MapBasic

11.0

REFERENCE

Pitney Bowes
Business Insight

Information in this document is subject to change without notice and does not represent a commitment on the part of the vendor or its representatives. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, without the written permission of Pitney Bowes Software Inc., One Global View, Troy, New York 12180-8399.

© 2011 Pitney Bowes Software Inc. All rights reserved. MