-bit integer value to the left.
NumRolX()	Rotates bits of a numeric integer value to the left.
NumXOR()	Performs a bitwise XOR operation with two numeric 32-bit integer values.
NumXorX()	Performs a bitwise XOR operation with two numeric integer values.
SetBit()	Sets one or more bits of a numeric integer value to 1.

Numeric functions

Abs()	Returns the absolute value of a numeric expression.
Chr()	Converts a numeric ASCII code to a character.
Exp()	Calculates the value of e raised by an exponent.
HB_Random()	Generates a random number between two boundaries.
HB_RandomInt()	Generates a random integer number between two boundaries.
HB_RandomSeed()	Sets the seed for generating random numbers.
HexToNum()	Converts a Hex string to a numeric value.
Int()	Converts a numeric value to an integer.
LenNum()	Returns the number of characters a numeric value needs for display.
Log()	Calculates the natural logarithm of a numeric value.
Max()	Returns the larger value of two Numerics or Dates.
Min()	Returns the smallerr value of two Numerics or Dates.
NumToHex()	Converts a numeric value or a pointer to a Hex string.
Round()	Rounds a numeric value to a specified number of digits
Sqrt()	Calculates the square root of a positive number

Object functions

DbSkipper()	Helper function for browse objects to skip a database
Error()	Creates a new Error object.
ErrorNew()	Creates a new Error object and optionally fills it with data.
FileReader()	Returns a new TStreamFileReader object.
FileWriter()	Returns a new TStreamFileWriter object.
Get()	Creates a new Get object.
GetNew()	Creates a new Get object.
HbCheckBox()	Creates a new HbCheckBox object.
HbListBox()	Creates a new HbListBox object
HBObject()	Abstract base class for user-defined classes.
HBPersistent()	Abstract base class for user-defined persistent object.

HbPushButton()	Creates a new HbPushButton object.
HbRadioButton()	Creates a new HRadioButton object.
HbRadioGroup()	Creates a new HbRadioGroup object.
HbScrollBar()	Creates a new HbScrollBar object.
HB_ArrayId()	Returns a unique identifier for an array or an object variable.
HB_QSelf()	Retrieves the !Iself!EI object during method execution.
HB_QWith()	Retrieves the !Iwith!EI object during WITH OBJECT execution.
HB_ResetWith()	Replaces the !Iwith!EI object during WITH OBJECT execution.
HB_SetWith()	Changes the !Iwith!EI object during WITH OBJECT execution.
HB_ThisArray()	Retrieves an array or object from its pointer.
HB_WithObjectCounter()	Determines the nesting level of WITH OBJECT statements.
MenuItem()	Creates a new MenuItem object.
Popup()	Creates a new Popup menu object.
TBColumn()	Creates a new TBColumn object.
TBrowse()	Creates a new TBrowse object.
TBrowseDB()	Creates a new TBrowse object to be used with a database.
TBrowseNew()	Creates a new TBrowse object.
THtmlDocument()	Creates a new THtmlDocument object.
THtmlIterator()	Creates a new THtmlIterator object.
THtmlIteratorRegEx()	Creates a new THtmlIteratorRegEx object.
THtmlIteratorScan()	Creates a new THtmlIteratorScan object.
THtmlNode()	Creates a new THtmlNode object.
TIpClient()	Abstract class for internet communication.
TIpClientFtp()	Creates a new TIpClientFtp object.
TIpClientHttp()	Creates a new TIpClientHttp object.
TIpClientPop()	Creates a new TIpClientPop object.
TIpClientSmtp()	Creates a new TIpClientSmtp object.
TIpMail()	Creates a new TIpMail object.
TopBarMenu()	Creates a new TopBarMenu object.
TStream()	Abstract class for low-level file data streams.
TStreamFileReader()	Creates a new TStreamFileReader object.
TStreamFileWriter()	Creates a new TStreamFileWriter object.
TUrl()	Creates a new TUrl object.
TXmlDocument()	Creates a new TXmlDocument object.
TXmlIterator()	Creates a new TXmlIterator object.
TXmlIteratorRegEx()	Creates a new TXmlIteratorRegEx object.
TXmlIteratorScan()	Creates a new TXmlIteratorScan object.
TXmlNode()	Creates a new TXmlNode object.
Win32Bmp()	Creates a new Win32Bmp object.
Win32Prn()	Creates a new Win32Prn object.

OLE Automation

CreateObject()	Instantiates a new OLE Automation object.
GetActiveObject()	Retturns an instantiated OLE Automation object.
Ole2TxtError()	Returns the last OLE error code as character string.
OleError()	Returns the last OLE error code.

Operators

$	Substring operator (binary): search substring in string.
%	Modulus operator (binary): calculates the remainder of a division.
& (bitwise AND)	Bitwise AND operator (binary): performs a logical AND operation.
& (macro operator)	Macro operator (unary): compiles a character string at runtime.
( )	Execution or grouping operator.
*	Multiplication operator (binary): multiplys numeric values.
**	Exponentiation (binary): raises a number to the power of an exponent.

<table>
<tr><td>+</td><td>Plus operator: add values, concatenate values and unary positive.</td></tr>
<tr><td>++</td><td>Increment operator (unary): prefix / postfix increment.</td></tr>
<tr><td>-</td><td>Minus operator: add values, concatenate values and unary negative.</td></tr>
<tr><td>--</td><td>Decrement operator (unary): Prefix / postfix decrement</td></tr>
<tr><td>-></td><td>Alias operator (binary): identifies a work area.</td></tr>
<tr><td>.AND.</td><td>Logical AND operator (binary).</td></tr>
<tr><td>.NOT.</td><td>Logical NOT operator (unary).</td></tr>
<tr><td>.OR.</td><td>Logical OR operator (binary).</td></tr>
<tr><td>/</td><td>Division operator (binary): divides numeric values.</td></tr>
<tr><td>:</td><td>Send operator (unary): sends a message to an object.</td></tr>
<tr><td><</td><td>Less than operator (binary): compares the size of two values.</td></tr>
<tr><td><<</td><td>Left-shift operator (binary): shifts bits to the left.</td></tr>
<tr><td><=</td><td>Less than or equal operator (binary): compares the size of two values.</td></tr>
<tr><td><> != #</td><td>Not equal operator (binary): compares two values for inequality.</td></tr>
<tr><td><|| ></td><td>Extended literal code block.</td></tr>
<tr><td>= (assignment)</td><td>Simple assignment operator (binary): assigns a value to a variable.</td></tr>
<tr><td>= (comparison)</td><td>Equal operator (binary): compares two values for equality.</td></tr>
<tr><td>= (compound assignment)</td><td>Compound assignment (binary): inline operation with assignment.</td></tr>
<tr><td>==</td><td>Exact equal operator (binary): compares two values for identity.</td></tr>
<tr><td>></td><td>Greater than operator (binary): compares the size of two values.</td></tr>
<tr><td>>=</td><td>Greater than or equal operator (binary): compares the size of two values.</td></tr>
<tr><td>>></td><td>Right-shift operator (binary): shifts bits to the right.</td></tr>
<tr><td>@</td><td>Pass-by-reference operator (unary): passes variables by reference.</td></tr>
<tr><td>@()</td><td>Function-reference operator (unary): obtains a function pointer.</td></tr>
<tr><td>HAS</td><td>Searches a character string for a matching regular expression.</td></tr>
<tr><td>IN</td><td>Searches a value in another value.</td></tr>
<tr><td>LIKE</td><td>Compares a character string with a regular expression.</td></tr>
<tr><td>[ ] (array)</td><td>Array element operator (unary): retrieves array elements.</td></tr>
<tr><td>[ ] (string)</td><td>Character operator (unary): retrieves a character from a string.</td></tr>
<tr><td>^^</td><td>Bitwise XOR operator (binary): performs a logical XOR operation.</td></tr>
<tr><td>{ }</td><td>Literal array.</td></tr>
<tr><td>{=>}</td><td>Literal hash.</td></tr>
<tr><td>{^ }</td><td>Literal DateTime value.</td></tr>
<tr><td>{|| }</td><td>Literal code block.</td></tr>
<tr><td>| (bitwise OR)</td><td>Bitwise OR operator (binary): performs a logical OR operation.</td></tr>
</table>

Output commands

<table>
<tr><td>@...BOX</td><td>Displays a box on the screen.</td></tr>
<tr><td>@...CLEAR</td><td>Clears the contents of the screen.</td></tr>
<tr><td>@...SAY</td><td>Displays data at a particular row and column position.</td></tr>
<tr><td>@...TO</td><td>Displays a single or double lined frame on the screen.</td></tr>
<tr><td>CLEAR GETS</td><td>Releases all Get objects in the current GetList array.</td></tr>
<tr><td>CLEAR SCREEN</td><td>Clears the screen in text mode.</td></tr>
<tr><td>LIST</td><td>Lists records of a work area to the console, file or printer.</td></tr>
<tr><td>SET COLOR</td><td>Defines default colors for text-mode applications.</td></tr>
<tr><td>SET CURSOR</td><td>Toggles the display of the screen cursor in text-mode applications</td></tr>
<tr><td>SET DECIMALS</td><td>Defines the number of decimal places for displaying numeric values on the screen.</td></tr>
<tr><td>SET DELIMITERS</td><td>Defines delimiting characters for GET entry fields and their visibility.</td></tr>
<tr><td>SET DEVICE</td><td>Selects the ouput device for @...SAY commands.</td></tr>
<tr><td>SET INTENSITY</td><td>Toggles usage of enhanced colors for GET and PROMPT</td></tr>
<tr><td>SET MARGIN</td><td>Defines the left margin for text-mode print output.</td></tr>
<tr><td>SET MESSAGE</td><td>Defines the screen row for @...PROMPT messages.</td></tr>
<tr><td>SET SCOREBOARD</td><td>Toggles the display of messages from READ and MemoEdit().</td></tr>
<tr><td>SET VIDEOMODE</td><td>Changes the current video mode of the application.</td></tr>
<tr><td>SET WRAP</td><td>Toggles wrapping of the highligh bar in text-mode menus.</td></tr>
</table>

Output functions

DevOut()	Outputs a value to the current device.
DevOutPict()	Outputs a PICTURE formatted value to the current device.
DevPos()	Moves the cursor or printhead to a row and column coordinate
DispOut()	Displays a value on the screen
DispOutAt()	Displays a value on the screen at a certain position.
DispOutAtSetPos()	Toggles update of the screen cursor with DispOutAt().
OutErr()	Writes values to the standard error device.
OutStd()	Writes values to the standard output device.
QOut() | QQOut()	Displays values of expressions to the console window.

Pointer functions

HB_Exec()	Executes a function, procedure or method from its pointer.
HB_ExecFromArray()	Executes a function, procedure or method indirectly.
HB_Pointer2String()	Reads bytes from a pointer into a character string.
HB_String2Pointer()	Obtains the pointer for a character string.
Valtype()	Determines the data type of the value returned by an expression.

Preprocessor directives

#command | #translate	User defined command or translation rule for the preprocessor.
#define	Defines a symbolic constant or pseudo-function.
#error	Raises an explicit compiler error along with a message.
#if	Compile a section of code based on a condition.
#ifdef	Compiles a section of code depending on the presence of a #define constant.
#ifndef	Compiles a section of code depending on the absence of a #define constant.
#include	Inserts the contents of a file into the current source code file.
#pragma	Controls compiler switches at compile time.
#stdout	Sends a compiler message to StdOut.
#uncommand | #untranslate	Voids a previously defined #command or #translate directive.
#undef	Voids a #define constant or pseudo-function.
#xcommand | #xtranslate	User defined command or translation rule for the preprocessor.
#xuncommand | #xuntranslate	Voids a previously defined #xcommand or #xtranslate directive.

Printer commands

EJECT	Ejects the current page from the printer.
SET PRINTER	Enables or disables output to the printer or redirects printer output.

Printer functions

GetDefaultPrinter()	Retrieves the name of a computer's default printer.
GetPrinters()	Retrieves information about available printers.
IsPrinter()	Determines if print output can be processed.
PrinterExists()	Checks if a particular printer is installed.
PrinterPortToName()	Retrieves the name of the printer connected to a printer port.
PrintFileRaw()	Prints a file to a Windows printer in RAW mode.
PrintReady()	Tests if a printer connected to a specified port is ready.

PrintSend()	Sends a string or a single character to the printer.
PrintStat()	Returns the status of a printer.
Win32Bmp()	Creates a new Win32Bmp object.
Win32Prn()	Creates a new Win32Prn object.

Process functions

HB_CloseProcess()	Closes a child process
HB_OpenProcess()	Opens a child process.
HB_ProcessValue()	Retrieves the return value of a child process.

Random generator

HB_Random()	Generates a random number between two boundaries.
HB_RandomInt()	Generates a random integer number between two boundaries.
HB_RandomSeed()	Sets the seed for generating random numbers.

Registry functions

GetRegistry()	Retrieves the value of a registry entry
QueryRegistry()	Checks if a particular registry key with specified value exists.
SetRegistry()	Creates a key/value pair in the registry.

Regular expressions

HAS	Searches a character string for a matching regular expression.
HB_AtX()	Locates a substring within a character string based on a regular expression.
HB_IsRegExString()	Checks if a character string is a compiled regular expression.
HB_RegEx()	Searches a string using a regular expression
HB_RegExAll()	Parses a string and fills an array with parsing information.
HB_RegExAtX()	Parses a string and fills an array with parsing information.
HB_RegExComp()	Compiles a regular expression into a binary search pattern.
HB_RegExMatch()	Tests if a string contains a substring using a regular expression
HB_RegExReplace()	Searches and replaces characters within a character string using a regular expression.
HB_RegExSplit()	Parses a string using a regular expression and fills an array.
LIKE	Compares a character string with a regular expression.
Wild2RegEx()	Converts a character string including wild card characters to a regular expression.

Screen functions

CharEven()	Extracts characters at even positions from a string.
CharOdd()	Extracts characters at odd positions from a string.
CharWin()	Replaces characters in a specified screen area.
ClearEol()	Clears a row on the screen beginning at a specified position.
ClearSlow()	Clears a screen area incrementally with a delayed imploding effect.
ClearWin()	Clears all or parts of the screen.
ClEol()	Clears characters and colors in a row on the screen.
ClWin()	Clears characters and colors on the screen.
Col()	Returns the current column position of the screen cursor
ColorRepl()	Replaces color attributes on the screen.
ColorSelect()	Selects a color from the current SetColor() string.

ColorToN()	Converts a color value to a numeric color attribute.
ColorWin()	Replaces a color attribute in a screen region.
DispBegin()	Initiates buffering of console window output.
DispBox()	Displays a box on the screen.
DispCount()	Retrieves the number of pending DispEnd() calls.
DispEnd()	Displays buffered screen output.
DispOut()	Displays a value on the screen
DispOutAt()	Displays a value on the screen at a certain position.
DispOutAtSetPos()	Toggles update of the screen cursor with DispOutAt().
Enhanced()	Selects the enhanced color of SetColor().
FileScreen()	Reads the contents of a screen from a file.
GetClearA()	Returns the default color attribute for clearing the screen.
GetClearB()	Returns the default character for clearing the screen.
GetClrBack()	Returns the background color of a color value.
GetClrFore()	Returns the foreground color of a color value.
GetClrPair()	Extracts a color value from a color string.
GetPairLen()	Returns the length of a color value within a color string.
GetPairPos()	Returns the absolute position of a color value in a color string.
HB_ClrArea()	Replaces a color attribute in a screen region.
HB_ColorIndex()	Extracts a color value from a color string.
HB_ColorToN()	Converts a color value to a numeric color attribute.
HB_Shadow()	Displays a shadow around a rectangular area on the screen.
InvertAttr()	Exchanges the foreground and background color.
InvertWin()	Exchanges the foreground and background color on the screen.
IsDefColor()	Checks if the default color is set.
IsLocked()	Checks if a record is locked.
MaxCol()	Determines the rightmost column position of the screen buffer.
MaxRow()	Determines the bottom row position of the screen buffer.
NtoColor()	Converts a numeric color attribute to a color string.
RestCursor()	Restores shape and position of the screen cursor.
RestScreen()	Displays a SaveScreen() string.
Row()	Returns the current row position of the screen cursor.
SaveCursor()	Saves the current cursor shape and position.
SaveScreen()	Saves a rectangular screen region for later display.
SayDown()	Outputs a string vertically to the bottom of the screen.
SayMoveIn()	Outputs a string on the screen using a "move in" effect.
SayScreen()	Displays a string on screen keeping existing color attributes.
SaySpread()	Outputs a string on the screen using a "spread" effect.
ScreenAttr()	Returns the numeric color attribute for a specified coordinate on the screen.
ScreenFile()	Writes the contents of the current screen to a file.
ScreenMark()	Searches strings on the screen and changes their color.
ScreenMix()	Mixes a character string with color attributes.
ScreenStr()	Returns the screen contents beginning at a specified position.
Scroll()	Scrolls a screen region horizontally and/or vertically.
Scrollfixed()	Scrolls a screen region horizontally and/or vertically.
SetBlink()	Determines how to treat the asterisk in a SetColor() string.
SetClearA()	Sets the default color attribute for clearing the screen.
SetClearB()	Sets the default character attribute for clearing the screen.
SetClrPair()	Replaces a color value in a color string.
SetColor()	Retrieves and/or changes the current color setting for text mode.
SetCursor()	Queries or changes the shape of the cursor on the screen.
SetMode()	Changes the size of a console window.
SetNetMsgColor()	Queries or changes the color for displaying failure messages of Net*() functions.
SetPos()	Changes the position of the screen cursor in text mode.
SetPosBS()	Moves the screen cursor (text mode) one column to the right.
Standard()	Selects the standard color of SetColor().
StrScreen()	Displays a screen string at the specified position.

Unselected()	Selects the unselected color of SetColor().
UntextWin()	Removes all text characters from the screen.

Serialization functions

HB_CreateLen8()	Converts a numeric value to an eight byte character string in network byte order.
HB_DeserialBegin()	Initiates deserialization of a group of variables of simple or complex data types.
HB_DeSerialize()	Converts a binary string back to its original data type.
HB_DeserializeSimple()	Deserializes values of simple data types.
HB_DeserialNext()	Deserialization the next variable of simple or complex data types.
HB_GetLen8()	Retrieves the length of data in a serialized binary string.
HB_Serialize()	Converts an arbitrary value to a binary string.
HB_SerializeSimple()	Serializes values of simple data types.
HB_SerialNext()	Returns the position of the next chunk of binary data to retrieve.

SET commands

SET AUTOPEN	Toggles automatic opening of a structural index file.
SET AUTORDER	Defines the default controlling index for automatically opened index files.
SET AUTOSHARE	Defines network detection for shared file access.
SET BACKGROUND TASKS	Enables or disables the activity of background tasks.
SET BACKGROUNDTICK	Defines the processing interval for background tasks.
SET BELL	Toggles automatic sounding of the bell in the GET system.
SET CENTURY	Sets the date format to include two or four digits.
SET COLOR	Defines default colors for text-mode applications.
SET CONFIRM	Determines how a GET entry field is exited.
SET CURSOR	Toggles the display of the screen cursor in text-mode applications
SET DATE	Specifies the date format for input and display.
SET DBFLOCKSCHEME	Selects the locking scheme for shared database access.
SET DECIMALS	Defines the number of decimal places for displaying numeric values on the screen.
SET DEFAULT	Sets the default drive and directory.
SET DELETED	Specifies visibility of records marked for deletion.
SET DELIMITERS	Defines delimiting characters for GET entry fields and their visibility.
SET DESCENDING	Changes the descending flag of the controlling index at runtime.
SET DIRCASE	Specifies how directories are accessed on disk.
SET DIRSEPARATOR	Specifies the default separator for directories.
SET EOL	Defines the end-of-line character(s) for ASCII text files.
SET EPOCH	Determines the interpretation of date values without century digits.
SET ERRORLOG	Defines the default error log file.
SET ERRORLOOP	Defines the maximum recursion depth for error handling.
SET ESCAPE	Sets the ESC key as a READ exit key.
SET EVENTMASK	Sets which events should be returned by the Inkey() function.
SET EXACT	Determines the mode for character string comparison.
SET EXCLUSIVE	Sets the global EXCLUSIVE open mode for databases.
SET FILECASE	Specifies how files are accessed on disk.
SET FILTER	Defines a condition for filtering records in the current work area.
SET FIXED	Toggles fixed formatting for displaying numbers in text-mode.
SET FUNCTION	Associates a character string with a function key.
SET HARDCOMMIT	Toggles immediate committing of changes to record buffers.
SET INTENSITY	Toggles usage of enhanced colors for GET and PROMPT
SET KEY	Associates a key with a procedure.
SET MARGIN	Defines the left margin for text-mode print output.
SET MEMOBLOCK	Defines the default block size for memo files.
SET MESSAGE	Defines the screen row for @...PROMPT messages.

SET OPTIMIZE	Toggles filter optimization with indexed databases.
SET PATH	Set the search path for opening files.
SET PRINTER	Enables or disables output to the printer or redirects printer output.
SET SCOREBOARD	Toggles the display of messages from READ and MemoEdit().
SET SOFTSEEK	Enables or disables relative seeking.
SET STRICTREAD	Toggles read optimization for database access.
SET TIME	Specifies the time format for input and display.
SET TRACE	Toggles output of function TraceLog().
SET TYPEAHEAD	Dimensions the size of the keyboard buffer.
SET UNIQUE	Includes or excludes non-unique keys to/from an index.
SET VIDEOMODE	Changes the current video mode of the application.
SET WRAP	Toggles wrapping of the highligh bar in text-mode menus.

Six driver

SX_Decrypt()	Decrypts an encrypted character string.
SX_DtoP()	Converts a Date value into a 3-byte character string.
SX_Encrypt()	Encrypts a character string.
SX_FCompress()	Compresses a file.
SX_FDecompress()	Decompresses a compressed file.
SX_PtoD()	Unpacks a packed 3-byte date value.

Sockets functions

INetAccept()	Waits for an incoming connection on a server side socket.
INetAddress()	Determines the internet address of a remote station.
INetCleanup()	Releases memory resources for sockets.
INetClearError()	Resets the last error code of a socket.
INetClearPeriodCallback()	Voids a callback function for a socket.
INetClearTimeout()	Voids a timout value for a socket.
INetClose()	Closes a connection on a socket.
INetConnect()	Establishes a sockets connection to a server.
INetConnectIP()	Establishes a sockets connection to a server using the IP address.
INetCount()	Returns the number of bytes transferred in the last sockets operation
INetCreate()	Creates a raw, unconnected socket.
INetCRLF()	Returns new line characters used in internet communications.
INetDataReady()	Tests if incoming data is available to be read.
INetDGram()	Creates an unbound datagram oriented socket.
INetDGramBind()	Creates a bound datagram oriented socket.
INetDGramRecv()	Reads data from a datagram socket.
INetDGramSend()	Sends data to a datagram socket.
INetErrorCode()	Returns the last sockets error code.
INetErrorDesc()	Returns a descriptive error message
INetGetAlias()	Retrieves alias names from a server.
INetGetHosts()	Queries IP addresses associated with a name.
INetGetPeriodCallback()	Queries a callback associated with a socket.
INetGetTimeout()	Queries a timout value for a socket.
INetInit()	Initializes the sockets subsystem.
INetPort()	Determines the port number a socket is connected to.
INetRecv()	Reads data from a socket.
INetRecvAll()	Reads all data from a socket.
INetRecvEndblock()	Reads one block of data until an end-of-block marker is detected.
INetRecvLine()	Reads one line of data until CRLF is detected.
INetSend()	Sends data to a socket.
INetSendAll()	Sends all data to a socket.
INetServer()	Creates a server side socket.
INetSetPeriodCallback()	Associates callback information with a socket.

- INetSetTimeout() — Sets a timout value in milliseconds for a socket.

Special operators

- & (macro operator) — Macro operator (unary): compiles a character string at runtime.
- ( ) — Execution or grouping operator.
- -> — Alias operator (binary): identifies a work area.
- : — Send operator (unary): sends a message to an object.
- <|| > — Extended literal code block.
- @ — Pass-by-reference operator (unary): passes variables by reference.
- @() — Function-reference operator (unary): obtains a function pointer.
- [ ] (array) — Array element operator (unary): retrieves array elements.
- [ ] (string) — Character operator (unary): retrieves a character from a string.
- { } — Literal array.
- {=>} — Literal hash.
- {^ } — Literal DateTime value.
- {|| } — Literal code block.

Statements

- ANNOUNCE — Declaration of a module identifier name.
- BEGIN SEQUENCE — Declares a control structure for error handling.
- DO — Executes a function or procedure.
- DO CASE — Executes a block of statements based on one or more conditions.
- DO WHILE — Executes a block of statements while a condition is true.
- EXIT PROCEDURE — Declares a procedure to execeute when a program terminates.
- EXTERNAL — Declares the symbolic name of an external function or procedure for the linker.
- FIELD — Declares a field variable
- FOR — Executes a block of statements a specific number of times.
- FOR EACH — Iterates elements of data types that can be seen as a collection.
- FUNCTION — Declares a function along with its formal parameters.
- GLOBAL — Declares and optionally initializes a GLOBAL memory variable.
- IF — Executes a block of statements based on one or more conditions.
- INIT PROCEDURE — Declares a procedure to execeute when a program starts.
- LOCAL — Declares and optionally initializes a local memory variable.
- MEMVAR — Declares PRIVATE or PUBLIC variables.
- PARAMETERS — Declares PRIVATE function parameters.
- PRIVATE — Creates and optionally initializes a PRIVATE memory variable.
- PROCEDURE — Declares a procedure along with its formal parameters.
- PUBLIC — Creates and optionally initializes a PUBLIC memory variable.
- RETURN — Branches program control to the calling routine.
- RUN — Executes an operating system command.
- STATIC — Declares and optionally initializes a STATIC memory variable.
- SWITCH — Executes one or more blocks of statements.
- TRY...CATCH — Declares a control structure for error handling.
- WITH OBJECT — Identifies an object to receive multiple messages.

Text file functions

- HB_FEof() — Tests if the end-of-file is reached in the currently selected text file.
- HB_FGoBottom() — Moves the file pointer to the last line in a text file.
- HB_FGoto() — Moves the record pointer to a specific line in the currently selected text file.
- HB_FGoTop() — Moves the record pointer to the begin-of-file.
- HB_FInfo() — Retrieves status information about the currently selected text file.

HB_FLastRec()	Returns the number of lines in the currently selected text file.
HB_FReadAndSkip()	Reads the current line and moves the record pointer.
HB_FreadLN()	Reads the current line and without moving the record pointer.
HB_FRecno()	Returns the current line number of the currently selected text file.
HB_FSelect()	Queries or changes the currently selected text file area.
HB_FSkip()	Moves the record pointer in the currently selected text file.
HB_FUse()	Opens or closes a text file in a text file area.

Token functions

AtToken()	Returns the position of the n-th token in a string.
HB_ATokens()	Splits a string into tokens based on a delimiter.
NumToken()	Returns the number of tokens in a string.
RestToken()	Restores the global environment of the incremental tokenizer.
SaveToken()	Saves the global environment of the incremental tokenizer.
Token()	Retrieves the n-th token from a string.
TokenAt()	Returns the start and end position of a token.
TokenEnd()	Tests if tokens can still be found with TokenNext().
TokenExit()	Releases memory resources of the global tokenizer environment.
TokenInit()	Initializes the environment for the incremental tokenizer.
TokenLower()	Changes the first character of tokens to lower case.
TokenNext()	Retrieves the next token from a string.
TokenNum()	Returns the number of tokens in a tokenizer environment.
TokenSep()	Retrieves the separating characters of a token.
TokenUpper()	Changes the first character of tokens to upper case.

Trigonometric functions

ACos()	Calculates the arc cosine.
ASin()	Calculates the arc sine.
ATan()	Calculates the arc tangent.
ATn2()	Calculates the radians of an angle from sine and cosine.
Cos()	Calculates the cosine for an angle.
CosH()	Calculates the hyperbolic cosine for an angle.
Cot()	Calculates the cotangent.
DtoR()	Converts an angle from degrees to radians.
GetPrec()	Retrieves computing precision for trigonometric functions.
Pi()	Returns Pi with highest accuracy.
RtoD()	Converts angles from radians to degrees.
SetPrec()	Specifies the computing precision for trigonometric functions.
Sin()	Calculates the sine.
SinH()	Calculates the hyperbolic sine.
Tan()	Calculates the tangent.
TanH()	Calculates the hyperbolic tangent.

UI functions

AChoice()	Displays a list of items to select one from.
Alert()	Displays a text-mode dialog box with a message.
Browse()	Browse a database file
DbEdit()	Browse records in a table.
GetNew()	Creates a new Get object.
MemoEdit()	Displays and/or edits character strings and memo fields in text mode.
MenuModal()	Activates the menu system represented by a TopBarMenu object.
ReadModal()	Activates editing of @...GET entry fields in text mode.

User-defined RDD

RddRegister()	Registers a user defined Replaceable Database Driver (RDD).
UsrRdd_AreaData()	Queries or attaches user-defined data to a work area of a user-defined RDD.
UsrRdd_AreaResult()	Queries the result of the last operation of a user-defined RDD.
UsrRdd_ID()	Queries the ID of a user-defined RDD.
UsrRdd_RddData()	Queries or attaches user-defined data a user-defined RDD.
UsrRdd_SetBof()	Sets the BoF() flag in a work area of a user-defined RDD.
UsrRdd_SetBottom()	Defines the bottom scope value for a work area of a user-defined RDD.
UsrRdd_SetEof()	Sets the Eof() flag in a work area of a user-defined RDD.
UsrRdd_SetFound()	Sets the Found() flag in a work area of a user-defined RDD.
UsrRdd_SetTop()	Defines the top value for a work area of a user-defined RDD.

Windows (text mode)

WAClose()	Closes all windows.
WBoard()	Determines the area that can be used for displaying windows.
WBox()	Draws a frame around the current window.
WCenter()	Centers a window on the screen or makes it entirely visible.
WClose()	Closes the current window and selects the next window as current window.
WCol()	Returns the left column position of the selected window.
WfCol()	Returns the left column position of the usable area in a formatted window.
WfLastCol()	Returns the right column position of the usable area in a formatted window.
WfLastRow()	Returns the bottom row position of the usable area in a formatted window.
WFormat()	Defines the usable display area inside the current window.
WfRow()	Returns the top row position of the usable area in a formatted window.
WInfo()	Returns all coordinates of the current window.
WLastCol()	Returns the right column position of the selected window.
WLastRow()	Returns the bottom row position of the selected window.
WMode()	Determines on which side windows are allowed to be moved off the screen.
WMove()	Changes the upper left coordinate for the current window.
WMSetPos()	Moves the mouse cursor to a new position in a window.
WNum()	Returns the largest window ID of all windows.
WOpen()	Creates a new window.
WRow()	Returns the top row position of the selected window.
WSelect()	Returns and/or selects an open window as the current window.
WSetMouse()	Determines the visibility and/or position of the mouse cursor.
WSetMove()	Toggles the setting for moving windows interactively.
WSetShadow()	Defines the shadow color for windows.
WStack()	Returns window IDs of all open windows.
WStep()	Sets the increments for horizontal and vertical window movement.

xHarbour extensions

#if	Compile a section of code based on a condition.
#pragma	Controls compiler switches at compile time.
#uncommand | #untranslate	Voids a previously defined #command or #translate directive.
#xuncommand | #xuntranslate	Voids a previously defined #xcommand or #xtranslate directive.
& (bitwise AND)	Bitwise AND operator (binary): performs a logical AND operation.
(struct)	Creates a new structure object
<<	Left-shift operator (binary): shifts bits to the left.

<table>
<tr><td><|| ></td><td>Extended literal code block.</td></tr>
<tr><td>>></td><td>Right-shift operator (binary): shifts bits to the right.</td></tr>
<tr><td>@()</td><td>Function-reference operator (unary): obtains a function pointer.</td></tr>
<tr><td>ACCESS</td><td>Declares an ACCESS method of a class.</td></tr>
<tr><td>ALenAlloc()</td><td>Determines for how much array elements memory is pre-allocated.</td></tr>
<tr><td>AnsiToHtml()</td><td>Inserts HTML character entities into an ANSI text string,</td></tr>
<tr><td>ASizeAlloc()</td><td>Pre-allocates memory for an array.</td></tr>
<tr><td>ASSIGN</td><td>Declares an ASSIGN method of a class.</td></tr>
<tr><td>ASSOCIATE CLASS</td><td>Defines a scalar class for a native data type.</td></tr>
<tr><td>AtSkipStrings()</td><td>Locates the position of a substring within a character string.</td></tr>
<tr><td>C Structure class</td><td>Abstract class for C structure support.</td></tr>
<tr><td>CallDll()</td><td>Executes a function located in a dynamically loaded external library.</td></tr>
<tr><td>CharHist()</td><td>Creates a histogram of characters in a character string</td></tr>
<tr><td>CharRLL()</td><td>Rotates bits in a character string to the left.</td></tr>
<tr><td>CharRLR()</td><td>Rotates bits in a character string to the right.</td></tr>
<tr><td>CharSHL()</td><td>Shifts bits in a character string to the left.</td></tr>
<tr><td>CharSHR()</td><td>Shifts bits in a character string to the right.</td></tr>
<tr><td>CharSList()</td><td>Removes duplicate characters from a string and sorts the result.</td></tr>
<tr><td>CharSub()</td><td>Creates a string by subtracting ASCII codes of two strings.</td></tr>
<tr><td>CLASS</td><td>Declares the class function of a user-defined class.</td></tr>
<tr><td>CLASSDATA</td><td>Declares a class variable of a class.</td></tr>
<tr><td>CLASSMETHOD</td><td>Declares the symbolic name of a class method.</td></tr>
<tr><td>CLOSE LOG</td><td>Closes all open log channels.</td></tr>
<tr><td>CosH()</td><td>Calculates the hyperbolic cosine for an angle.</td></tr>
<tr><td>CreateObject()</td><td>Instantiates a new OLE Automation object.</td></tr>
<tr><td>CStr()</td><td>Converts a value to a character string.</td></tr>
<tr><td>CStrToVal()</td><td>Converts a character string to a value of specific data type.</td></tr>
<tr><td>CtoT()</td><td>Converts a character string into a DateTime value</td></tr>
<tr><td>CurDirX()</td><td>Returns the current directory of a drive including directory separators.</td></tr>
<tr><td>DATA</td><td>Declares an instance variable of a class.</td></tr>
<tr><td>DateTime()</td><td>Returns the current date and time from the operating system.</td></tr>
<tr><td>DbCopyExtStruct()</td><td>Creates a structure extended database file.</td></tr>
<tr><td>DbCopyStruct()</td><td>Creates a new database based on the current database structure.</td></tr>
<tr><td>DbJoin()</td><td>Merges records of two work areas into a new database.</td></tr>
<tr><td>Dblist()</td><td>Displays records of a work area to the console, printer or file.</td></tr>
<tr><td>DbSkipper()</td><td>Helper function for browse objects to skip a database</td></tr>
<tr><td>DbSort()</td><td>Creates a new, physically sorted database.</td></tr>
<tr><td>DbTotal()</td><td>Creates a new database summarizing numeric fields by an expression.</td></tr>
<tr><td>DbUpdate()</td><td>Updates records in the current work area from a second work area.</td></tr>
<tr><td>Default()</td><td>Assigns a default value to a variable.</td></tr>
<tr><td>DefPath()</td><td>Returns the SET DEFAULT directory.</td></tr>
<tr><td>DELEGATE</td><td>Declares a message to be directed to a contained object.</td></tr>
<tr><td>DESTRUCTOR</td><td>Declares a method to be called by the garbage collector.</td></tr>
<tr><td>DirectoryRecurse()</td><td>Loads file information recursively into a two-dimensional array.</td></tr>
<tr><td>DisableWaitLocks()</td><td>Toggles the exclusive file opening mode.</td></tr>
<tr><td>DiskChange()</td><td>Changes the current disk drive.</td></tr>
<tr><td>DiskUsed()</td><td>Returns the used storage space of a disk drive in bytes.</td></tr>
<tr><td>DispOutAt()</td><td>Displays a value on the screen at a certain position.</td></tr>
<tr><td>DispOutAtSetPos()</td><td>Toggles update of the screen cursor with DispOutAt().</td></tr>
<tr><td>DllCall()</td><td>Executes a function located in a dynamically loaded external library.</td></tr>
<tr><td>DllExecuteCall()</td><td>Executes a DLL function via call template.</td></tr>
<tr><td>DllLoad()</td><td>Loads a DLL file into memory.</td></tr>
<tr><td>DllPrepareCall()</td><td>Creates a call template for an external DLL function.</td></tr>
<tr><td>DllUnload()</td><td>Releases a dynamically loaded external DLL from memory.</td></tr>
<tr><td>ENABLE TYPE CLASS</td><td>Activates scalar classes for native data tyes.</td></tr>
<tr><td>ERROR HANDLER</td><td>Declares the method for error handling within a class.</td></tr>
<tr><td>EXPORTED:</td><td>Declares the EXPORTED attribute for a group of member variables and/or methods.</td></tr>
<tr><td>EXTEND CLASS...WITH DATA</td><td>Adds a new member variable to an existing class.</td></tr>
</table>

EXTEND CLASS...WITH METHOD	Adds a new method to an existing class.
FCharCount()	Counts the number of characters in a text file ignoring white space characters.
FieldDec()	Retrieves the number of decimal places of a field variable.
FieldLen()	Retrieves the number of bytes occupied by a field variable.
FieldType()	Retrieves the data type of a field variable.
FileReader()	Returns a new TStreamFileReader object.
FileStats()	Retrieves file information for a single file.
FileWriter()	Returns a new TStreamFileWriter object.
FLineCount()	Counts the lines in an ASCII text file.
FOR EACH	Iterates elements of data types that can be seen as a collection.
FParse()	Parses a delimited text file and loads it into an array.
FParseEx()	Parses a delimited text file and loads it into an array (optimized).
FParseLine()	Parses one line of a delimited text and loads it into an array.
FreeLibrary()	Releases a dynamically loaded external DLL from memory.
FWordCount()	Counts the words in a text file.
GetActiveObject()	Retturns an instantiated OLE Automation object.
GetCurrentThread()	Retrieves the handle of the current thread.
GetDefaultPrinter()	Retrieves the name of a computer's default printer.
GetLastError()	Retrieves the error code of the last dynamically called DLL function.
GetPrinters()	Retrieves information about available printers.
GetProcAddress()	Retrieves the memory address of a function in a dynamically loaded DLL.
GetRegistry()	Retrieves the value of a registry entry
GetSystemThreadID()	Retrieves the numeric system Thread ID of a thread.
GetThreadID()	Retrieves the numeric application Thread ID of a thread.
GetVolInfo()	Retrieves the volume label of a disk.
GLOBAL	Declares and optionally initializes a GLOBAL memory variable.
HaaDelAt()	Removes a key/value pair from an associative array.
HaaGetKeyAt()	Retrieves the key from an associative array by its ordinal position.
HaaGetPos()	Retrieves the ordinal position of a key in an associative array.
HaaGetRealPos()	Retrieves the sort order of a key in an associative array.
HaaGetValueAt()	Retrieves the value from an associative array by its ordinal position.
HaaSetValueAt()	Changes the value in an associative array by its ordinal position.
HAllocate()	Pre-allocates memory for a large hash.
HAS	Searches a character string for a matching regular expression.
Hash()	Creates a new hash.
HbConsoleLock()	Locks the console for the current thread
HbConsoleUnlock()	Releases the console lock.
HBObject()	Abstract base class for user-defined classes.
HBPersistent()	Abstract base class for user-defined persistent object.
HB_AExpressions()	Parses a character string into an array of macro expressions.
HB_AnsiToOem()	Converts a character string from the ANSI to the OEM character set.
HB_AParams()	Collects values of all parameters passed to a function, method or procedure.
HB_ArgC()	Returns the number of command line arguments.
HB_ArgCheck()	Checks if an internal switch is set on the command line.
HB_ArgString()	Retrieves the vale of an internal switch set on the command line.
HB_ArgV()	Retrieves the value of a command line argument.
HB_ArrayBlock()	Creates a set/get code block for an array.
HB_ArrayId()	Returns a unique identifier for an array or an object variable.
HB_ArrayToStructure()	Converts an array to a binary C structure.
HB_AtX()	Locates a substring within a character string based on a regular expression.
HB_BackGroundActive()	Queries and/or changes the activity of a single background task.
HB_BackGroundAdd()	Adds a new background task.
HB_BackGroundDel()	Removes a background task from the internal task list.
HB_BackGroundReset()	Resets the internal counter of background tasks.
HB_BackGroundRun()	Enforces execution of one or all background tasks.
HB_BackGroundTime()	Queries or changes the wait interval in milliseconds after which the task is executed.

HB_Base64Decode()	Decodes a base 64 encoded character string.
HB_Base64DecodeFile()	Decodes a base 64 encoded file.
HB_Base64Encode()	Encodes a character string base 64.
HB_Base64EncodeFile()	Encodes a file base 64.
HB_BitAnd()	Performs a bitwise AND operation with numeric integer values.
HB_BitIsSet()	Checks if a bit is set in a numeric integer value.
HB_BitNot()	Performs a bitwise NOT operation with a numeric integer value.
HB_BitOr()	Performs a bitwise OR operation with numeric integer values.
HB_BitReset()	Sets a bit in a numeric integer value to 0.
HB_BitSet()	Sets a bit in a numeric integer value to 1.
HB_BitShift()	Shifts bits in a numeric integer value.
HB_BitXOr()	Performs a bitwise XOR operation with numeric integer values.
HB_BldLogMsg()	Converts a list of parameters into a text string.
HB_BuildDate()	Retrieves the formatted build date of the xHarbour compiler
HB_BuildInfo()	Retrieves build information of the xHarbour compiler.
HB_CheckSum()	Calculates the checksum for a stream of data using the Adler32 algorithm.
HB_Clocks2Secs()	Calculates seconds from CPU ticks.
HB_CloseProcess()	Closes a child process
HB_ClrArea()	Replaces a color attribute in a screen region.
HB_CmdArgArgV()	Returns the first command line argument (EXE file name).
HB_ColorIndex()	Extracts a color value from a color string.
HB_ColorToN()	Converts a color value to a numeric color attribute.
HB_Compiler()	Retrieves the version of the C compiler shipped with xHarbour.
HB_Compress()	Compresses a character string (ZIP).
HB_CompressBufLen()	Calculates the buffer size required for compression.
HB_CompressError()	Returns the error code of the last (un)compression.
HB_CompressErrorDesc()	Returns an error description from a numeric error code.
Hb_CRC32()	Calculates the checksum for a stream of data using the CRC 32 algorithm.
HB_CreateLen8()	Converts a numeric value to an eight byte character string in network byte order.
HB_Crypt()	Encrypts a character string.
HB_Decode()	Provides a functional equivalent for the DO CASE statement.
HB_DecodeOrEmpty()	Provides a functional equivalent for the DO CASE statement.
HB_Decrypt()	Decrypts an encrypted character string.
HB_DeserialBegin()	Initiates deserialization of a group of variables of simple or complex data types.
HB_DeSerialize()	Converts a binary string back to its original data type.
HB_DeserializeSimple()	Deserializes values of simple data types.
HB_DeserialNext()	Deserialization the next variable of simple or complex data types.
HB_DiskSpace()	Returns the free storage space for a disk drive by drive letter.
HB_DumpVar()	Converts a value to a character string holding human readable text.
HB_EnumIndex()	Returns the current ordinal position of a FOR EACH iteration.
HB_Exec()	Executes a function, procedure or method from its pointer.
HB_ExecFromArray()	Executes a function, procedure or method indirectly.
HB_FCommit()	Forces a disk write.
HB_FCreate()	Creates and/or opens a binary file.
HB_FEof()	Tests if the end-of-file is reached in the currently selected text file.
HB_FGoBottom()	Moves the file pointer to the last line in a text file.
HB_FGoto()	Moves the record pointer to a specific line in the currently selected text file.
HB_FGoTop()	Moves the record pointer to the begin-of-file.
HB_FInfo()	Retrieves status information about the currently selected text file.
HB_FLastRec()	Returns the number of lines in the currently selected text file.
HB_FNameMerge()	Composes a file name from individual components.
HB_FNameSplit()	Splits a file name into individual components.
HB_FReadAndSkip()	Reads the current line and moves the record pointer.
HB_FReadLine()	Extracts the next line from a text file.
HB_FreadLN()	Reads the current line and without moving the record pointer.
HB_FRecno()	Returns the current line number of the currently selected text file.

HB_FSelect()	Queries or changes the currently selected text file area.
HB_FSize()	Returns the size of a file in bytes.
HB_FSkip()	Moves the record pointer in the currently selected text file.
HB_FTempCreate()	Creates and opens a temporary file.
HB_FuncPtr()	Obtains the pointer to a function or procedure.
HB_FUse()	Opens or closes a text file in a text file area.
HB_F_Eof()	Tests for End-of-file with binary files.
HB_GCAll()	Scans the memory and releases all garbage memory blocks.
HB_GCStep()	Invokes the garbage collector for one collection cycle.
HB_GetLen8()	Retrieves the length of data in a serialized binary string.
HB_IdleAdd()	Adds a background task for being executed during idle states.
HB_IdleDel()	Removes a task from the list of idle tasks.
HB_IdleReset()	Resets the internal counter of idle tasks.
HB_IdleSleep()	Halts idle task processing for a number of seconds.
HB_IdleSleepMSec()	Queries or changes the default time interval for idle task processing.
HB_IdleState()	Signals an idle state.
HB_IdleWaitNoCPU()	Toggles the mode for CPU usage in Idle wait states.
HB_IsArray()	Tests if the value returned by an expression is an array.
HB_IsBlock()	Tests if the value returned by an expression is a Code block.
HB_IsByRef	Tests if a parameter is passed by reference.
HB_IsDate()	Tests if the value returned by an expression is a Date.
HB_IsDateTime()	Tests if the value returned by an expression is a DateTime value.
HB_IsHash()	Tests if the value returned by an expression is a Hash.
HB_IsLogical()	Tests if the value returned by an expression is a logical value.
HB_IsMemo()	Tests if the value returned by an expression is a Memo value.
HB_IsNIL()	Tests if the value returned by an expression is NIL.
HB_IsNull()	Tests if the value returned by an expression is empty.
HB_IsNumeric()	Tests if the value returned by an expression is a numeric value.
HB_IsObject()	Tests if the value returned by an expression is an object.
HB_IsPointer()	Tests if the value returned by an expression is a pointer.
HB_IsRegExString()	Checks if a character string is a compiled regular expression.
HB_IsString()	Tests if the value returned by an expression is a character string.
HB_LangErrMsg()	Returns language specific error messages.
HB_LangMessage()	Returns language specific messages or text strings.
HB_LangName()	Returns the currently selected language.
HB_LangSelect()	Queries or changes the current national language .
HB_LibDo()	Executes a function located in a dynamically loaded xHarbour DLL.
HB_LogDateStamp()	Returns a character string holding a date stamp.
HB_MacroCompile()	Compiles a macro string into a PCode sequence.
HB_MD5()	Calculates a message digest for a stream of data using the MD5 algorithm.
HB_MD5File()	Calculates a message digest for a file using the MD5 algorithm.
HB_MultiThread()	Checks if an application is created for multi-threading.
HB_MutexCreate()	Creates a Mutex.
HB_MutexLock()	Obtains a permanent lock on a Mutex.
HB_MutexTimeoutLock()	Tries to obtain a lock on a Mutex with timeout.
HB_MutexTryLock()	Tries to obtain a permanent lock on a Mutex.
HB_MutexUnlock()	Unlocks a Mutex.
HB_ObjMsgPtr()	Retrieves the pointer to a method.
HB_OemToAnsi()	Converts a character string from the OEM to the ANSI character set.
HB_OpenProcess()	Opens a child process.
HB_OsDriveSeparator()	Returns the operating specific drive separator character.
HB_OsError()	Returns the operating specific error code of the last low-level file operation.
HB_OsNewLine()	Returns the end-of-line character(s) to use with the current operating system.
HB_OsPathDelimiters()	Returns the operating specific characters for paths.
HB_OsPathListSeparator()	Returns the operating specific separator character for a path list.
HB_OsPathSeparator()	Returns the operating specific separator character for a path.
HB_PCodeVer()	Retrieves the PCode version of the current xHarbour build.

HB_Pointer2String()	Reads bytes from a pointer into a character string.
HB_ProcessValue()	Retrieves the return value of a child process.
HB_QSelf()	Retrieves the !Iself!EI object during method execution.
HB_QWith()	Retrieves the !Iwith!EI object during WITH OBJECT execution.
HB_Random()	Generates a random number between two boundaries.
HB_RandomInt()	Generates a random integer number between two boundaries.
HB_RandomSeed()	Sets the seed for generating random numbers.
HB_ReadIni()	Reads an INI file from disk.
HB_ReadLine()	Scans a formatted character string or memo field for text lines.
HB_RegEx()	Searches a string using a regular expression
HB_RegExAll()	Parses a string and fills an array with parsing information.
HB_RegExAtX()	Parses a string and fills an array with parsing information.
HB_RegExComp()	Compiles a regular expression into a binary search pattern.
HB_RegExMatch()	Tests if a string contains a substring using a regular expression
HB_RegExReplace()	Searches and replaces characters within a character string using a regular expression.
HB_RegExSplit()	Parses a string using a regular expression and fills an array.
HB_ResetWith()	Replaces the !Iwith!EI object during WITH OBJECT execution.
HB_RestoreBlock()	Converts binary information back to a code block.
HB_SaveBlock()	Utility function for code block serialization.
HB_Serialize()	Converts an arbitrary value to a binary string.
HB_SerializeSimple()	Serializes values of simple data types.
HB_SerialNext()	Returns the position of the next chunk of binary data to retrieve.
HB_SetCodePage()	Queries or changes the current code page.
HB_SetIniComment()	Defines the delimiting characters for comments and inline comments.
HB_SetKeyArray()	Associates a code block with multiple keys.
HB_SetKeyCheck()	Evaluates a code block associated with a key.
HB_SetKeyGet()	Retrieves code blocks associated with a key.
HB_SetKeySave()	Queries or changes all SetKey() code blocks.
HB_SetMacro()	Enables or disables runtime behavior of the macro compiler.
HB_SetWith()	Changes the !Iwith!EI object during WITH OBJECT execution.
HB_Shadow()	Displays a shadow around a rectangular area on the screen.
HB_SizeofCStructure()	Calculates the amount of memory required to store a C structure.
HB_String2Pointer()	Obtains the pointer for a character string.
HB_StructureToArray()	Converts values contained in a binary C structure string to an array.
HB_SysLogClose()	Closes the log file of the operating system.
HB_SysLogMessage()	Adds a new entry to the log file of the operating system.
HB_SysLogOpen()	Opens the log file of the operating system
HB_ThisArray()	Retrieves an array or object from its pointer.
HB_Translate()	Converts a character string from one code page to another one.
HB_Uncompress()	Uncompresses a compressed character string (ZIP).
HB_UUDecode()	Decodes a UUEncoded character string.
HB_UUDecodeFile()	Decodes a UUEncoded file.
HB_UUEncode()	UUEncodes a character string.
HB_UUEncodeFile()	UUEncodes a file.
HB_ValToStr()	Converts values of simple data types to character string.
HB_VMExecute()	Executes a PCode string.
HB_VMMode()	Indicates the creation mode of the xHarbour virtual machine.
HB_WithObjectCounter()	Determines the nesting level of WITH OBJECT statements.
HB_WriteIni()	Creates an INI file and writes hash data to it.
HB_XmlErrorDesc()	Retrieves a textual error description for XML file parsing errors.
HClone()	Creates an entire copy of a hash.
HCopy()	Copies key/value pairs from a hash into another hash.
HDel()	Removes a key/value pair from the hash by its key.
HDelAt()	Removes a key/value pair from the hash by its ordinal position.
HEval()	Evaluates a code block with each hash element.
HexToNum()	Converts a Hex string to a numeric value.
HexToStr()	Converts a Hex encoded character string to an ASCII string.
HFill()	Copies the same value into all key/value pairs.

HGet()	Retrieves the value associated with a specified key.
HGetAACompatibility()	Checks if a hash is compatible with an associative array.
HGetAutoAdd()	Retrieves the AutoAdd attribute of a hash.
HGetCaseMatch()	Retrieves the case sensitivity attribute of a hash.
HGetKeyAt()	Retrieves the key from a hash by its ordinal position.
HGetKeys()	Collects all keys from a hash in an array.
HGetPairAt()	Retrieves a key/value pair from a hash by its ordinal position.
HGetPartition()	Checks if a hash is partitioned.
HGetPos()	Retrieves the ordinal position of a key in a hash.
HGetVaaPos()	Retrieves the sort order of all keys in an associative array.
HGetValueAt()	Retrieves the value from a hash by its ordinal position.
HGetValues()	Collects all values from a hash in an array.
HHasKey()	Determines if a key is present in a hash.
HIDDEN:	Declares the HIDDEN attribute for a group of member variables and/or methods.
HMerge()	Merges the contents of an entire hash into another hash.
Hour()	Extracts the hour from a DateTime value
HScan()	Searches a value in a hash.
HSet()	Associates a value with a key in a hash.
HSetAACompatibility()	Enables or disables associative array compatibility for an empty hash.
HSetAutoAdd()	Changes the AutoAdd attribute of a hash.
HSetCaseMatch()	Changes the case sensitivity attribute of a hash.
HSetPartition()	Partitions a linear hash for improved performance.
HSetValueAt()	Changes the value in a hash by its ordinal position.
HS_Add()	Adds a text string entry to a HiPer-SEEK index file.
HS_Close()	Closes a HiPer-SEEK index file.
HS_Create()	Creates a new HiPer-SEEK index file.
HS_Delete()	Marks an index entry as deleted in a HiPer-SEEK index file.
HS_Filter()	Uses a HiPer-SEEK index file as filter for a work area
HS_IfDel()	Checks if a HiPer-SEEK index entry is marked as deleted
HS_Index()	Creates a new HiPer-SEEK index file and fills it with index entries.
HS_KeyCount()	Returns the number of index entries in a HiPer-SEEK index file.
HS_Next()	Searches a HiPer-SEEK index file for a matching index entry.
HS_Open()	Opens a HiPer-SEEK index file.
HS_Replace()	Changes a HiPer-SEEK index entry.
HS_Set()	Defines a search string for subsequent HS_Next() calls.
HS_Undelete()	Removes the deletion mark from an index entry in a HiPer-SEEK index file.
HS_Verify()	Verifies a HS_Next() match against the index key.
HS_Version()	Returns version information for HiPer-SEEK functions.
HtmlToAnsi()	Converts an HTML formatted text string to the ANSI character set.
HtmlToOem()	Converts an HTML formatted text string to the OEM character set
IN	Searches a value in another value.
INetAccept()	Waits for an incoming connection on a server side socket.
INetAddress()	Determines the internet address of a remote station.
INetCleanup()	Releases memory resources for sockets.
INetClearError()	Resets the last error code of a socket.
INetClearPeriodCallback()	Voids a callback function for a socket.
INetClearTimeout()	Voids a timout value for a socket.
INetClose()	Closes a connection on a socket.
INetConnect()	Establishes a sockets connection to a server.
INetConnectIP()	Establishes a sockets connection to a server using the IP address.
INetCount()	Returns the number of bytes transferred in the last sockets operation
INetCreate()	Creates a raw, unconnected socket.
INetCRLF()	Returns new line characters used in internet communications.
INetDataReady()	Tests if incoming data is available to be read.
INetDGram()	Creates an unbound datagram oriented socket.
INetDGramBind()	Creates a bound datagram oriented socket.
INetDGramRecv()	Reads data from a datagram socket.

INetDGramSend()	Sends data to a datagram socket.
INetErrorCode()	Returns the last sockets error code.
INetErrorDesc()	Returns a descriptive error message
INetGetAlias()	Retrieves alias names from a server.
INetGetHosts()	Queries IP addresses associated with a name.
INetGetPeriodCallback()	Queries a callback associated with a socket.
INetGetTimeout()	Queries a timout value for a socket.
INetInit()	Initializes the sockets subsystem.
INetPort()	Determines the port number a socket is connected to.
INetRecv()	Reads data from a socket.
INetRecvAll()	Reads all data from a socket.
INetRecvEndblock()	Reads one block of data until an end-of-block marker is detected.
INetRecvLine()	Reads one line of data until CRLF is detected.
INetSend()	Sends data to a socket.
INetSendAll()	Sends all data to a socket.
INetServer()	Creates a server side socket.
INetSetPeriodCallback()	Associates callback information with a socket.
INetSetTimeout()	Sets a timout value in milliseconds for a socket.
INIT LOG	Initializes the Log system and opens requested log channels.
INLINE METHOD	Declares and implements an inline method that spans across multiple lines.
IsAffirm()	Converts "Yes" in a national language to a logical value.
IsAlNum()	Checks if the first character of a string is alpha-numeric.
IsAscii()	Checks if the first character of a string is a 7-bit ASCII character.
IsCntrl()	Checks if the first character of a string is a control character.
IsDirectory()	Checks if a character string contains the name of an existing directory.
IsDisk()	Verify if a drive is ready
IsGraph()	Checks if the first character of a string is a 7-bit graphical ASCII character.
IsLocked()	Checks if a record is locked.
IsNegative()	Converts "No" in a national language to a logical value.
IsPrint()	Checks if the first character of a string is a printable 7-bit ASCII character.
IsPunct()	Checks if the first character of a string is a punctuation character.
IsSameThread()	Compares two thread handles.
IsSpace()	Checks if the first character of a string is a white-space character.
IsValidThread()	Checks if an expression is the thread handle of a running thread.
IsXDigit()	Checks if the first character of a string is a hexadecimal digit.
JoinThread()	Suspends the current thread until a second thread has terminated.
KillAllThreads()	Kills all running threads except for the main thread.
KillThread()	Kills a running thread.
LenNum()	Returns the number of characters a numeric value needs for display.
LibFree()	Releases a dynamically loaded xHarbour DLL from memory.
LibLoad()	Loads an xHarbour DLL file into memory.
LIKE	Compares a character string with a regular expression.
LoadLibrary()	Loads an external DLL file into memory.
LOG	Sends a message to open log channels.
Memory()	Returns memory statistics.
MESSAGE	Declares a message name for a method.
METHOD (declaration)	Declares the symbolic name of a method.
METHOD (implementation)	Declares the implementation code of a method.
METHOD...OPERATOR	Declares a method to be executed with an operator.
METHOD...VIRTUAL	Declares a method as virtual.
Minute()	Extracts the minute from a DateTime value
NationMsg()	Returns application specific messages.
NetAppend()	Appends a new record to a database open in shared mode in a work area.
NetCommitAll()	Writes database and index buffers of all used work areas to disk.
NetDbUse()	Opens a database file for shared access in a work area.
NetDelete()	Marks records for deletion.
NetError()	Determines if a Net*() function has failed.

NetFileLock()	Applies a file lock to an open, shared database.
NetFunc()	Evaluates a code block until a timeout period expires.
NetLock()	Applies locks to a database file or record with timeout.
NetOpenFiles()	Opens databases and associated index files.
NetRecall()	Recalls a record previousy marked for deletion.
NetRecLock()	Locks the current record for write access.
Notify()	Resumes a single thread blocked by a particular Mutex.
NotifyAll()	Resumes all threads blocked by a particular Mutex.
NumAndX()	Performs bitwise AND operations for a list of integer values.
NumMirrX()	Mirrors bits of a numeric integer value.
NumNotX()	Performs a bitwise NOT operation with a numeric integer value.
NumOrX()	Performs a bitwise OR for a list of integer values.
NumRolX()	Rotates bits of a numeric integer value to the left.
NumToHex()	Converts a numeric value or a pointer to a Hex string.
NumXorX()	Performs a bitwise XOR operation with two numeric integer values.
OemToHtml()	Inserts HTML character entities into an OEM text string.
Ole2TxtError()	Returns the last OLE error code as character string.
OleError()	Returns the last OLE error code.
OPERATOR	Overloads an operator and declares a method to be invoked for it.
OrdCount()	Determines the number of indexes open in a work area.
OrdCustom()	Determines if an index is a custom index.
OrdFindRec()	Searches a record number in the controlling index.
OrdKeyRelPos()	Returns or sets the relative position of the current record.
OrdSkipRaw()	Moves the record pointer via the controlling index.
OrdWildSeek()	Searches a value in the controlling index using wild card characters.
Os_IsWin2000()	Checks if the application is running on Windows 2000.
Os_IsWin2000_Or_Later()	Checks if the application is running on Windows version 2000 or later
Os_IsWin2003()	Checks if the application is running on Windows 2003.
Os_IsWin95()	Checks if the application is running on Windows 95.
Os_IsWin98()	Checks if the application is running on Windows 98.
Os_IsWin9X()	Checks if the application is running on a Windows 9x platform.
Os_IsWinME()	Checks if the application is running on Windows ME.
Os_IsWinNT()	Checks if the application is running on a Windows NT platform.
Os_IsWinNT351()	Checks if the application is running on Windows NT 3.51.
Os_IsWinNT4()	Checks if the application is running on Windows NT 4.0.
Os_IsWinVista()	Checks if the application is running on Windows Vista.
Os_IsWinXP()	Checks if the application is running on Windows XP.
Os_IsWtsClient()	Checks if the application is running on a Windows Terminal Server client.
Os_NetRegOk()	Checks for correct network registry settings on Windows platforms.
OS_NetVRedirOk()	Checks for the correct VREDIR.VXD file.
Os_VersionInfo()	Retrieves specific version information about the operating system.
OVERRIDE METHOD	Replaces a method in an existing class.
pragma pack()	Defines the byte alignment for the next structure declaration.
PrgExpToVal()	Converts a character string obtained from ValToPrgExp() back to the original data type.
PrinterExists()	Checks if a particular printer is installed.
PrinterPortToName()	Retrieves the name of the printer connected to a printer port.
PrintFileRaw()	Prints a file to a Windows printer in RAW mode.
ProcFile()	Determines the current PRG source code file.
PROTECTED:	Declares the PROTECTED attribute for a group of member variables and methods.
PValue()	Retrieves the value of a parameter passed to a function, method or procedure.
QueryRegistry()	Checks if a particular registry key with specified value exists.
RAscan()	Searches a value in an array beginning with the last element.
RddInfo()	Queries and/or changes configuration data of RDDs.
RddRegister()	Registers a user defined Replaceable Database Driver (RDD).
Scrollfixed()	Scrolls a screen region horizontally and/or vertically.
SecondsCpu()	Returns the CPU time used by the current process.

SecondsSleep()	Suspends thread execution for a number of seconds.
SET AUTOPEN	Toggles automatic opening of a structural index file.
SET AUTORDER	Defines the default controlling index for automatically opened index files.
SET AUTOSHARE	Defines network detection for shared file access.
SET BACKGROUND TASKS	Enables or disables the activity of background tasks.
SET BACKGROUNDTICK	Defines the processing interval for background tasks.
SET DBFLOCKSCHEME	Selects the locking scheme for shared database access.
SET DIRCASE	Specifies how directories are accessed on disk.
SET DIRSEPARATOR	Specifies the default separator for directories.
SET EOL	Defines the end-of-line character(s) for ASCII text files.
SET ERRORLOG	Defines the default error log file.
SET ERRORLOOP	Defines the maximum recursion depth for error handling.
SET FILECASE	Specifies how files are accessed on disk.
SET HARDCOMMIT	Toggles immediate committing of changes to record buffers.
SET LOG STYLE	Selects a style for logging data to log channels.
SET STRICTREAD	Toggles read optimization for database access.
SET TIME	Specifies the time format for input and display.
SET TRACE	Toggles output of function TraceLog().
SetErrorMode()	Queries or changes the behavior with operating system errors.
SetLastError()	Sets a numeric value as last error code.
SetNetDelay()	Queries or changes the timeout period for Net*() functions.
SetNetMsgColor()	Queries or changes the color for displaying failure messages of Net*() functions.
SetNewDate()	Changes the system date from Numeric values.
SetNewTime()	Changes the system time from Numeric values.
SetRegistry()	Creates a key/value pair in the registry.
SinH()	Calculates the hyperbolic sine.
StartThread()	Starts a new thread.
StopThread()	Stops a thread from outside.
StoT()	Converts a "YYYYMMDDhhmmss.ccc" formatted string to a DateTime value
Strdel()	Deletes characters from a string based on a mask string.
StringToLiteral()	Creates a literal character string from a string.
StrToHex()	Converts a character string to a Hex string.
Subscribe()	Subscribes for notifications on a Mutex.
SubscribeNow()	Subscribes for notifications on a Mutex and discards pending notifications.
SWITCH	Executes one or more blocks of statements.
TanH()	Calculates the hyperbolic tangent.
TBMouse()	Moves the browse cursor to the mouse pointer.
ThreadSleep()	Puts a thread to sleep.
Throw()	Throws an exception.
THtmlCleanup()	Releases memory tables required for HTML classes.
THtmlDocument()	Creates a new THtmlDocument object.
THtmlInit()	Initializes memory tables required for HTML classes.
THtmlIsValid()	Validates a HTML tag name and attribute.
THtmlIterator()	Creates a new THtmlIterator object.
THtmlIteratorRegEx()	Creates a new THtmlIteratorRegEx object.
THtmlIteratorScan()	Creates a new THtmlIteratorScan object.
THtmlNode()	Creates a new THtmlNode object.
TIpClient()	Abstract class for internet communication.
TIpClientFtp()	Creates a new TIpClientFtp object.
TIpClientHttp()	Creates a new TIpClientHttp object.
TIpClientPop()	Creates a new TIpClientPop object.
TIpClientSmtp()	Creates a new TIpClientSmtp object.
TIpMail()	Creates a new TIpMail object.
TokenExit()	Releases memory resources of the global tokenizer environment.
TokenNum()	Returns the number of tokens in a tokenizer environment.
TraceLog()	Traces and logs the contents of one or more variables.
TRY...CATCH	Declares a control structure for error handling.

TStream()	Abstract class for low-level file data streams.
TStreamFileReader()	Creates a new TStreamFileReader object.
TStreamFileWriter()	Creates a new TStreamFileWriter object.
TtoC()	Converts a DateTime value to a character string in SET DATE and SET TIME format.
TtoS()	Converts a Date value to a character string in YYYYMMDDhhmmss.ccc format.
TUrl()	Creates a new TUrl object.
TXmlDocument()	Creates a new TXmlDocument object.
TXmlIterator()	Creates a new TXmlIterator object.
TXmlIteratorRegEx()	Creates a new TXmlIteratorRegEx object.
TXmlIteratorScan()	Creates a new TXmlIteratorScan object.
TXmlNode()	Creates a new TXmlNode object.
typedef struct	Declares a new structure in C syntax.
UsrRdd_AreaData()	Queries or attaches user-defined data to a work area of a user-defined RDD.
UsrRdd_AreaResult()	Queries the result of the last operation of a user-defined RDD.
UsrRdd_ID()	Queries the ID of a user-defined RDD.
UsrRdd_RddData()	Queries or attaches user-defined data a user-defined RDD.
UsrRdd_SetBof()	Sets the BoF() flag in a work area of a user-defined RDD.
UsrRdd_SetBottom()	Defines the bottom scope value for a work area of a user-defined RDD.
UsrRdd_SetEof()	Sets the Eof() flag in a work area of a user-defined RDD.
UsrRdd_SetFound()	Sets the Found() flag in a work area of a user-defined RDD.
UsrRdd_SetTop()	Defines the top value for a work area of a user-defined RDD.
ValToPrg()	Converts a value to PRG code.
ValToPrgExp()	Converts a value to a character string holding a macro-expression.
VAR	Declares an instance variable of a class.
WaitForThreads()	Suspends the current thread until all other threads have terminated.
Wild2RegEx()	Converts a character string including wild card characters to a regular expression.
WildMatch()	Tests if a string begins with a search pattern.
Win32Bmp()	Creates a new Win32Bmp object.
Win32Prn()	Creates a new Win32Prn object.
WInfo()	Returns all coordinates of the current window.
WITH OBJECT	Identifies an object to receive multiple messages.
WMSetPos()	Moves the mouse cursor to a new position in a window.
WSetMouse()	Determines the visibility and/or position of the mouse cursor.
WStack()	Returns window IDs of all open windows.
[ ] (string)	Character operator (unary): retrieves a character from a string.
^^	Bitwise XOR operator (binary): performs a logical XOR operation.
{=>}	Literal hash.
{^ }	Literal DateTime value.
| (bitwise OR)	Bitwise OR operator (binary): performs a logical OR operation.

ZIP compression

HB_Compress()	Compresses a character string (ZIP).
HB_CompressBufLen()	Calculates the buffer size required for compression.
HB_CompressError()	Returns the error code of the last (un)compression.
HB_CompressErrorDesc()	Returns an error description from a numeric error code.
HB_Uncompress()	Uncompresses a compressed character string (ZIP).

Index

- (operator) ... 2061
-- (operator) ... 2063
- <cTagName>()
  THtmlNode() ... 181
!= (comparison) ... 2078
(comparison) ... 2078
#command ... 2118
#command | #translate ... 2118
#define ... 2125
#error ... 2128
#if ... 2129
#ifdef ... 2131
#ifndef ... 2133
#include ... 2134
#pragma ... 2136
#stdout ... 2138
#translate ... 2118
#uncommand | #untranslate ... 2139
#undef ... 2140
#xcommand | #xtranslate ... 2141
#xuncommand | #xuntranslate ... 2142
$ (operator) ... 2045
% (operator) ... 2046
%= (operator) ... 2085
& (bitwise AND) ... 2047
& (macro operator) ... 2049
& (operator) ... 2047, 2049
( ) (operator) ... 2054
(struct) ... 2144
* (operator) ... 2056
** (operator) ... 2057
**= (operator) ... 2085
*= (operator) ... 2085
... (parameter list) ... 2196
.AND. ... 2066
.NOT. ... 2067
.OR. ... 2068
/ (operator) ... 2069
: (operator) ... 2070
:<tagName> --> oTHtmlNode()
  THtmlNode() ... 182
:<tagName>S[()
  THtmlNode() ... 183
:= (assignment) ... 2072
:aAttributes
  TXmlNode() ... 265
:aCMembers
  C Structure class ... 5
:aCTypes
  C Structure class ... 5
:addBelow()
  TXmlNode() ... 268
:addColumn()
  TBrowse() ... 126
:addGetForm()
  TUrl() ... 242
:addItem()
  HbListBox() ... 51
  HbRadioGroup() ... 87
  Popup() ... 106
  TopBarMenu() ... 143
:addNode()
  THtmlNode() ... 176
:applyKey()
  TBrowse() ... 136
:arc()
  Win32Prn() ... 302
:args
  Error() ... 12
:array()
  C Structure class ... 6
:assign()
  Get() ... 28
:attach()
  TIpMail() ... 220
:attachFile()
  TIpMail() ... 217
:attr
  THtmlNode() ... 167
:attrToString()
  THtmlNode() ... 173
:auth()
  TIpClientSmtp() ... 212
:authPlain()
  TIpClientSmtp() ... 212
:autoLite
  TBrowse() ... 119
:backspace()
  Get() ... 33
:badDate
  Get() ... 20
:barLength
  HbScrollBar() ... 94
:binNumber
  Win32Prn() ... 287
:bitmap
  Win32Bmp() ... 276
:bitmapsOk
  Win32Prn() ... 280
:bkColor
  Win32Prn() ... 280
:block
  Get() ... 20
  TBColumn() ... 113
:body
  THtmlDocument() ... 152
:bold()
  Win32Prn() ... 296
:border
  Popup() ... 104
  TBrowse() ... 119
:bottom

HbListBox() 44
HbRadioGroup() 82
Popup() 104
:bottomMargin
Win32Prn() 285
:box()
Win32Prn() 303
:buffer
Get() 21
HbCheckBox() 37
HbListBox() 45
HbPushButton() 69
HbRadioButton() 76
HbRadioGroup() 83
:buffer()
C Structure class 6
:buildAddress()
TUrl() 242
:buildQuery()
TUrl() 242
:cAddress
TUrl() 239
:canDefault
Error() 12
:canRetry
Error() 13
:canSubstitute
Error() 13
:capCol
Get() 21
HbCheckBox() 38
HbListBox() 45
HbRadioButton() 76
HbRadioGroup() 83
:capRow
Get() 21
HbCheckBox() 38
HbListBox() 45
HbRadioButton() 77
HbRadioGroup() 83
:caption
Get() 21
HbCheckBox() 38
HbListBox() 45
HbPushButton() 70
HbRadioButton() 76
HbRadioGroup() 83
MenuItem() 100
:cargo
Error() 13
Get() 22
HbCheckBox() 38
HbListBox() 46
HbPushButton() 70
HbRadioButton() 77
HbRadioGroup() 84
HbScrollBar() 94
MenuItem() 100
Popup() 104
TBColumn() 114
TBrowse() 119
TopBarMenu() 141
:cData
TXmlNode() 265
:cFile
TStreamFileReader() 230
TStreamFileWriter() 235
TUrl() 240
:changed
Get() 22
THtmlDocument() 153
:charHeight
Win32Prn() 283
:charSet()
Win32Prn() 297
:charWidth
Win32Prn() 283
:checked
MenuItem() 100
:childNodes
THtmlNode() 167
:className()
HBObject() 62
:classSel()
HBObject() 62
:clear
Get() 22
:clone()
THtmlIterator() 159
TXmlIterator() 255
TXmlNode() 269
:cloneTree()
TXmlNode() 269
:close()
HbListBox() 51
Popup() 109
TIpClient() 188
TStreamFileReader() 230
TStreamFileWriter() 235
:cName
TXmlNode() 265
:col
Get() 22
HbCheckBox() 39
HbPushButton() 70
HbRadioButton() 77
:colCount
TBrowse() 119
:coldBox
HbListBox() 46
HbRadioGroup() 84
:collect()
THtmlDocument() 154
THtmlNode() 177
:colorBlock
TBColumn() 114
:colorDisp()
Get() 28

:colorRect()
TBrowse() ... 128
:colorSpec
Get() ... 23
HbCheckBox() ... 39
HbListBox() ... 46
HbPushButton() ... 70
HbRadioButton() ... 77
HbRadioGroup() ... 84
HbScrollBar() ... 94
Popup() ... 104
TBrowse() ... 120
TopBarMenu() ... 142
:colPos
TBrowse() ... 120
:colSep
TBColumn() ... 114
TBrowse() ... 120
:colWidth()
TBrowse() ... 126
:commit()
TIpClient() ... 188
:configure()
TBrowse() ... 129
:control
Get() ... 23
:copies
Win32Prn() ... 288
:copyTo()
TStream() ... 228
:countAttachments()
TIpMail() ... 220
:cPassword
TUrl() ... 240
:cPath
TUrl() ... 240
:cProto
TUrl() ... 240
:cQuery
TUrl() ... 241
:create()
Win32Prn() ... 312
:cReply
TIpClient() ... 186
:cServer
TUrl() ... 241
:current
HbScrollBar() ... 95
Popup() ... 105
TopBarMenu() ... 142
:cUserId
TUrl() ... 241
:cwd()
TIpClientFtp() ... 194
:data
MenuItem() ... 101
:data()
TIpClientSmtp() ... 213
:decPos
Get() ... 24
:defColor
TBColumn() ... 114
:deHilite()
TBrowse() ... 129
:delAttribute()
THtmlNode() ... 174
:delAttributes()
THtmlNode() ... 174
:delColumn()
TBrowse() ... 126
:dele()
TIpClientFtp() ... 196
:delEnd()
Get() ... 34
:delete()
Get() ... 34
THtmlNode() ... 177
TIpClientPop() ... 206
:delItem()
HbListBox() ... 52
HbRadioGroup() ... 87
Popup() ... 107
TopBarMenu() ... 144
:delLeft()
Get() ... 34
:delRight()
Get() ... 34
:delWordLeft()
Get() ... 35
:delWordRight()
Get() ... 35
:depth()
TXmlNode() ... 272
:description
Error() ... 13
:destroy()
Win32Prn() ... 312
:detachFile()
TIpMail() ... 217
:deValue()
C Structure class ... 7
:display()
Get() ... 29
HbCheckBox() ... 41
HbListBox() ... 52
HbPushButton() ... 73
HbRadioButton() ... 79
HbRadioGroup() ... 88
HbScrollBar() ... 97
Popup() ... 110
TopBarMenu() ... 147
:document
THtmlNode() ... 168
:down()
TBrowse() ... 132
:downloadFile()
TIpClientFtp() ... 198
:draw()

Win32Bmp() 277
:drawBitmap()
Win32Prn() 303
:dropDown
HbListBox() 47
:ellipse()
Win32Prn() 304
:enabled
MenuItem() 101
:end
HbScrollBar() 95
:end()
Get() 32
TBrowse() 132
:endDoc()
Win32Prn() 293
:endPage()
Win32Prn() 293
:error()
HBObject() 63
:exGauge
TIpClient() 186
:exitState
Get() 24
:fBlock
HbCheckBox() 39
HbListBox() 47
HbPushButton() 71
HbRadioButton() 78
HbRadioGroup() 85
:fileName
Error() 14
Win32Bmp() 277
:fillRect()
Win32Prn() 304
:finalize()
TStreamFileReader() 230
TStreamFileWriter() 235
:find()
THtmlIteratorRegEx() 162
THtmlIteratorScan() 164
TXmlIteratorRegEx() 258
TXmlIteratorScan() 261
:findFirst()
THtmlDocument() 155
TXmlDocument() 249
:findFirstRegex()
THtmlDocument() 155
:findFirstRegEx()
TXmlDocument() 250
:findNext()
THtmlDocument() 156
TXmlDocument() 251
:findText()
HbListBox() 52
:firstNode()
THtmlNode() 177
:fontName
Win32Prn() 283
:fontPointSize
Win32Prn() 284
:fontWidth
Win32Prn() 284
:footing
TBColumn() 115
:footSep
TBColumn() 115
TBrowse() 120
:forceStable()
TBrowse() 129
:formType
Win32Prn() 288
:freeze
TBrowse() 121
:fromString()
TIpMail() 221
:genCode
Error() 14
:getAccel()
HbRadioGroup() 88
Popup() 110
TopBarMenu() 147
:getAttachment()
TIpMail() 221
:getAttribute()
THtmlNode() 174
TXmlNode() 267
:getAttributes()
THtmlNode() 175
:getBody()
TIpMail() 218
:getCharEncoding()
TIpMail() 221
:getCharHeight()
Win32Prn() 295
:getCharWidth()
Win32Prn() 295
:getColumn()
TBrowse() 127
:getContentType()
TIpMail() 222
:getData()
HbListBox() 53
:getDeviceCaps()
Win32Prn() 301
:getFieldOption()
TIpMail() 222
:getFieldPart()
TIpMail() 222
:getFileName()
TIpMail() 218
:getFirst()
Popup() 107
TopBarMenu() 144
:getFonts()
Win32Prn() 297
:getItem()
HbListBox() 53

HbRadioGroup() 89
Popup() 107
TopBarMenu() 144
:getLast()
Popup() 108
TopBarMenu() 145
:getMultiParts()
TIpMail() 218
:getNext()
Popup() 108
TopBarMenu() 145
:getNode()
THtmlDocument() 156
THtmlIterator() 159
TXmlIterator() 255
:getNodes()
THtmlDocument() 157
:getPointer()
C Structure class 7
:getPrev()
Popup() 108
TopBarMenu() 145
:getShortct()
Popup() 110
:getText()
HbListBox() 54
THtmlNode() 180
:getTextHeight()
Win32Prn() 295
:getTextWidth()
Win32Prn() 296
:goBottom()
TBrowse() 132
:goBottomBlock
TBrowse() 121
:goTop()
TBrowse() 133
:goTopBlock
TBrowse() 121
:handle
TStreamFileReader() 230
TStreamFileWriter() 235
:hasFocus
Get() 24
HbCheckBox() 40
HbListBox() 47
HbPushButton() 71
HbRadioButton() 78
HbRadioGroup() 85
:havePrinted
Win32Prn() 289
:head
THtmlDocument() 153
:heading
TBColumn() 115
:headSep
TBColumn() 115
TBrowse() 121
:hHeaders
TIpMail() 217
:hilite()
TBrowse() 130
:hitBottom
TBrowse() 122
:hitTest()
Get() 29
HbCheckBox() 42
HbListBox() 54
HbPushButton() 73
HbRadioButton() 80
HbRadioGroup() 89
HbScrollBar() 98
Popup() 111
TBrowse() 137
TopBarMenu() 147
:hitTop
TBrowse() 122
:home()
Get() 32
TBrowse() 133
:hotBox
HbListBox() 48
HbRadioGroup() 85
:hPrinterDc
Win32Prn() 289
:htmlEndTagName
THtmlNode() 169
:htmlTagName
THtmlNode() 169
:htmlTagType
THtmlNode() 170
:id
MenuItem() 101
:inch_To_PosX()
Win32Prn() 290
:inch_To_PosY()
Win32Prn() 290
:init()
HBObject() 61
:initClass()
HBObject() 61
:insColumn()
TBrowse() 127
:insert()
Get() 35
:insertAfter()
THtmlNode() 178
TXmlNode() 269
:insertBefore()
THtmlNode() 178
TXmlNode() 269
:insertBelow()
THtmlNode() 178
TXmlNode() 270
:insItem()
HbListBox() 55
HbRadioGroup() 90
Popup() 108

TopBarMenu() 146
:invalidate()
TBrowse() 130
:isAccel()
HbRadioButton() 80
:isAttribute()
THtmlNode() 175
:isBlock
THtmlNode() 170
:isDerivedFrom()
HBObject() 62
:isEmpty
THtmlNode() 171
:isInline
THtmlNode() 171
:isKindOf()
HBObject() 63
:isMultipart()
TIpMail() 219
:isNode
THtmlNode() 172
:isOpen
HbListBox() 48
:isOpen()
Popup() 111
:isOptional
THtmlNode() 172
:isPopUp()
MenuItem() 102
:isType()
THtmlNode() 179
:italic()
Win32Prn() 298
:itemCount
HbListBox() 48
HbRadioGroup() 86
Popup() 105
TopBarMenu() 142
:killFocus()
Get() 30
HbCheckBox() 42
HbListBox() 55
HbPushButton() 74
HbRadioButton() 81
HbRadioGroup() 90
:landscape
Win32Prn() 288
:lastErrorCode()
TIpClient() 190
:lastErrorMessage()
TIpClient() 190
:lastNode()
THtmlNode() 179
:lCanRead
TStream() 227
:lCanWrite
TStream() 227
:left
HbListBox() 49
HbRadioGroup() 86
Popup() 105
TopBarMenu() 143
:left()
Get() 32
TBrowse() 133
:leftMargin
Win32Prn() 286
:leftVisible
TBrowse() 122
:line()
Win32Prn() 305
:lineHeight
Win32Prn() 284
:list()
TIpClientFtp() 196
TIpClientPop() 207
:listFiles()
TIpClientFtp() 196
:loadFile()
Win32Bmp() 277
:loadFromFile()
HBPersistent() 67
:loadFromText()
HBPersistent() 67
:lTrace
TIpClient() 187
:mail()
TIpClientSmtp() 213
:maxCol()
Win32Prn() 291
:maxRow()
Win32Prn() 291
:mColPos
TBrowse() 122
:message
Get() 25
HbCheckBox() 40
HbListBox() 49
HbPushButton() 71
HbRadioGroup() 86
MenuItem() 101
TBrowse() 123
:mget()
TIpClientFtp() 198
:minus
Get() 25
:mkd()
TIpClientFtp() 195
:mm_To_PosX()
Win32Prn() 290
:mm_To_PosY()
Win32Prn() 291
:modal()
TopBarMenu() 148
:moduleName
Error() 15
:mput()
TIpClientFtp() 199

:mRowPos
TBrowse() ... 123
:msgNotFound()
HBObject() ... 64
:nAlign
C Structure class ... 6
:name
Get() ... 25
:nBeginLine
TXmlNode() ... 266
:nBottom
TBrowse() ... 123
:nConnTimeout
TIpClient() ... 187
TIpClientFtp() ... 194
TIpClientHttp() ... 202
TIpClientPop() ... 206
TIpClientSmtp() ... 211
:nDefaultPort
TIpClient() ... 187
TIpClientFtp() ... 194
TIpClientHttp() ... 203
TIpClientPop() ... 206
TIpClientSmtp() ... 211
:nError
TXmlDocument() ... 246
:new()
HBObject() ... 60
:newLine()
Win32Prn() ... 294
:newPage()
Win32Prn() ... 294
:next()
THtmlIterator() ... 160
THtmlIteratorRegEx() ... 163
THtmlIteratorScan() ... 165
TXmlIterator() ... 256
TXmlIteratorRegEx() ... 259
TXmlIteratorScan() ... 262
:nextAttachment()
TIpMail() ... 223
:nextInTree()
TXmlNode() ... 270
:nextItem()
HbListBox() ... 56
HbRadioGroup() ... 90
:nextNode
THtmlNode() ... 168
:nLastRead
TIpClient() ... 187
:nLastWrite
TIpClient() ... 188
:nLeft
TBrowse() ... 123
:nLength
TStream() ... 228
:nLine
TXmlDocument() ... 247
:nNodeCount
TXmlDocument() ... 247
:noOp()
TIpClientPop() ... 207
:nPort
TUrl() ... 241
:nPosition
TStream() ... 228
:nRight
TBrowse() ... 124
:nStatus
TXmlDocument() ... 246
:nTop
TBrowse() ... 124
:nType
TXmlNode() ... 266
:numColors
Win32Prn() ... 281
:oChild
TXmlNode() ... 266
:oErrorNode
TXmlDocument() ... 247
:offset
HbScrollBar() ... 95
:oNext
TXmlNode() ... 267
:oParent
TXmlNode() ... 267
:open()
HbListBox() ... 56
Popup() ... 112
TIpClient() ... 189
:operation
Error() ... 15
:oPrev
TXmlNode() ... 267
:orient
HbScrollBar() ... 95
:original
Get() ... 25
:oRoot
TXmlDocument() ... 246
:osCode
Error() ... 16
:oUrl
TIpClient() ... 188
:overStrike()
Get() ... 35
:pageDown()
TBrowse() ... 133
:pageHeight
Win32Prn() ... 286
:pageUp()
TBrowse() ... 134
:pageWidth
Win32Prn() ... 286
:panEnd()
TBrowse() ... 134
:panHome()
TBrowse() ... 134

:panLeft()
TBrowse() 134
:panRight()
TBrowse() 135
:parentNode
THtmlNode() 168
:path()
TXmlNode() 272
:pCol()
Win32Prn() 292
:penColor
Win32Prn() 281
:penStyle
Win32Prn() 281
:penWidth
Win32Prn() 281
:picture
Get() 26
TBColumn() 116
:pixelsPerInchX
Win32Prn() 285
:pixelsPerInchY
Win32Prn() 285
:pointer()
C Structure class 7
:pos
Get() 26
:postBlock
Get() 26
TBColumn() 116
:posX
Win32Prn() 282
:posY
Win32Prn() 282
:preBlock
Get() 26
TBColumn() 116
:prevItem()
HbListBox() 56
HbRadioGroup() 91
:prevNode
THtmlNode() 168
:printerName
Win32Prn() 289
:printing
Win32Prn() 289
:procLine
Error() 16
:procName
Error() 16
:pRow()
Win32Prn() 292
:pwd()
TIpClientFtp() 195
:quit()
TIpClientSmtp() 213
:rcpt()
TIpClientSmtp() 214
:read()
TIpClient() 189
TStreamFileReader() 231
TXmlDocument() 247
:readAll()
TIpClientHttp() 203
:readByte()
TStreamFileReader() 231
:reader
Get() 27
:readFile()
THtmlDocument() 153
:readToFile()
TIpClient() 191
:rect
Win32Bmp() 276
:refreshAll()
TBrowse() 130
:refreshCurrent()
TBrowse() 130
:rejected
Get() 27
:rename()
TIpClientFtp() 199
:reset()
C Structure class 8
Get() 30
TIpClient() 189
:resetAttachment()
TIpMail() 223
:retrieve()
TIpClientPop() 207
:rewind()
THtmlIterator() 160
TXmlIterator() 256
:right
HbListBox() 49
HbRadioGroup() 86
Popup() 106
TopBarMenu() 143
:right()
Get() 32
TBrowse() 135
:rightMargin
Win32Prn() 286
:rightVisible
TBrowse() 124
:rmd()
TIpClientFtp() 195
:root
THtmlDocument() 153
:row
Get() 27
HbCheckBox() 40
HbPushButton() 71
HbRadioButton() 78
TopBarMenu() 143
:rowCount
TBrowse() 124
:rowPos

TBrowse() 125
:saveToFile()
HBPersistent() 66
:saveToText()
HBPersistent() 66
:sBlock
HbCheckBox() 40
HbListBox() 49
HbPushButton() 72
HbRadioButton() 79
HbScrollBar() 96
:scroll()
HbListBox() 56
:seek()
TStreamFileReader() 232
TStreamFileWriter() 236
:select()
HbCheckBox() 42
HbListBox() 57
HbPushButton() 74
HbRadioButton() 81
HbRadioGroup() 91
Popup() 112
TopBarMenu() 149
:sendMail()
TIpClientSmtp() 211
:setAddress()
TUrl() 243
:setAttribute()
THtmlNode() 175
TXmlNode() 268
:setAttributes()
THtmlNode() 176
:setBkMode()
Win32Prn() 305
:setBody()
TIpMail() 219
:setColor()
HbRadioGroup() 91
Win32Prn() 306
:setColumn()
TBrowse() 128
:setContext()
THtmlIterator() 160
TXmlIterator() 256
:setData()
HbListBox() 57
:setDefaultFont()
Win32Prn() 299
:setDuplexType()
Win32Prn() 301
:setFieldOption()
TIpMail() 223
:setFieldPart()
TIpMail() 224
:setFocus()
Get() 30
HbCheckBox() 43
HbListBox() 58
HbPushButton() 74
HbRadioButton() 81
HbRadioGroup() 92
:setFont()
Win32Prn() 299
:setFontOk
Win32Prn() 285
:setHeader()
TIpMail() 219
:setItem()
HbListBox() 58
Popup() 109
TopBarMenu() 146
:setKey()
TBrowse() 137
:setPen()
Win32Prn() 307
:setPos()
Win32Prn() 307
:setPrc()
Win32Prn() 292
:setPrintQuality()
Win32Prn() 302
:setStyle()
HbRadioGroup() 92
TBColumn() 117
TBrowse() 131
:setText()
HbListBox() 59
:severity
Error() 16
:shortCut
MenuItem() 102
:siblingNodes
THtmlNode() 169
:sizeOf()
C Structure class 8
:skipBlock
TBrowse() 125
:stabilize()
TBrowse() 131
:stable
TBrowse() 125
:start
HbScrollBar() 96
:startDoc()
Win32Prn() 294
:startPage()
Win32Prn() 294
:stat()
TIpClientPop() 208
:style
HbCheckBox() 41
HbListBox() 50
HbPushButton() 72
HbRadioButton() 79
HbScrollBar() 96
MenuItem() 102
:subCode

Error() 17
:subscript
Get() 27
:subSystem
Error() 17
:text
THtmlNode() 173
:textAlign
Win32Prn() 282
:textAtFont()
Win32Prn() 308
:textColor
Win32Prn() 282
:textOut()
Win32Prn() 310
:textOutAt()
Win32Prn() 311
:thumbPos
HbScrollBar() 97
:toArray()
TXmlNode() 272
:toDecPos()
Get() 33
:top
HbListBox() 50
HbRadioGroup() 87
Popup() 106
:top()
TIpClientPop() 208
:topItem
HbListBox() 50
:topMargin
Win32Prn() 287
:toString()
THtmlDocument() 154
THtmlNode() 180
TIpMail() 224
TXmlDocument() 248
TXmlNode() 271
:total
HbScrollBar() 97
:tries
Error() 17
:type
Get() 28
:typeOut
Get() 28
HbCheckBox() 41
HbListBox() 50
HbPushButton() 72
HbRadioGroup() 87
:uidl()
TIpClientPop() 208
:underLine()
Win32Prn() 298
:undo()
Get() 30
:unlink()
THtmlNode() 179
TXmlNode() 270
:unTransform()
Get() 31
:up()
TBrowse() 135
:update()
HbScrollBar() 97
:updateBuffer()
Get() 31
:uploadFile()
TIpClientFtp() 200
:value()
C Structure class 8
:varGet()
Get() 31
:varPut()
Get() 31
:vScroll
HbListBox() 51
:width
Popup() 106
TBColumn() 116
:wordLeft()
Get() 33
:wordRight()
Get() 33
:write()
TIpClient() 190
TStreamFileWriter() 236
TXmlDocument() 249
TXmlNode() 271
:writeByte()
TStreamFileWriter() 237
:writeFile()
THtmlDocument() 154
:writeFromFile()
TIpClient() 191
? | ?? 316
@ (operator) 2094
@() (operator) 2095
@...BOX 318
@...CLEAR 320
@...GET .... 322, *see also:* Get object
@...GET CHECKBOX 326
@...GET LISTBOX 329
@...GET PUSHBUTTON 332
@...GET RADIOGROUP 336
@...GET TBROWSE 339
@...PROMPT 343
@...SAY 344
defining output device 461
@...TO 347
[ ] (array) 2103
[ ] (string) 2104
^^2106
^= 2085
{ } (array) 2108
{^ } 2111
{|| } (code block) 2113

{=>} (hash) 2109
| 2115
| (bitwise OR) 2115
+ (operator) 2058
+ <cTagName>()
  THtmlNode() 180
++ (operator) 2060
+= (operator) 2085
< (comparison) 2073
<|| > (extended code block) 2080
<< (operator) 2075
<= (comparison) 2076
<> != # (comparison) 2078
<> (comparison) 2078
= (assignment) 2082, 2083, 2085
= (comparison) 2083
= (compound assignment) 2085
-= (operator) 2085
== (operator) 2087
> (comparison) 2089
-> (operator) 2064
>= (comparison) 2091
>> (operator) 2093
AAdd() 533
Abs() 534
Abstract class 60
ACCEPT 349
ACCESS 2145
AChoice() 535
AClone() 540
ACopy() 542
ACos() 544
AddASCII() 545
Addition 2058
AddMonth() 546
ADel() 547
ADir() 549
Adler32 1120
AEval() 551
AFields() 553
AFill() 555
AfterAtNum() 557
AIns() 558
ALenAlloc() 560
Alert() 561
Alias name
  of a work area 563
Alias operator 2064
Alias() 563
AllTrim() 564
Alt+C 1840
AltD() 565
AmPm() 567
AND operator 2066
ANNOUNCE 2147
ANSI
  conversion of character strings 710, 1077
  converting to HTML 568
AnsiToHtml() 568
Antilogarithm 930
APPEND BLANK 350
APPEND FROM 352
Application
  setting the exit code 922
Arc cosine 544
Arc sine 577
Arc tangent 587
Argument
  getting all 1079
  getting the value 1744
  number of passed 1713
Array 2108
  add one element 533
  changing the number of elements 578
  contents of last element 586
  copying elements 542
  creating 571
  creating set/get code block 1085
  deep copy 540
  deleting elements 547
  executable 1152
  executing a code block for each array element 551
  filling with a value 555
  filling with database structure 837
  filling with directory information 549
  filling with field information 553
  from pointer 1307
  inserting an element 558
  number of elements 1507
  pre-allocated memory 560
  pre-allocating memory 579
  searching from bottom 1753
  searching from top 573
  sorting 580
  unique identifier 1086
Array element
  operator 2103
Array() 571
Asc() 572
AScan() 573
ASCIISum() 575
AscPos() 576
ASin() 577
ASize() 578
ASizeAlloc() 579
ASort() 580
ASSIGN 2148
Assignment
  compound 2085
  inline 2072
  simple 2082
  to a field variable 427
ASSOCIATE CLASS 2151
Associative array 1357
  change value by ordinal position 1067
  deleting an element 1057
  ordinal key position 1061
  retrieve key by ordinal position 1059

retrieve value by ordinal position....................... 1065
sort order of all keys.. 1346
sort order of keys.. 1063
At()... 582
AtAdjust() 584
ATail() 586
ATan()... 587
ATn2() 588
AtNum()... 589
AtRepl() 591
AtSkipStrings() 593
AtToken()... 595
Automation
creating OLE object 718
retrieves existing OLE object............................ 1017
AVERAGE 356
Background tasks ... 1094, *see also:* Idle tasks, *see also:* Idle tasks
(de)activating 1093
activating.. 443
changing processing interval............................... 444
deleting.. 1096
forced execution.. 1098
resetting... 1097
time interval to wait 1099
Base 64
decoding a character string................................ 1100
decoding a file.. 1101
encoding a character string................................ 1102
encoding a file.. 1103
BeforAtNum().. 596
Begin of file 623
BEGIN SEQUENCE.. 2155
Bin2I()... 597
Bin2L().. 598
Bin2U() 599
Bin2W() 600
Binary large object
changing data 610
exporting to file... 603
importing from file.. 608
load into memory 606
Binary large object file
deleting root area... 617
locking the root area.. 619
reading the root area.. 618
storing data in the root area................................. 620
unlocking the root area.. 622
Binary string
deserialize 1142
serialize 1283
Bit functions
AND.. 1104
bitwise NOT...1618, 1619
bitwise XOR...1627, 1628
clearing on or more bits....................................... 691
converting bits to string....................................... 601
converting string to bits....................................... 737
left rotation...1622, 1624
mirrors bits...1616, 1617
NOT..1107
OR...1108
set bit to 0..1109
set bit to 1..1111
setting on or more bits 1838
shifting bits 1113
test if bit is set 1105
testing single bit 1461
XOR...1115
Bitmap object..276
BitToC()..601
Bitwise AND 2047
with list of numbers 1607, 1608
Bitwise OR..2115
with list of numbers 1620, 1621
Bitwise XOR...2106
Blank spaces
removing from character strings 564
removing leading 1519
removing trailing 1799, 1955
Blank() 602
Blink attribute
for colors..1839
BlobDirectExport() 603
BlobDirectGet() 606
BlobDirectImport() 608
BlobDirectPut()..610
BlobExport() 612
BlobGet() 614
BlobImport() 615
BlobRootDelete()...617
BlobRootGet() 618
BlobRootLock() 619
BlobRootPut()..620
BlobRootUnlock() 622
BoF()...623
BoM() 625
BoQ() 626
BoY() 627
BREAK...2155
Break() 628
Browse() ....630, *see also:* TBrowse object
Browser
for text mode (simple)..630
in text mode (with user function)..........................772
C structure
converting to array 1302
C Structure
creating from array...1087
size of...1299
C Structure class 4
Calendar day 750
Calendar week 2002
in a month 2024
Calendar year .... *see:* year
CallDll()...632
CANCEL 358
Caps lock 1496

Carriage return
- replacing in a character string 1544
- replacing soft with hard 1070

CATCH 2242
CDoW() 634
Ceiling() 635
Celsius() 636
Center() 637
CFTSAdd() 639
CFTSClose() 640
CFTSCrea() 641
CFTSDelete() 643
CFTSIfDel() 644
CFTSNext() 645
CFTSOpen() 646
CFTSRecn() 647
CFTSReplac() 648
CFTSSet() 649
CFTSUndel() 650
CFTSVeri() 651
CFTSVers() 652
Character
- and ASCII Code 572
- first position in a character string 582, 593
- last position in a character string 1755
- wild match pattern 2014

Character operator 2104
Character string
- adding numeric ASCII code 545
- adding numeric ASCII codes 653
- ASCII code of character 576
- binary AND 654
- binary NOT 663
- binary OR 667
- binary XOR 687
- bitwise left rotation 674
- bitwise left shift 676
- bitwise right rotation 675
- bitwise right shift 677
- check if it begins with a 7-bit ASCII character 1460
- check if it begins with a control character 1462
- check if it begins with a digit 1465
- check if it begins with a grahpical 7-bit ASCII character 1470
- check if it begins with a hex-digit character 1485
- check if it begins with a letter 1459
- check if it begins with a lowercase letter 1473
- check if it begins with a printable 7-bit ASCII character 1475
- check if it begins with a punctuation character 1478
- check if it begins with a white-space character 1481
- check if it begins with an alpha-numeric character 1458
- check if it begins with an uppercase letter 1482
- checksum 688
- comparison with RegEx 2101
- compressing 668, 1129
- consisting of blank spaces 1880
- converting to a numeric value 1980
- converting to ANSI 710, 1077
- converting to date value 738
- converting to DateTime value 743
- converting to lower case 1516
- converting to OEM 711, 1235
- converting to upper case 1969
- count char from left 716
- count char from right 717
- counting substrings 1609
- countng substring 1629
- creating literal 1896
- decrypting 1913
- deleting character(s) 1720
- deleting characters 672
- deleting characters on both sides 1778
- deleting characters on the left 1779
- deleting characters on the right 1780
- deleting of characters 1891, 1906
- deleting range 1751
- deleting words 2031
- encrypting 725, 1915
- exchanging adjacent chars 683
- exchanging adjacent words 2033
- exchanging characters 1900
- expanding 931
- expanding Tab chars 1919
- extracting even characters 655
- extracting odd characters 664
- extracting substring 557, 596
- extracting substring from the left 1506
- extracting substring from the right 1792
- extracting substrings 1911
- histogram 656
- inserting character(s) 1723
- inserting of characters 1906
- inserting Tab chars 1920
- justifying as block 681
- justifying centered 637
- justifying left 1488
- justifying right 1489
- justifying substring 584
- merging two strings 660
- missing characters 662
- multi-pass mode 726
- multiple search 589
- multiple search and replace 591
- number of characters 1507
- number of tokens 1626
- numeric value of digit 1982
- obtaining pointer 1301
- padding left 1708
- padding right 1709
- padding with a fill character 1706
- pass by reference mode 731
- phonetic similarity 1879
- position of first alpha char 1716
- position of first lower char 1724
- position of first upper char 1728
- related search 669

related search and replace 670
removing all chars but 666
removing all words but 2030
removing blank spaces 564
removing duplicate adjacent characters 665
removing duplicate adjacent words 2029
removing duplicate characters 658
removing leading blank spaces 1519
removing trailing blank spaces 1799, 1955
removing/sorting duplicate characters 678
replacing character(s) 1727
replacing characters 1902
replacing characters on both sides 1782
replacing characters on the left 1784
replacing characters on the right 1785
replacing range 1752
replacing single char 1718
replacing words 2032
replicating 1783
reverse order 659
search and replace 673
search mode 1836
search substring with RegEx 2097
searching a range of chars 1726
searching different chars 1721
searching equal chars 1722
searching token 595
similarity 1892
sorting 679
sorting weight 1287
starting position of a line 1558
subtracting numeric ASCII codes 682
sum of ASCII codes 575
tokenize 1089
uncompressing 684, 1309
words to chars 2034

Character string comparison
activating exact comparison 472

Character value
converting to Hex string 1901

CharAdd() 653
CharAND() 654
CharEven() 655
CharHist() 656
CharList() 658
CharMirr() 659
CharMix() 660
CharNoList() 662
CharNOT() 663
CharOdd() 664
CharOne() 665
CharOnly() 666
CharOR() 667
CharPack() 668
CharRela() 669
CharRelRep() 670
CharRem() 672
CharRepl() 673
CharRLL() 674
CharRLR() 675
CharSHL() 676
CharSHR() 677
CharSList() 678
CharSort() 679
CharSpread() 681
CharSub() 682
CharSwap() 683
CharUnpack() 684
CharWin() 685
CharXOR() 687
Check box (text mode) 37, 326

Checksum
using Adler32 algorithm 1120
using CRC32 algorithm 1134
using MD5 algorithm 1223

CheckSum() 688
Child work area 817
Chr() 689

Class
adding member variables 2184
adding methods 2186

CLASS 2157
CLASSDATA 2162
CLASSMETHOD 2165
CLEAR ALL 359
CLEAR GETS 360
CLEAR MEMORY 361
CLEAR SCREEN 362
CLEAR TYPEAHEAD 363
ClearBit() 691
ClearEol() 692
ClearSlow() 694
ClearWin() 696
ClEol() 698
CLOSE 364
CLOSE LOG 366

Closing
all or a group of files 364

CLS 362
ClWin() 699
CMonth() 700
Code block 2113
associating with a key 1854
associating with multiple keys 1290
creating for field access 943, 956
executing 927
extended 2080
for dynamic memory variables 1546
for error handling 919
restoring from binary 1281

Code page 1287
converting character strings 1308

Col() 702
Collection 2193
index of item 1148

Color
attribute as character 1605
Blink attribute in text mode 1839

extracting background 1022
extracting by ordinal position 1024, 1125
extracting foreground 1023
getting default attribute 1020
in text mode 1845
inverting attribute 1454
length of color value 1036
numeric attribute 706, 1127, 1812
position of color value 1037
replacing 1844
replacing attributes 703, 1123
replacing attributes in a region 707
selecting in text mode 704
setting default attribute 1842
setting in text mode 447

Color index for text mode 704
ColorRepl() 703
ColorSelect() 704
ColorToN() 706
ColorWin() 707
command *see:* #command
Command line
EXE file name 929, 1124
number of arguments 1080
parameter list 901
test internal switches 1081
value of arguments 1083
value of internal switch 1082

Command shell 431
COMMIT 367
Comparison
equal 2083
for identity 2087
greater than 2089
greater than or equal 2091
less than 2073
less than or equal 2076
not equal 2078

Compiler
build date 1117
build information 1118
C compiler used for build 1128
conditional compilation 2129, 2131, 2133
displaying message 2138

Compiling
conditional 2129, 2131, 2133

Complement() 709
Concatenation 2058, 2061
CONTINUE 368
Control structure
branch 2173, 2202, 2240
for error handling 2155, 2242
iteration 2191, 2193
iteration index 1148
loop 2175
terminate a function 2237

Controlling index 493, 1676
automatic selecting 441
position in index list 1402

Converting
16 bit integer (signed) to numeric value 597
16 bit integer (unsigned) to numeric 600
32 bit integer (signed) to numeric value 598
32 bit integer (unsigned) to numeric 599
8 byte binary to float number 740
any value to macro-expression 1985
any value to PRG code 1983
any value to string 734
any value to text 1147
ASCII Code to character 689
Celsius to Fahrenheit 934
character in date value 738
character in DateTime value 743
character string to a numeric value 1980
character string to DateTime value 1888
character string YYYYMMDD to date value 1886
character to ASCII Code 572
character value to Hex string 1901
characters between codepages 1308
date into numeric calendar day 750
date value in character string (current date format) 905
date value in character string (YYYYMMDD) 908
date value in numeric weekday 902
date value into name of the month 700
date value into numeric month 1561
date value into the weekday name 634
date value to numeric year 2043
date value without century digits 466
DateTime in character string 1960
DateTime in character string (current date format) 1959
day number to day name 1603
degrees to radians 907
Fahrenheit to Celsius 636
float number to 8 byte binary 1011
Hex string to character string 1333
Hex string to Numeric value 1332
integer to string of digits 1602
logical value to character 1517
logical value to numeric 1518
month number to month name 1604
numeric value to 16 bit integer (signed) 1395
numeric value to 16 bit integer (unsigned) 1992
numeric value to 32 bit integer (signed) 1500
numeric value to 32 bit integer (unsigned) 1963
numeric value to character string 1889
numeric value to Hex string 1625
radians to degrees 1798
seconds to days 751
simple values to string 1315
string of digits to integer 742
with PICTURE format 1953

ConvToAnsiCP() 710
ConvToOemCP() 711
COPY FILE 369
COPY STRUCTURE 370
COPY STRUCTURE EXTENDED 371

COPY TO.......... 373
Copying
  a multi-dimensional array 540
  database structure.......... 370
  one-dimensional arrays 542
Cos().......... 712
CosH().......... 713
Cosine 712
  hyperbolic 713
Cot() 714
Cotangent 714
COUNT.......... 377
Counter.......... 1611
CountGets().......... 715
CountLeft() 716
CountRight() 717
CPU
  ticks to seconds 1121
  time usage 1824
CRC32 1134
CREATE.......... 379
CREATE FROM.......... 381
CreateObject().......... 718
Creating a compiler error 2128
CRLF 1416
Crypt().......... 725
CSetAtMuPa().......... 726
CSetCent().......... 728
CSetCurs().......... 729
CSetKey() 730
CSetRef() 731
CSetSafety().......... 733
CStr() 734
CStrToVal() 736
CSV files.......... 997
CtoBit() 737
CtoD() 738
CtoDoW() 739
CtoF().......... 740
CtoMonth() 741
CtoN() 742
CtoT().......... 743
Ctrl+Break.......... 1840
CurDir().......... 744
CurDirX().......... 745
CurDrive().......... 746
Current work area.......... 822
CurrentGet().......... 747
Cursor (text mode)
  changing the position 1866
  changing the shape 1847
  positioning on current output device.......... 856
  restoring 1786
  saving 1801
  switching on or off..........450, 729
Custom index
  adding a key 1653
  deleting a key 1657
Cyclic Redundancy Code.......... 1134
DATA 2166
Data types
  and scalar classes 2151
  as scalar classes..........2177
  checking for Array 1195
  checking for Character..........1210
  checking for Code block 1196
  checking for Date..........1199
  checking for DateTime 1200
  checking for Hash 1201
  checking for Logical 1202
  checking for Memo 1203
  checking for NIL..........1204
  checking for null values..........1205
  checking for Numeric 1206
  checking for Object..........1207
  checking for Pointer..........1208
  of expressions 1961
  of values..........1987
  testing empty values..........911
Database
  appending a record..........350, 754
  appending a record in network..........1575
  ascending or descending navigation 1644
  automatic commiting 481
  automatic opening of index..........439
  clearing a relation..........758
  closing..........364, 760
  closing all..........359
  closing of 759
  copying structure to file 371
  copying the structure..........370
  creating from a file..........764
  creating from an array..........766
  creating from structure extended file 381
  creating from two work areas 404
  creating from two work areas creating..........798
  creating structure extended file..........379
  date of the last change..........1520
  defining a link 829, 1678
  defining relation..........498
  descending navigation..........460
  displaying records..........408, 799
  field information 780
  file extension..........839
  filtering records..........476
  indexing 395, 1641
  indexing (compatibility) 769
  length of the file header 1329
  locking file..........989
  locking file in network..........1585
  locking scheme 453
  number of records..........1775
  opening 528, 845
  opening exclusively by default 474
  opening in network 1578
  opening index..........482
  read optimization 507
  writing all file buffers 367

writing file buffers 761

Database Drivers
- available 1761
- default driver 1765
- used in work area 1763

Database operation
- calculating the average 356
- calculating the sum of numeric expressions 520
- creating sum for numeric fields 523, 840
- number of records matching a condition 377
- sorting records 517, 835
- writing records from two work areas to file 404

Database processing
- listing records in a file 408, 799
- sequential search 410
- writing records from two work areas in one file ... 798

Database structure
- copying to new database 764
- defining in an array 766
- loading to array 837

Datagram 1418

Date format
- dd Month yyyy 897
- defining country-specific 451
- Month dd yyyy 1530

Date value
- adding months 546
- converting to character string (current date format) 905
- converting to character string (YYYYMMDD) 908
- converting to numeric weekday 902
- extracting calendar day as numeric 750
- extracting month as numeric 1561
- extracting the year 2043
- name of the month 700
- name of the weekday 634
- packing 1914
- retrieving the system date 748
- unpacking 1918
- without century digits 466

Date() 748

DateTime 2111
- converting to character string 1960
- converting to character string (current date format) 1959
- extracting Hour 1352
- extracting Minute 1552
- extracting Seconds 1826
- filled with system date and time 749

DateTime() 749

Day() 750

Days() 751

DaysInMonth() 752

DaysToMonth() 753

DbAppend() 754

DbClearFilter() 756

DbClearIndex() 757

DbClearRelation() 758

DbCloseAll() 759

DbCloseArea() 760

DbCommit() 761

DbCommitAll() 762

DbCopyExtStruct() 763

DbCopyStruct() 764

DbCreate() 766

DbCreateIndex() 769

DbDelete() 771

DbEdit() .... 772, *see also:* TBrowse object

DbEval() 777

Dbf() 779

DbFieldInfo() 780

DbFileGet() 782

DbFilePut() 783

DbFilter() 784

DbfSize() 786

DbGoBottom() 787

DbGoto() 788

DbGoTop() 790

DbInfo() 791

DbJoin() 798

Dblist() 799

DbOrderInfo() 801

DbRecall() 808

DbRecordInfo() 809

DbReindex() 811

DbRelation() 812

DbRLock() 814

DbRLockList() 816

DbRSelect() 817

DbRUnlock() 819

DbSeek() 820

DbSelectArea() 822

DbSetDriver() 824

DbSetFilter() 825

DbSetIndex() 827

DbSetOrder() 828

DbSetRelation() 829

DbSkip() 831

DbSkipper() 833

DbSort() 835

DbStruct() 837

DbTableExt() 839

DbTotal() 840

DbUnlock() 842

DbUnlockAll() 843

DbUpdate() 844

DbUseArea() 845

Debugger 565

Decimal numbers
- rounding 1795

declaring
- a class variable 2162

Declaring
- a class 2157
- a dynamic memory variable 2210
- a field variable 2189
- a function 2195
- a lexical memory variable 2198, 2208, 2238

a procedure 2231
a virtual method 2222
an instance method 2214
an instance variable 2166, 2247
external functions 2188, 2236
Decoding
a character string base 64 1100
a file base 64 1101
UUDecode a file 1312
UUDecode a string 1311
Decrement 2063
Decrypting
a character string 1140
Default output device 1705
for errors 1704
DEFAULT TO 383
Default values 383, 847
Default() 847
Deferred method 2222
define .... *see:* #define
Defining current work area 435
DefPath() 848
DELEGATE 2169
DELETE 384
DELETE FILE 386
DELETE TAG 387
Deleted() 849
DeleteFile() 851
Deleting
a group of files 969
all records 531
characters in a character string 1906
dynamic memory variable 359
entire screen output in text mode 362
file 386, 391, 941
index from an index file 1646
index key from index file 387
of records 384, 416, 771
of records in network 1580
recalling records 420
removing the deletion flag for records 808
removing the deletion flag for records in network 1591
screen output in text mode 320
Deletion flag
checking if set 849
Descend() 852
Deserialization 1142
complex data types (first) 1141
complex data types (next) 1144
next data 1286
retrieving memory requirements 1186
simple data types 1143
DESTRUCTOR 2171
DevOut() 854
DevOutPict() 855
DevPos() 856
DirChange() 857
Directives
for commands 2118
Directory
change current 857
check if it exists 1466, 1467
completing name 1957
creating 862
creating a directory 1521
defining default directory 456
determining current 744, 745
loading into array 858
loading recursively into array 860
name of 863
reading to arrays 549
removing 864
Directory() 858
DirectoryRecurse() 860
DirMake() 862
DirName() 863
DirRemove() 864
DisableWaitLocks() 865
Disk .... *see also:* Floppy disk
changing current 866
check for readiness 872
check if drive is ready 1468
check if it can be written to 873
get current 871
Disk drive 904
DiskChange() 866
DiskFormat() 868
DiskFree() 870
DiskName() 871
DiskReady() 872
DiskReadyW() 873
DiskSpace() 874
DiskTotal() 876
DiskUsed() 877
DispBegin() 878
DispBox() 880
DispCount() 882
DispEnd() 883
DispOut() 884
DispOutAt() 885
DispOutAtSetPos() 887
Division 2069
DLL file
loading external at runtime 1512
loading xHarbour DLL at runtime 1511
unloading external 1006
unloading xHarbour DLL 1510
DLL function
calling dynamically at runtime 888, 1219
calling dynamically from pointer 632
memory address of 1043
preparing a call template 894
retrieving error code 1033
setting error code 1857
using a call template 891
DllCall() 888
DllExecuteCall() 891

DllLoad() ... 893
DllPrepareCall() ... 894
DllUnload() ... 896
DMY() ... 897
DO ... 389
DO CASE ... 2173
  functional equivalent ... 1137
DO WHILE ... 2175
DosError() ... 898
DosParam() ... 901
DoW() ... 902
DoY() ... 903
Drive
  available storage space ... 1145
  changing current ... 866
  check if drive is ready ... 1468
  check network ... 1581
  current directory ... 744, 745
  defining default drive ... 456
  determining available storage space ... 874
  determining or changing current ... 746
  free storage space ... 870
  get current ... 871
  getting label ... 1052
  separator ... 1239
  serial number ... 1990
  setting label ... 1991
  total storage space ... 876
  type of ... 904
  used storage space ... 877
DriveType() ... 904
DtoC() ... 905
DtoR() ... 907
DtoS() ... 908
Duration of a loan ... 1714
Dynamic memory variable
  creating set/get code block ... 1546
  declaring ... 2210
  deleting ... 359, 361, 423
  determining the data type ... 1961
  initializing and creating ... 2229, 2234
  loading from a MEM file ... 429
  saving to file ... 432
EJECT ... 390
ElapTime() ... 910
ellipsis ... 2196
EMail
  creating ... 216
  reading ... 205
  sending ... 210
Empty values ... 602, 911
Empty() ... 911
ENABLE TYPE CLASS ... 2177
Encoding
  a character string base 64 ... 1102
  a file base 64 ... 1103
  UUEncode a file ... 1314
  UUEncode a string ... 1313
Encrypting
  a character string ... 1136
End of file ... 914
Enhanced() ... 913
Entry field
  activating ... 418
  with Get command ... 322
Eof() ... 914
EoM() ... 916
EoQ() ... 917
EoY() ... 918
Equal ... 2083
ERASE ... 391
error ... *see also:* #error
Error codes
  of operating system ... 898
ERROR HANDLER ... 2179
Error handling
  database error in network ... 1582
  default error code block ... 926
  default error log file ... 467
  error code of the operating system ... 898
  in classes ... 2179
  maximum recursion in ... 468
  setting error code block ... 919
  throwing an exception ... 1928
Error message
  displaying during compiling ... 2128
Error object ... 11
  creating ... 924
Error() ... 11
ErrorBlock() ... 919
ErrorLevel() ... 922
ErrorNew() ... 924
ErrorSys() ... 926
Esc key for READ ... 469
Eval() ... 927
Events
  filtering ... 470
EXE file
  name of ... 929, 1124
Executable array ... 1152
ExeName() ... 929
Existence
  of a Directory ... 958
  of a file ... 958
  of a value in a database ... 995
EXIT PROCEDURE ... 2181
Exp() ... 930
Expand() ... 931
Exponent() ... 932
Exponentiation ... 2057
EXPORTED: ... 2183
exporting
  records ... 373
Expression
  depending on a condition ... 1397
  determining the data type ... 1961, 1987
  displaying on screen ... 316, 1745
  formatting result with PICTURE format ... 1953

EXTEND CLASS...WITH DATA 2184
EXTEND CLASS...WITH METHOD 2186
Extension
  for database files 839
  for index files 1633
  for index files (compatibility) 1399
EXTERNAL 2188
Extracting
  lines from text 1260, 1538
Fact() 933
Factorial 933
Fahrenheit() 934
FCharCount() 935
FClose() 936
FCount() 938
FCreate() 939
FErase() 941
FError() 942
Field
  declaring 2189
  export to external file 782
  import an external file 783
FIELD 2189
Field information 780
Field structure
  copying to file 371
Field variable
  assigning a value 427
FieldBlock() 943
FieldDec() 945
FieldDeci() 946
FieldGet() 947
FieldLen() 948
FieldName() 949
FieldNum() 950
FieldPos() 951
FieldPut() 952
Fields ....*see also:* Memo field
  creating a set/get code block 943
  creating a set/get-code block 956
  data type of 955
  determining the data type 1961
  determining the number of 938
  length of field 948
  number of decimal places 945
  reading the value 947
  retrieving the name by ordinal position 949
  retrieving the ordinal position by name 951
  writing a value 952
FieldSize() 954
FieldType() 955
FieldWBlock() 956
File ....*see:* king multiple
  backup copy 966
  compressing 1916
  concatenating 960
  copying to an output device 369
  decompressing 1917
  defining search path 495
  deleting 386, 391, 851
  deleting one or more 969
  information on 978
  last change date 968
  last time changed 981
  loading name and attributes into array 858
  loading name and attributes recursively into array860
  moving 970
  reading attributes 961
  reading completely from disk 1541
  reading part of 980
  renaming 425, 1007, 1781
  safety switch 733
  setting attributes 1851
  setting date and time 1853
  size of 976
  testing the existence 958
  validating name 983
  writing part of 1894
File handle 993
file header 1329
File lock
  releasing 525, 842
  releasing all 843
  setting 989
  setting in network 1585
File name
  composing 1165
  splitting 1166
  temporary 1176
File pointer
  positioning in a file 1009
File size 1174
File() 958
FileAppend() 960
FileAttr() 961
FileCClose() 963
FileCCont() 964
FileCOpen() 965
FileCopy() 966
FileDate() 968
FileDelete() 969
FileMove() 970
Filename
  validating 983
FileReader() 971
FileScreen() 973
FileSeek() 974
FileSize() 976
FileStats() 978
FileStr() 980
FileTime() 981
FileValid() 983
FileWriter() 985
Fill character 1706
Filter
  defining or releasing 476
  optimization 492
Filter condition

clearing of ... 756
retrieving current ... 784
setting current ... 825
FINALLY ... 2242
FIND ... 392
FkLabel() ... 986
FkMax() ... 987
FLineCount() ... 988
Floating point number
exponent ... 932
mantissa ... 1523
FLock() ... 989
Floor() ... 991
Floppy disk ... 992
formatting ... 868
FloppyType() ... 992
FOpen() ... 993
FOR ... 2191
FOR condition
and index creation ... 1636
for index creation ... 395
for sequential search ... 410
in database processing ... 777
of an index ... 1649
FOR EACH ... 2193
Formatted output
choosing the date format ... 451
choosing the time format ... 508
fixed number of decimal places ... 478
number of decimal places ... 455
to current output device ... 855
Found() ... 995
FParse() ... 997
FParseEx() ... 999
FParseLine() ... 1000
FRead() ... 1002
FReadStr() ... 1004
FreeLibrary() ... 1006
FRename() ... 1007
FSeek() ... 1009
FtoC() ... 1011
FTP ... 193
Function
number of passed arguments ... 1713
FUNCTION ... 2195
getting all parameters ... 1079
getting the pointer ... 1178
Function key
associating with a procedure ... 479
Function pointer
executing ... 1150
obtaining ... 1178
operator ... 2095
Future value ... 1012
Fv() ... 1012
FWordCount() ... 1013
FWrite() ... 1014
Garbage collector ... 1184
calling a method ... 2171
incremental collection ... 1185
Get object ... 19
create with function ... 1034
creating and displaying ... 322
editing ... 418
Get system
activating ... 418, 1771
current Get ... 747
current Get object ... 1016
defining exit keys ... 1767
delimiters for entry fields ... 459
executing SetKey() code block ... 1027
exit key ... 1769
getting Updated flag ... 1968
hidden input ... 1047
name of edited variable ... 1032
number of Gets ... 715
post validation ... 1038, 1054
pre validation ... 1040, 1055
prosessing user input ... 1056
restoring Gets ... 1787
saving Gets ... 1802
screen column of Get ... 1030
screen row of Get ... 1031
sending key codes ... 1019
sending key codes to control ... 1053
setting Updated flag ... 1773
terminating input ... 360
terminating READ ... 1770
toggle insert mode ... 1768
Get() ... 19
GetActive() ... 1016
GetActiveObject() ... 1017
GetApplyKey() ... 1019
GetClearA() ... 1020
GetClearB() ... 1021
GetClrBack() ... 1022
GetClrFore() ... 1023
GetClrPair() ... 1024
GetCurrentThread() ... 1025
GetDefaultPrinter() ... 1026
GetDoSetKey() ... 1027
GetE() ... 1028
GetEnv() ... 1029
GetFldCol() ... 1030
GetFldRow() ... 1031
GetFldVar() ... 1032
GetLastError() ... 1033
GetNew() ... 1034
GetPairLen() ... 1036
GetPairPos() ... 1037
GetPostValidate() ... 1038
GetPrec() ... 1039
GetPreValidate() ... 1040
GetPrinters() ... 1041
GetProcAddress() ... 1043
GetRegistry() ... 1045
GetSecret() ... 1047
GetSystemThreadID() ... 1049

GetThreadID() 1050
GetVolInfo() 1052
GLOBAL 2198
GO 393
Greater than 2089
Greater than or equal 2091
GuiApplyKey() 1053
GuiGetPostValidate() 1054
GuiGetPrevalidate() 1055
GuiReader() 1056
HaaDelAt() 1057
HaaGetKeyAt() 1059
HaaGetPos() 1061
HaaGetRealPos() 1063
HaaGetValueAt() 1065
HaaSetValueAt() 1067
HAllocate() 1069
HardCR() 1070
HAS 2097
Hash 2109
  as associative array 1357
  associate value with key 1355
  automatic creation of elements 1337
  case sensitive keys 1338
  change value by ordinal position 1365
  check if a key exists 1350
  check partitioning 1343
  copy entirely 1351
  copy partially 1324
  creating 1071
  entire copy 1322
  evaluate a code block 1330
  filling with a unique value 1334
  number of key/value pairs 1507
  ordinal key position 1344
  partitioning 1363
  pre-allocating memory 1069
  removing key/value pair by key 1327
  removing key/value pair by ordinal 1328
  retrieve all keys 1341
  retrieve all values 1349
  retrieve key by ordinal position 1340
  retrieve key/value pair by ordinal position 1342
  retrieve value by ordinal position 1348
  retrieving value by key 1335
  search a value in a hash 1353
  test for associative array 1336
  toggle automatic creation of elements 1359
  toggle case sensitivity 1361
Hash() 1071
HB_AExpressions() 1075
HB_AnsiToOem() 1077
HB_AParams() 1079
HB_ArgC() 1080
HB_ArgCheck() 1081
HB_ArgString() 1082
HB_ArgV() 1083
HB_ArrayBlock() 1085
HB_ArrayId() 1086
HB_ArrayToStructure() 1087
HB_ATokens() 1089
HB_AtX() 1091
HB_BackGroundActive() 1093
HB_BackGroundAdd() 1094
HB_BackGroundDel() 1096
HB_BackGroundReset() 1097
HB_BackGroundRun() 1098
HB_BackGroundTime() 1099
HB_Base64Decode() 1100
HB_Base64DecodeFile() 1101
HB_Base64Encode() 1102
HB_Base64EncodeFile() 1103
HB_BitAnd() 1104
HB_BitIsSet() 1105
HB_BitNot() 1107
HB_BitOr() 1108
HB_BitReset() 1109
HB_BitSet() 1111
HB_BitShift() 1113
HB_BitXOr() 1115
HB_BldLogMsg() 1116
HB_BuildDate() 1117
HB_BuildInfo() 1118
HB_CheckSum() 1120
HB_Clocks2Secs() 1121
HB_CloseProcess() 1122
HB_ClrArea() 1123
HB_CmdArgArgV() 1124
HB_ColorIndex() 1125
HB_ColorToN() 1127
HB_Compiler() 1128
HB_Compress() 1129
HB_CompressBufLen() 1131
HB_CompressError() 1132
HB_CompressErrorDesc() 1133
Hb_CRC32() 1134
HB_CreateLen8() 1135
HB_Crypt() 1136
HB_Decode() 1137
HB_DecodeOrEmpty() 1139
HB_Decrypt() 1140
HB_DeserialBegin() 1141
HB_DeSerialize() 1142
HB_DeserializeSimple() 1143
HB_DeserialNext() 1144
HB_DiskSpace() 1145
HB_DumpVar() 1147
HB_EnumIndex() 1148
HB_Exec() 1150
HB_ExecFromArray() 1152
HB_F_Eof() 1183
HB_FCommit() 1156
HB_FCreate() 1157
HB_FEof() 1159
HB_FGoBottom() 1160
HB_FGoto() 1161
HB_FGoTop() 1162
HB_FInfo() 1163

HB_FLastRec()......1164
HB_FNameMerge()......1165
HB_FNameSplit()......1166
HB_FReadAndSkip() 1168
HB_FReadLine() 1169
HB_FreadLN()......1171
HB_FRecno()......1172
HB_FSelect()......1173
HB_FSize()......1174
HB_FSkip() 1175
HB_FTempCreate()......1176
HB_FuncPtr() 1178
HB_FUse() 1179
HB_GCAll() 1184
HB_GCStep() 1185
HB_GetLen8()......1186
HB_IdleAdd()......1187
HB_IdleDel()......1189
HB_IdleReset()......1190
HB_IdleSleep()......1191
HB_IdleSleepMSec()......1192
HB_IdleState()......1193
HB_IdleWaitNoCPU() 1194
HB_IsArray()......1195
HB_IsBlock()......1196
HB_IsByRef 1197
HB_IsDate()......1199
HB_IsDateTime() 1200
HB_IsHash()......1201
HB_IsLogical()......1202
HB_IsMemo()......1203
HB_IsNIL() 1204
HB_IsNull()......1205
HB_IsNumeric() 1206
HB_IsObject() 1207
HB_IsPointer()......1208
HB_IsRegExString()......1209
HB_IsString() 1210
HB_KeyPut()......1211
HB_LangErrMsg()......1212
HB_LangMessage()......1213
HB_LangName() 1215
HB_LangSelect() 1216
HB_LibDo()......1219
HB_LogDateStamp()......1221
HB_MacroCompile()......1222
HB_MD5() 1223
HB_MD5File() 1224
HB_MultiThread()......1225
HB_MutexCreate() 1226
HB_MutexLock() 1229
HB_MutexTimeoutLock()......1230
HB_MutexTryLock()......1231
HB_MutexUnlock()......1232
HB_ObjMsgPtr() 1233
HB_OemToAnsi() 1235
HB_OpenProcess() 1236
HB_OsDriveSeparator() 1239
HB_OsError() 1240
HB_OsNewLine() 1241
HB_OsPathDelimiters() 1242
HB_OsPathListSeparator()......1244
HB_OsPathSeparator()......1245
HB_PCodeVer() 1246
HB_Pointer2String() 1247
HB_ProcessValue()......1249
HB_QSelf() 1250
HB_QWith()......1251
HB_Random() 1252
HB_RandomInt()......1254
HB_RandomSeed() 1256
HB_ReadIni()......1258
HB_ReadLine() 1260
HB_RegEx()......1263
HB_RegExAll()......1266
HB_RegExAtX()......1270
HB_RegExComp() 1272
HB_RegExMatch()......1273
HB_RegExReplace() 1275
HB_RegExSplit() 1277
HB_ResetWith()......1279
HB_RestoreBlock()......1281
HB_SaveBlock() 1282
HB_Serialize()......1283
HB_SerializeSimple() 1285
HB_SerialNext()......1286
HB_SetCodePage() 1287
HB_SetIniComment() 1289
HB_SetKeyArray()......1290
HB_SetKeyCheck()......1291
HB_SetKeyGet() 1292
HB_SetKeySave() 1293
HB_SetMacro() 1295
HB_SetWith()......1296
HB_Shadow()......1298
HB_SizeofCStructure() 1299
HB_String2Pointer() 1301
HB_StructureToArray() 1302
HB_SysLogClose() 1304
HB_SysLogMessage()......1305
HB_SysLogOpen()......1306
HB_ThisArray() 1307
HB_Translate()......1308
HB_Uncompress()......1309
HB_UUDecode()......1311
HB_UUDecodeFile()......1312
HB_UUEncode()......1313
HB_UUEncodeFile()......1314
HB_ValToStr()......1315
HB_VMExecute() 1316
HB_VMMode()......1317
HB_WithObjectCounter() 1318
HB_WriteIni() 1319
HB_XmlErrorDesc() 1321
HbCheckBox()......37
HbConsoleLock() 1073
HbConsoleUnlock()......1074
HbListBox() 44

HBObject() ... 60
HBPersistent() ... 66
HbPushButton() ... 69
HbRadioButton() ... 75
HbRadioGroup() ... 82
HbScrollBar() ... 93
HClone() ... 1322
HCopy() ... 1324
HDel() ... 1327
HDelAt() ... 1328
Header() ... 1329
HEval() ... 1330
Hex string
  converting to character string ... 1333
  converting to Numeric value ... 1332
HexToNum() ... 1332
HexToStr() ... 1333
HFill() ... 1334
HGet() ... 1335
HGetAACompatibility() ... 1336
HGetAutoAdd() ... 1337
HGetCaseMatch() ... 1338
HGetKeyAt() ... 1340
HGetKeys() ... 1341
HGetPairAt() ... 1342
HGetPartition() ... 1343
HGetPos() ... 1344
HGetVaaPos() ... 1346
HGetValueAt() ... 1348
HGetValues() ... 1349
HHasKey() ... 1350
HIDDEN: ... 2200
HiPer-SEEK
  adding an entry ... 1367
  changing an index entry ... 1386
  check deletion flag ... 1377
  closing file ... 1369
  creating file ... 1370
  creating populated index file ... 1378
  defining search ... 1388
  deleting entry ... 1373
  finding next index entry ... 1382
  number of index entries ... 1381
  open a file ... 1384
  setting filter ... 1375
  undeleting entry ... 1389
  verifying a match ... 1390
  version information ... 1392
HMerge() ... 1351
Hour() ... 1352
HS_Add() ... 1367
HS_Close() ... 1369
HS_Create() ... 1370
HS_Delete() ... 1373
HS_Filter() ... 1375
HS_IfDel() ... 1377
HS_Index() ... 1378
HS_KeyCount() ... 1381
HS_Next() ... 1382
HS_Open() ... 1384
HS_Replace() ... 1386
HS_Set() ... 1388
HS_Undelete() ... 1389
HS_Verify() ... 1390
HS_Version() ... 1392
HScan() ... 1353
HSet() ... 1355
HSetAACompatibility() ... 1357
HSetAutoAdd() ... 1359
HSetCaseMatch() ... 1361
HSetPartition() ... 1363
HSetValueAt() ... 1365
HTML
  converting to ANSI ... 1393
  converting to OEM ... 1394
  create a HTML node object ... 166
  create HTML document object ... 152
  initializes HTML lookup tables ... 1930
  releases HTML lookup tables ... 1929
  searching document with RegEx ... 162
  searching node in document ... 164
  traversing document ... 159
  verifying tag and/or attribute names ... 1931
HtmlToAnsi() ... 1393
HtmlToOem() ... 1394
HTTP ... 202
I2Bin() ... 1395
Idle tasks ... 1187
  default wait period ... 1192
  deleting ... 1189
  resetting ... 1190
  signalling ... 1193
  wait state ... 1191
  waiting without CPU ... 1194
if *see:* #if
IF2202
If() ... 1397
If() | IIf() ... 1397
ifdef ... *see:* #ifdef, *see:* #ifdef
IIf() ... 1397
Importing
  records ... 352
IN ... 2099
include ... *see:* #include
Increment ... 2060
Index
  activating ... 1666
  automatic opening ... 439
  automatic selecting ... 441
  changing custom status ... 1643
  changing descend order ... 1644
  closing ... 364
  closing files ... 1668
  closing files (compatibility) ... 757
  creating ... 395, 1641
  creating (compatibility) ... 769
  current value ... 1665
  custom key ... 1653

defining UNIQUE status ... 512
deleting from an index file ... 387, 1646
descending flag ... 460
determining the TAG name ... 1670
extension ... 1633
getting position by name ... 1672
open file ... 827
opening file ... 482
position of the controlling ... 1402
remove custom key ... 1657
reorganizing ... 422, 1669
reorganizing (compatibility) ... 811
retrieving FOR condition ... 1649
searching a record ... 433, 820
searching a record number ... 1647
selecting the controlling ... 493, 1676
selecting the controlling (compatibility) ... 828
setting conditions for index creation ... 1636
testing UNIQUE flag ... 1650
total number of keys ... 1655
wild card searching ... 1683

INDEX ... 395

Index key
creating for descending order ... 852
name of the corresponding index file ... 1634
retrieve expression ... 1651
retrieving (compatibility) ... 1400

IndexExt() ... 1399
IndexKey() ... 1400
IndexOrd() ... 1402
INetAccept() ... 1403
INetAddress() ... 1404
INetCleanup() ... 1405
INetClearError() ... 1406
INetClearPeriodCallback() ... 1407
INetClearTimeout() ... 1408
INetClose() ... 1409
INetConnect() ... 1410
INetConnectIP() ... 1412
INetCount() ... 1413
INetCreate() ... 1415
INetCRLF() ... 1416
INetDataReady() ... 1417
INetDGram() ... 1418
INetDGramBind() ... 1419
INetDGramRecv() ... 1420
INetDGramSend() ... 1421
INetErrorCode() ... 1425
INetErrorDesc() ... 1426
INetGetAlias() ... 1427
INetGetHosts() ... 1428
INetGetPeriodCallback() ... 1430
INetGetTimeout() ... 1431
INetInit() ... 1432
INetPort() ... 1433
INetRecv() ... 1434
INetRecvAll() ... 1436
INetRecvEndblock() ... 1437
INetRecvLine() ... 1439
INetSend() ... 1440
INetSendAll() ... 1442
INetServer() ... 1443
INetSetPeriodCallback() ... 1447
INetSetTimeout() ... 1449
Infinity() ... 1450

INI file
comment delimiters ... 1289
reading ... 1258
writing ... 1319

INIT LOG ... 399
INIT PROCEDURE ... 2204
Inkey() ... 1451, *see also:* SET EVENTMASK
Inline assignment ... 2072
INLINE METHOD ... 2206
INPUT ... 403

Inserting
characters in a character string ... 1906
include file in a program file ... 2134

Instance method
declaring ... 2214

Int() ... 1453
Integers ... 1453
Interest rate ... 1757

Internet
connecting ... 1412
connecting to server ... 1410
listening for connections ... 1443

Internet address ... 1404

Internet client
abstract class ... 186
for eMail reading ... 205
for eMail sending ... 210
for file exchange ... 193
for WWW ... 202

InvertAttr() ... 1454
InvertWin() ... 1455
Is identical ... 2087
IsAffirm() ... 1457
IsAlNum() ... 1458
IsAlpha() ... 1459
IsAscii() ... 1460
IsBit() ... 1461
IsCntrl() ... 1462
IsColor() ... 1463
IsDefColor() ... 1464
IsDigit() ... 1465
IsDir() ... 1466
IsDirectory() ... 1467
IsDisk() ... 1468
IsGraph() ... 1470
IsLeap() ... 1471
IsLocked() ... 1472
IsLower() ... 1473
IsNegative() ... 1474
IsPrint() ... 1475
IsPrinter() ... 1476
IsPunct() ... 1478
IsSameThread() ... 1479

IsSpace() ... 1481
IsUpper() ... 1482
IsValidThread() ... 1483
IsXDigit() ... 1485
Iteration
  for databases ... 777
JOIN ... 404
JoinThread() ... 1486
JustLeft() ... 1488
JustRight() ... 1489
KbdStat() ... 1490
Key
  associated code block ... 730
  associated code blocks ... 1293
  associating with a procedure call ... 485
  associating with code block ... 1854
  Caps lock ... 1496
  changing last ... 1858
  evaluating code block ... 1291
  insert/overwrite ... 1497
  Num lock ... 1498
  query associated code block ... 1292
  restoring associated code blocks ... 1790
  saving associated code blocks ... 1805
  Scroll lock ... 1499
Keyboard
  special keys ... 1490
KEYBOARD ... 406
Keyboard buffer
  defining the size ... 511
  reading and removing next key ... 1451
  removing all characters ... 363
  retrieve last key ... 1503
  retrieve next key ... 1596
  writing characters ... 406
  writing characters at a time ... 1493
  writing characters with timer ... 1492
KeySec() ... 1492
KeyTime() ... 1493
KillAllThreads() ... 1494
KillThread() ... 1495
KSetCaps() ... 1496
KSetIns() ... 1497
KSetNum() ... 1498
KSetScroll() ... 1499
L2bin() ... 1500
Language ... 1216
  application messages ... 1573
  current ... 1215
  error messages ... 1212
  regular messages ... 1213
  user confirmation ... 1457
  user negation ... 1474
Larger value of two values ... 1524
Last record ... 1505
LastDayoM() ... 1502
LastKey() ... 1503
LastRec() ... 1505
Left shift ... 2075
Left() ... 1506
Len() ... 1507
Length
  of a record ... 1777
  of an array character string or hash ... 1507
LenNum() ... 1509
Less than ... 2073
Less than or equal ... 2076
Lexical memory variable
  determining the data type ... 1987
  initializing and declaring ... 2198, 2208, 2238
LibFree() ... 1510
LibLoad() ... 1511
LIKE ... 2101
Line end characters ... 1241
Link expression of a relation ... 812
LIST ... 408
List box (text mode) ... 44
Listbox
  in text mode ... 535
Literal value
  array ... 2108
  code block ... 2113
  DateTime ... 2111
  hash ... 2109
LoadLibrary() ... 1512
LOCAL ... 2208
Localization ... 1216
  application messages ... 1573
  current language ... 1215
  error messages ... 1212
  regular messages ... 1213
  user confirmation ... 1457
  user negation ... 1474
LOCATE ... 410
Locking schemes ... 453
LOG ... 412
Log file
  date stamp ... 1221
  parameter values ... 1116
Log system ... *see also:* System log file
  closing channels ... 366
  formatting rules for log entries ... 486
  opening log channels ... 399
  sending log messages ... 412
Log() ... 1514
Log10() ... 1515
Logarithm
  base 10 ... 1515
  natural ... 1514
Logical AND ... 2066
Logical NOT ... 2067
Logical OR ... 2068
Logical order of records ... 1676
Low level File
  information on ... 978
Lower case
  converting to upper case ... 1969
Lower() ... 1516

Low-level file
abstract stream object ... 227
closing ... 936
committing data ... 1156
counting text lines ... 988
creating ... 939
creating or opening ... 1157
deleting ... 941
navigating the file pointer ... 1009
opening ... 993
opening exclusive in network ... 865
reading ... 1002
reading a text line ... 1169
reading part of ... 980
reading up to Chr(0) ... 1004
renaming ... 1007
retrieving last error code ... 942
retrieving last OS error code ... 1240
size ... 1174
stream reader object ... 229
stream writer object ... 234
testing for End-of-file ... 1183
writing ... 1014
writing part of ... 1894
LtoC() ... 1517
LtoN() ... 1518
LTrim() ... 1519
LUpdate() ... 1520
Macro operator ... 2049
accessing variables ... 2050
calling functions ... 2052
creating variables ... 2050
runtime compilation ... 2051
sending message to object ... 2052
text substitution ... 2051
toggle features ... 1295
Macro string ... 1075
compiling ... 1222
MakeDir() ... 1521
Mantissa() ... 1523
Max() ... 1524
MaxCol() ... 1525
MaxLine() ... 1526
MaxRow() ... 1527
MCol() ... 1528
MD5 ... 1223
MDblClk() ... 1529
MDY() ... 1530
MEM file
load into memory ... 429
Member variable ... 2166, 2247
Memo field
displaying and editing (text mode) ... 1531
exporting to file ... 612
extracting one line of text ... 1538
importing a file ... 615
number of lines ... 1553
position of a character ... 1555
position of a line ... 1558
reading the value ... 614
replacing characters ... 1902
replacing of carriage returns ... 1544
replacing soft carriage return ... 1070
row and column position of a character ... 1562
scanning lines of text ... 1260
writing to a file ... 1545
Memo file
block size ... 489
MemoEdit() ... 1531
MemoLine() ... 1538
MemoRead() ... 1541
Memory() ... 1542
MemoTran() ... 1544
MemoWrit() ... 1545
MEMVAR ... 2210
MemVarBlock() ... 1546
Menu
activating in text mode ... 414
displaying in text mode ... 343
Menu classes (text-mode)
activating menu system ... 1547
menu item ... 99
popup menu ... 103
top bar menu ... 141
MENU TO ... 414
MenuItem() ... 99
MenuModal() ... 1547
MESSAGE ... 2211
Message digest
of a file ... 1224
of a string ... 1223
METHOD
getting the pointer ... 1233
METHOD (declaration) ... 2214
METHOD (implementation) ... 2217
Method declaration
ACCESS ... 2145
ASSIGN ... 2148
METHOD...OPERATOR ... 2220
METHOD...VIRTUAL ... 2222
MHide() ... 1549
MilliSec() ... 1550
Min() ... 1551
Minus operator ... 2061
Minute() ... 1552
MLCount() ... 1553
MLCToPos() ... 1555
MLeftDown() ... 1557
MlPos() ... 1558
Mod() ... 1560
Modal dialog box
in text mode ... 561
Modulus operator ... 2046
Month ... 625
days in ... 752
first day of ... 625
last day of ... 916, 1502
Month() ... 1561

Mouse
number of buttons ... 1610
retrieving event ... 1451
Mouse (text mode)
check if it exists... 1564
double click interval... 1529
restore saved settings ... 1565
restricting boundaries... 1569
save current settings ... 1568
status of left button... 1557
status of right button ... 1566
Mouse cursor
turning on/off ... 1861
Mouse cursor (text mode)
current column position... 1528
current row position ... 1567
determine visibility ... 1570
displaying mouse cursor... 1572
hiding mouse cursor ... 1549
set position ... 1571
set position in window ... 2022
Mouse events
filtering... 470
MPosToLC() ... 1562
MPresent()... 1564
MRestState() ... 1565
MRightDown()... 1566
MRow()... 1567
MSaveState()... 1568
MSetBounds() ... 1569
MSetCursor() ... 1570
MSetPos() ... 1571
MShow() ... 1572
Multi-dimensional array
copying... 540
Multi-dimensional Array
creating... 571
Multiplication... 2056
Multi-threading ... 1225
Mutex
creating... 1226
locking permanently... 1229
locking with timeout ... 1230
subscribing ... 1908
trying to lock ... 1231
unlocking... 1232
Mutual exclusive ... 1226
Name
changing for a file ...425, 1007
of an index... 1670
of an index file from index key ... 1634
of fields in a work area... 949
of the operating systems... 1685
National language ... 1216
Nationalization
application messages... 1573
current language... 1215
error messages... 1212
regular messages ... 1213
user confirmation ...1457
user negation ...1474
NationMsg()...1573
Navigation
absolute ...393, 788
ignoring filter ...1679
logical ...1659
relative ...515, 831
to unique records...1681
with TBrowse objects ...833
NetAppend() ...1575
NetCancel()...1576
NetCommitAll()...1577
NetDbUse()...1578
NetDelete() ...1580
NetDisk() ...1581
NetErr()...1582
NetError() ...1584
NetFileLock() ...1585
NetFunc()...1586
NetLock()...1587
NetName() ...1588
NetOpenFiles() ...1589
NetPrinter()...1590
NetRecall()...1591
NetRecLock() ...1592
NetRedir()...1593
NetRmtName() ...1594
Network
check drive...1581
check for driver...1701
check for printer...1590
check registry...1699
color for messages ...1863
connecting...1593
creating new database record ...1575
deleting records...1580
error in database command ...1582
error in network function ...1584
evaluating a code block...1586
file lock during concurrent access...989, 1585
locking records...1592
locking schemes...453
obtain user name or computer name ...1588
opening database...1578
opening database and index ...1589
recalling records...1591
releasing a record lock ...819
releasing connection ...1576
remote name...1594
testing existence...1595
testing Novell...1598
timeout period...1862
Network()...1595
NextKey() ...1596
NNetwork()...1598
Not equal...2078
NOT operator...2067
Notify() ...1599

NotifyAll() ... 1600
NtoC() ... 1602
NtoCDoW() ... 1603
NtoCMonth() ... 1604
NtoColor() ... 1605
Nul() ... 1606
Null string ... 1606
Num lock ... 1498
NumAND() ... 1607
NumAndX() ... 1608
NumAt() ... 1609
NumButtons() ... 1610
NumCount() ... 1611
NumDiskL() ... 1612
Numeric value
  antilogarithm ... 930
  calculating the natural logarithm ... 1514
  calculating the square root ... 1881
  converting to character ... 689
  converting to character string ... 1889
  converting to Hex string ... 1625
  converting to integer ... 1453
  determining the absolute value ... 534
  largest ... 1450
  next larger integer ... 635
  next smaller integer ... 991
  padding with a fill character ... 1706
  padding with leading zeros ... 1904
  required places for display ... 1509
  rounding ... 1795
NumHigh() ... 1613
NumLine() ... 1614
NumLow() ... 1615
NumMirr() ... 1616
NumMirrX() ... 1617
NumNOT() ... 1618
NumNotX() ... 1619
NumOR() ... 1620
NumOrX() ... 1621
NumRoL() ... 1622
NumRolX() ... 1624
NumToHex() ... 1625
NumToken() ... 1626
NumXOR() ... 1627
NumXorX() ... 1628
Object
  declaring a method ... 2214
  from pointer ... 1307
  method pointer ... 1233
  unique identifier ... 1086
Occurs() ... 1629
OEM
  conversion of character strings ... 711, 1235
  converting to HTML ... 1630
OemToHtml() ... 1630
OLE
  creating object ... 718
  last error as string ... 1631
  last error code ... 1632
  retrieves existing object ... 1017
Ole2TxtError() ... 1631
OleError() ... 1632
One-dimensional Array
  copying elements ... 542
Operating system
  check for Windows 2000 ... 1686
  check for Windows 2000 or later ... 1687
  check for Windows 2003 ... 1688
  check for Windows 95 ... 1689
  check for Windows 98 ... 1690
  check for Windows 9x ... 1691
  check for Windows ME ... 1692
  check for Windows NT ... 1693
  check for Windows NT 3.51 ... 1694
  check for Windows NT 4.0 ... 1695
  check for Windows Terminal Server Client ... 1698
  check for Windows Vista ... 1696
  check for Windows XP ... 1697
  retrieving error code ... 898
  retrieving the contents of environment variables 1029
  retrieving the version ... 1685
  version information ... 1702
OPERATOR ... 2223
Operator overloading ... 2220
Optimization ... 492
OR operator ... 2068
OrdBagExt() ... 1633
OrdBagName() ... 1634
OrdCondSet() ... 1636
OrdCount() ... 1639
OrdCreate() ... 1641
OrdCustom() ... 1643
OrdDescend() ... 1644
OrdDestroy() ... 1646
OrdFindRec() ... 1647
OrdFor() ... 1649
OrdIsUnique() ... 1650
OrdKey() ... 1651
OrdKeyAdd() ... 1653
OrdKeyCount() ... 1655
OrdKeyDel() ... 1657
OrdKeyGoTo() ... 1659
OrdKeyNo() ... 1661
OrdKeyRelPos() ... 1663
OrdKeyVal() ... 1665
OrdListAdd() ... 1666
OrdListClear() ... 1668
OrdListRebuild() ... 1669
OrdName() ... 1670
OrdNumber() ... 1672
OrdScope() ... 1674
OrdSetFocus() ... 1676
OrdSetRelation() ... 1678
OrdSkipRaw() ... 1679
OrdSkipUnique() ... 1681
OrdWildSeek() ... 1683
Os() ... 1685
Os_IsWin2000() ... 1686

Os_IsWin2000_Or_Later() 1687
Os_IsWin2003() 1688
Os_IsWin95() 1689
Os_IsWin98() 1690
Os_IsWin9X() 1691
Os_IsWinME() 1692
Os_IsWinNT() 1693
Os_IsWinNT351() 1694
Os_IsWinNT4() 1695
Os_IsWinVista() 1696
Os_IsWinXP() 1697
Os_IsWtsClient() 1698
Os_NetRegOk() 1699
OS_NetVRedirOk() 1701
Os_VersionInfo() 1702
OutErr() 1704
Output (text mode)
  changing the position 1866
  defining output device 461
  displaying a box on screen 318, 880
  displaying a frame on screen 347
  formatted to current output device 855
  list of expressions 1745
  of a list of expressions 316
  on screen 884, 885
  on screen or on the printer 344
  to current output device 854
Output device
  defining for @...SAY 461
  standard Error 1704
  standard Out 1705
OutStd() 1705
Overloading
  methods 2225
  operators 2223
OVERRIDE METHOD 2225
PACK 416
Pad() 1706
PadC() 1706
PadC() | PadL() | PadR() 1706
PadL() 1706
PadLeft() 1708
PadR() 1706
PadRight() 1709
Parameter
  formal parameter list 2195
  getting all 1079
  getting the value 1744
  number of passed 1713
  passing to code block 927
  undefined parameter list 2196
PARAMETERS 2227
Parent and child work areas 498, 829, 1678
Path
  delimiter 1245
  delimiters for multiple 1244
  delimiters for single 1242
  search path for files 495
Payment for loans 1710
Payment() 1710
PCode
  compiling 1222
  executing 1316
PCol() 1711
  resetting internal counter 1868
PCount() 1713
Periods() 1714
Persistent objects 66
Pi() 1715
PICTURE format
  for transforming values 1953
Plus operator 2058
Pointer
  to array or object 1086
  to function 1178
  to method 1233
  to string 1247
POP3 205
Popup() 103
PosAlpha() 1716
PosChar() 1718
PosDel() 1720
PosDiff() 1721
PosEqual() 1722
PosIns() 1723
Position of the record pointer
  logical 1661
  relative 1663
PosLower() 1724
PosRange() 1726
PosRepl() 1727
Postfix notation 2060, 2063
PosUpper() 1728
pragma pack() 2228
Prefix notation 2060, 2063
Present value 1743
PrgExpToVal() 1730
Printer
  available printers 1041
  check for installation 1731
  check if printer exists 1476
  column position of the print head 1711
  counter for print head position 1868
  default 1026
  getting the name from a port 1732
  raw printing 1733
  row position of the print head 1741
Printer object 279
PrinterExists() 1731
PrinterPortToName() 1732
PrintFileRaw() 1733
PrintReady() 1735
PrintSend() 1736
PrintStat() 1737
PRIVATE 2229
Procedure
  associating with a function key 479
  associating with a key 485

for the end of a program 2181
for the start of a program 2204
PROCEDURE 2231
Process
closing child process..1122
opening a child process.......................................1236
return value of..1249
ProcFile()...1738
ProcLine()...1739
ProcName()..1740
Program
setting the exit code 922
terminating with Alt+C.......................................1840
Program code
current file..1738
current line number...1739
for error handling.....................................2155, 2242
name of current function - method or procedure 1740
Program flow
branching 2173, 2202, 2240
interrupting BEGIN SEQUENCE 628
repeating a section of a program........................2175
repeating section of a program.................2191, 2193
terminating program 417
throwing an exception...1928
Program termination 417
and Alt+C 1840
Programming commands.......................................2118
PROTECTED:...2233
PRow()..1741
resetting internal counter 1868
PUBLIC...2234
Push button (text mode)...69
Pv() 1743
PValue()..1744
QOut()...1745
QOut() | QQOut() 1745
QQOut()..1745
Quarter...626, 1747
first day of..626
from Date...1747
last day of...917
Quarter() 1747
QueryRegistry() 1748
QUIT...417
Radio button (text mode)...75
Radio button group (text mode).................................82
Random number
creating float...1252
creating integer 1254
defining seed value 1256
RangeRem()..1751
RangeRepl()..1752
RAscan()...1753
RAt()...1755
Rate() 1757
RDD
current RDD 1763
default driver..1765
list available RDDs... 1761
RddInfo().. 1758
RddList() 1761
RddName()... 1763
RddRegister() 1764
RddSetDefault().. 1765
READ 418
ReadExit() 1767
ReadInsert()... 1768
ReadKey() 1769
ReadKill()... 1770
ReadModal()... 1771
ReadUpdated().. 1773
ReadVar() 1774
RECALL 420
RecCount() 1775
RecNo() 1776
Record
retrieving specific data 809
Record lock
for new records 350, 754
releasing 525, 819
releasing all 842
setting multiple 814
setting multiple in network................................ 1592
Record number 1776
Record pointer
checking if begin-of-file was reached 623
logical position 1661
logical positioning 1659
moving relative... 515, 831
navigating to the begin of file............................... 790
navigating to the end of file................................. 787
positioning 393, 788
relative position 1663
skipping with index 1679
testing the end of file 914
unique records 1681
Records
browsing in a work area 772
combining from two work areas.................. 404, 798
creating new 350, 754
creating sum 523, 840
current index value 1665
deleting all 531
deleting physically... 416
displaying in a window.. 630
exporting.. 373
importing 352
last in a work area... 1505
Length in bytes 1777
locking multiple... 814
locking multiple in network............................... 1592
locking one 1793
locking scheme 453
logical count 1655
logical order... 1676
marking as deleted...................................... 384, 771
marking as deleted in network........................... 1580

number of current 1776
reading value from field 947
recalling 420
recalling a deleted 808
recalling a deleted in network 1591
retrieving all locked 816
searching 433, 820
searching in index 1647
searching with wild cards 1683
selecting 393
selecting current 788
selecting first 790
selecting last 787
sorting 517, 835
test if locked 1472
updating 526, 844
writing values to fields 952
RECOVER 2155
RecSize() 1777
Reference parameters 2094, 2095
testing 1197
Registry
creating key/value pair 1871
existence of key value 1748
retrieving key value 1045
Regular expression 1263
comparing a string 2101
compiled 1209
compiling 1272
parsing a character string 1270
parsing a string 1266, 1277
searching and replacing 1275
searching in a character string 1091
searching in a string 2097
testing for a match 1273
REINDEX 422
Relation
clearing of 758
creating 829
creating scoped 1678
defining 498
determining the child work area 817
retrieving the linking expression 812
RELEASE 423
RemAll() 1778
RemLeft() 1779
RemRight() 1780
RENAME 425
RenameFile() 1781
Reorganizing
index file 1669
REPLACE 427
Replaceble Database Driver
configuration of 1758
Replaceble Database Drivers
available 1761
default driver 1765
Replacing
of characters in a character string 1902
soft carriage return 1070
ReplAll() 1782
Replicate() 1783
ReplLeft() 1784
ReplRight() 1785
REQUEST 2236
RestCursor() 1786
RestGets() 1787
RESTORE 429
RestScreen() 1788
RestSetKey() 1790
RestToken() 1791
RETURN 2237
RGB colors 280
Right shift 2093
Right() 1792
RLock() 1793
Round() 1795
Rounding numeric values 1795
Row() 1797
RtoD() 1798
RTrim() 1799
RUN .... 431, *see also:* HB_OpenProcess()
Runtime error 919
SAVE 432
SaveCursor() 1801
SaveGets() 1802
SaveScreen() 1803
SaveSetKey() 1805
SaveToken() 1806
SayDown() 1807
SayMoveIn() 1808
SayScreen() 1810
SaySpread() 1811
Scalar classes 2177
User defined 2151
Scope
of class members 2158
setting 1674
setting both 500
setting bottom 502
setting top 503
Screen (text mode)
clearing a region with color 696
clearing a region without color 699
clearing delayed 694
clearing one row with color 692
clearing one row without color 698
displaying saved screen contents 1788
maximum (bottom) screen row 1527
maximum (rightmost) screen column 1525
reading the contents from file 973
replacing characters 685
saving the contents 1803
setting the number of rows and columns 1859
writing the contents to file 1813
Screen (textmode)
restoring a portion 1899
saving a portion 1818

Screen cursor (text mode)
- column position 702
- row position 1797

Screen output (text mode)
- activating or suppressing 449
- additionally on the printer 496
- character for erasing 1021, 1843
- deleting display 320
- displaying a box 318, 880
- displaying a frame 347
- displaying a shadow 1298
- displaying a value 884, 885
- displaying screen buffers 883
- erasing the display 362
- initiating buffered 878
- locking for thread 1073
- number of open screen buffers 882
- releasing a lock 1074
- writing additionally to file 437

Screen output (textmode)
- effect "move in" 1808
- effect "spread" 1811
- vertical output 1807

ScreenAttr() 1812
ScreenFile() 1813
ScreenMark() 1814
ScreenMix() 1816
ScreenStr() 1818
Scroll bar (text mode) 93
Scroll lock 1499
Scroll() 1819
Scrollfixed() 1821

Scrolling
- screen in text mode 1819, 1821

Search
- an array 1753

Searching
- an array 573
- file 958
- in indexed database 433, 820, 1647
- multiple files 974
- resuming in database 368
- testing for success 995
- value in another 2099

Seconds 1823
- converting to days 751
- converting to time string 1958
- converting to time string with 1/100 seconds 1827

Seconds() 1823
SecondsCpu() 1824
SecondsSleep() 1825
Secs() 1826
SecToTime() 1827

see also
- SET OPTIMIZE 507

SEEK ... 433, *see also:* SET SOFTSEEK, *see also:* SET SCOPE
SELECT 435
Select() 1828
Self object 1250
Send operator 2070

Sequential search
- beginning 410
- resuming 368

Serialization 1283
- calulating memory requirements 1135
- of code blocks 1282
- simple data types 1285

SET ALTERNATE 437
SET AUTOPEN 439
SET AUTORDER 441
SET AUTOSHARE 442
SET BACKGROUND TASKS 443
SET BACKGROUNDTICK 444
SET BELL 445
SET CENTURY 446
SET COLOR 447
SET CONFIRM 448
SET CONSOLE 449
SET CURSOR 450
SET DATE 451
SET DBFLOCKSCHEME 453
SET DECIMALS 455
SET DEFAULT 456
SET DELETED 457
SET DELIMITERS 459
SET DESCENDING 460
SET DEVICE 461
SET DIRCASE 463
SET DIRSEPARATOR 464
SET EOL 465
SET EPOCH 466
SET ERRORLOG 467
SET ERRORLOOP 468
SET ESCAPE 469
SET EVENTMASK 470
SET EXACT 472
SET EXCLUSIVE 474
SET FILECASE 475
SET FILTER .... 476, *see also:* SET OPTIMIZE
SET FIXED 478
SET FUNCTION 479
SET HARDCOMMIT 481
SET INDEX 482
SET INTENSITY 484
SET KEY 485
SET LOG STYLE 486
SET MARGIN 488
SET MEMOBLOCK 489
SET MESSAGE 491
SET OPTIMIZE 492
SET ORDER 493
SET PATH 495
SET PRINTER 496
SET RELATION 498
SET SCOPE 500
SET SCOPEBOTTOM 502
SET SCOPETOP 503

SET SCOREBOARD 504
SET SOFTSEEK 505
SET STRICTREAD 507
SET TIME 508
SET TRACE 510
SET TYPEAHEAD 511
SET UNIQUE 512
SET VIDEOMODE 513
SET WRAP 514
Set() 1829
SetAtLike() 1836
SetBit() 1838
SetBlink() 1839
SetCancel() 1840
SetClearA() 1842
SetClearB() 1843
SetClrPair() 1844
SetColor() 1845
SetCursor() 1847
SetDate() 1849
SetErrorMode() 1850
SetFAttr() 1851
SetFCreate() 1852
SetFDaTi() 1853
SetKey() 1854
SetLastError() 1857
SetLastKey() 1858
SetMode() 1859
SetMouse() 1861
SetNetDelay() 1862
SetNetMsgColor() 1863
SetNewDate() 1864
SetNewTime() 1865
SetPos() 1866
SetPosBS() 1867
SetPrc() 1868
SetPrec() 1870
SetRegistry() 1871
SetTime() 1873
Settings
  changing of system settings 1829
  colors in text mode 447, 1845
  filtering deleted records 457
  for date values 451, 508
  for numeric values 455
  for RDDs 1758
  for records 809
  for work areas 791
Shift operator
  left 2075
  right 2093
ShowTime() 1874
Sign() 1876
Sin() 1877
Sine 1877
  hyperbolic 1878
SinH() 1878
SKIP 515
Skipping 831
  using index 1679
Smaller value of two values 1551
SMTP 210
Sockets
  alias names 1427
  bound UDP socket 1419
  clearing memory 1405
  client connection 1412
  client side socket 1410
  closing connection 1409
  error description 1426
  incoming connection 1403
  initializing 1432
  last error 1425
  port number 1433
  querying callback 1430
  querying IP address 1428
  querying timout 1431
  raw socket 1415
  reading all data 1436
  reading data 1434
  reading one block of data 1437
  reading one line of data 1439
  reset last error 1406
  sending all data 1442
  sending data 1440
  server side socket 1443
  setting a timeout value 1449
  setting callback 1447
  test for incoming data 1417
  transferred bytes 1413
  unbound UDP socket 1418
  voiding callback 1407
  voiding timeout 1408
Soft carriage return 1531
SORT 517
Sorting
  database logically 395, 1641
  of arrays 580
  of records 517, 835
SoundEx() 1879
Space() 1880
Speaker tone 1951
Sqrt() 1881
Square root 1881
Standard devices 1014
Standard() 1882
StartThread() 1883
STATIC 2238
STATIC FUNCTION 2195
STATIC PROCEDURE 2231
stdout .... *see:* #stdout
StoD() 1886
StopThread() 1887
Storage space
  available on disk 874
STORE 519
StoT() 1888
Str() 1889

Strdel() 1891
StrDiff() 1892
Stream
  creating reader 971
  creating writer 985
StrFile() 1894
StringToLiteral() 1896
StrPeek() 1897
StrPoke() 1898
StrScreen() 1899
StrSwap() 1900
StrToHex() 1901
StrTran() 1902
Structural index
  automatic opening 439
Structure extended file 763
  creating and opening 379
Structures
  abstract class 4
  byte alignment 2228
  creation 2144
  declaration 2244
StrZero() 1904
Stuff() 1906
Subscribe() 1908
SubscribeNow() 1910
SubStr() 1911
Substring
  extracting 1911
  extracting from the left 1506
  extracting from the right 1792
Substring operator 2045
Subtraction 2061
SUM 520
SWITCH 2240
SX_Decrypt() 1913
SX_DtoP() 1914
SX_Encrypt() 1915
SX_FCompress() 1916
SX_FDecompress() 1917
SX_PtoD() 1918
Symbolic constants 2125
Symbolic name
  declaring for a PRG file 2147
  declaring for external function 2188, 2236
System date 748
  setting from Date value 1849
  setting from numeric values 1864
System log file
  adding a new entry 1305
  closing 1304
  opening 1306
System settings
  changing 1829
System time 1823
  as formatted character string 1932
  setting from numeric values 1865
  setting from time string 1873
TabExpand() 1919
TabPack() 1920
TAG name of an index 1670
Tan() 1921
Tangent 1921
  hyperbolic 1922
TanH() 1922
TBColumn object 113
TBColumn() 113
TBMouse() 1923
TBrowse object 118
  for databases 1924
  mouse click 1923
TBrowse() 118
  function for :skipBlock 833
TBrowseDB() 1924
TBrowseNew() 1926
Temporary file 1176
Testing
  the data type 1961, 1987
Text
  displaying and editing (text mode) 1531
  extracting single lines 1538
  line count 1614
  longest line 1526
  number of lines 1553
  position of a character 1555
  position of a line 1558
  replacing of carriage returns 1544
  row and column position of a character 1562
  scanning lines 1260
  writing to a file 1545
TEXT 522
Text files
  currently selected 1173
  loading into array 997
  loading into array (optimized) 999
  moving record pointer 1175
  navigating 1161
  navigating to the begin-of-file 1162
  navigating to the end-of-file 1160
  number of characters 935
  number of lines 1164
  opening in a text file area 1179
  parsing a single line 1000
  reading current line 1171
  reading from disk 1541
  reading with navigation 1168
  record pointer 1172
  status information 1163
  testing end-of-file 1159
  word count 1013
Thread
  check if it is running 1483
  comparing thread handles 1479
  handle of current 1025
  killing all from outside 1494
  killing one from outside 1495
  putting to sleep 1927
  resume a single 1599

- resume all 1600
- starting new 1883
- stopping from outside 1887
- subscribe to a Mutex 1908
- subscribe to a Mutex immediately 1910
- suspending for seconds 1825
- synchronizing with second thread 1486
- system ID 1049
- using mutex 1226
- waiting for all 1994
- xHarbour ID 1050

ThreadSleep() 1927
Throw() 1928, 2242
THtmlCleanup() 1929
THtmlDocument() 152
THtmlInit() 1930
THtmlIsValid() 1931
THtmlIterator() 159
THtmlIteratorRegEx() 162
THtmlIteratorScan() 164
THtmlNode() 166
Time 1932
- am/pm formatting 567
- converting to seconds 1826
- defining a delay 1550
- defining a wait period 1995
- difference 910
- seconds elapsed since midnight 1823
- seconds since midnight 1933
- validating string 1934

Time format
- defining country-specific 508

Time() 1932
TimeToSec() 1933
TimeValid() 1934
TIpClient() 186
TIpClientFtp() 193
TIpClientHttp() 202
TIpClientPop() 205
TIpClientSmtp() 210
TIpMail() 216
Token() 1935
TokenAt() 1937
TokenEnd() 1938
TokenExit() 1939
TokenInit() 1940
Tokenizer 1089
- initializing environment 1940
- lower case tokens 1944
- position of current token 1947
- position of token 1937
- releasing memory resources 1939
- restoring environment 1791
- retrieving delimiters 1948
- retrieving next token 1946
- retrieving token 1935
- saving environment 1806
- testing for remaining tokens 1938
- upper case tokens 1949

TokenLower() 1944
TokenNext() 1946
TokenNum() 1947
TokenSep() 1948
TokenUpper() 1949
Tone() 1951
TopBarMenu() 141
TOTAL 523
Trace file 510
- adding an entry 1952

TraceLog() 1952
Transfer data 782, 783
Transform() 1953
translate .... *see:* #translate
Trigonometric functions
- getting precision 1039
- setting precision 1870

Trim() 1799, 1955
TrueName() 1957
TRY...CATCH 2242
TStream() 227
TStreamFileReader() 229
TStreamFileWriter() 234
TString() 1958
TtoC() 1959
TtoS() 1960
TUrl() 239
TXmlDocument() 245
TXmlIterator() 255
TXmlIteratorRegEx() 258
TXmlIteratorScan() 261
TXmlNode() 264
Type() 1961
typedef struct 2244
U2bin() 1963
UDP socket
- bound 1419
- receiving data 1420
- sending data 1421
- unbound 1418

UNIQUE 1650
UNIQUE index 1650
UNLOCK 525
Unselected() 1965
UntextWin() 1966
UPDATE 526
Updated() 1968
Updating
- of records 526, 844

Upper case
- converting to lower case 1516

Upper() 1969
URL object 239
USE 528
Used() 1970
User-defined
- class 2157
- command 2118
- function 2195

method 2217
User-defined RDD
attaching data to a work area 1971
attaching data to the RDD 1974
registration in RDD system 1764
result of last operation 1972
retrieving the ID 1973
setting Begin-of-file flag 1975
setting Bottom scope 1976
setting End-of-file flag 1977
setting Found flag 1978
setting Top scope 1979
UsrRdd_AreaData() 1971
UsrRdd_AreaResult() 1972
UsrRdd_ID() 1973
UsrRdd_RddData() 1974
UsrRdd_SetBof() 1975
UsrRdd_SetBottom() 1976
UsrRdd_SetEof() 1977
UsrRdd_SetFound() 1978
UsrRdd_SetTop() 1979
UUDecode
a file 1312
a string 1311
UUEncode
a file 1314
a string 1313
Val() 1980
ValPos() 1982
ValToPrg() 1983
ValToPrgExp() 1985
Valtype() 1987
VAR 2247
Version
operating system 1702
Version() 1989
Virtual method 2222
Visibility attribute for classes
EXPORTED: 2183
HIDDEN: 2200
PROTECTED: 2233
VolSerial() 1990
Volume
getting label 1052
serial number 1990
setting label 1991
Volume() 1991
W2bin() 1992
WAClose() 1993
WAIT 530
WaitForThreads() 1994
Waiting
until a key is pressed 530
WaitPeriod() 1995
WBoard() 1996
WBox() 1997
WCenter() 1999
WClose() 2000
WCol() 2001
Week() 2002
WfCol() 2003
WfLastCol() 2005
WfLastRow() 2007
WFormat() 2009
WfRow() 2010
WHILE condition
and index creation 1636
for index creation 395
for sequential search 410
in database processing 777
Wild card
converting to Regular Expression 2012
searching 1683
Wild2RegEx() 2012
WildMatch() 2014
Win32Bmp() 276
Win32Prn() 279
Window (text mode)
all open windows 2040
bottom row position 2019
bottom row position (formatted) 2007
closing 2000
closing all 1993
coordinates 2016
creating new window 2025
displaying centered 1999
drawing a frame 1997
erasing text only 1966
highest window ID 2023
left column position 2001
left column position (formatted) 2003
moving 2021
moving mode 2020
moving mouse cursor 2022
restricting screen area 1996
right column position 2018
right column position (formatted) 2005
selecting current 2036
setting mouse cursor 2037
shadow color 2039
step size for moving 2041
toggle move setting 2038
top row position 2035
top row position (formatted) 2010
usable area (formatted) 2009
Windows GDI printing 279
Windows registry .... *see:* Registry
WInfo() 2016
WITH OBJECT 2249
changing the With object 1296
nesting level 1318
replacing the With object 1279
WLastCol() 2018
WLastRow() 2019
WMode() 2020
WMove() 2021
WMSetPos() 2022
WNum() 2023

WoM() ... 2024
WOpen() ... 2025
Word
  high byte ... 1613
  low byte ... 1615
Word() ... 2028
WordOne() ... 2029
WordOnly() ... 2030
WordRem() ... 2031
WordRepl() ... 2032
WordSwap() ... 2033
WordToChar() ... 2034
Work area
  activating indexes ... 1666
  closing all files ... 359, 759
  closing file ... 364, 760
  closing index files ... 1668
  closing index files (compatibility) ... 757
  determining the alias name ... 563
  determining the field structure ... 553
  evaluating a code block for each record ... 777
  number of open indexes ... 1639
  number of records ... 1775
  opening database file ... 845
  opening database file shared ... 1578
  opening file ... 528
  retrieving configuration data ... 791
  retrieving number by alias name ... 1828
  selecting ... 435
  selecting current ... 822
  setting a scope ... 1674
  testing if a file is open ... 1970
  writing all file buffers ... 367
  writing file buffers into the files ... 761
WRow() ... 2035
WSelect() ... 2036
WSetMouse() ... 2037
WSetMove() ... 2038
WSetShadow() ... 2039
WStack() ... 2040
WStep() ... 2041
xHarbour
  C compiler used for build ... 1128
  currrent PCode version ... 1246
  version ... 1989
XML
  error description ... 1321
  node in document ... 264
  open document ... 245
  searching document with RegEx ... 258
  searching node in document ... 261
  traversing document ... 255
XtoC() ... 2042
Year
  checking for leap year ... 1471
  displaying two or four digits ... 446, 728
  extracting from date value ... 2043
  first day of ... 627
  last day of ... 918
Year() ... 2043
ZAP ... 531
ZIP
  compressing string ... 1129
  error code ... 1132
  error description ... 1133
  preallocate buffer ... 1131
  uncompressing string ... 1309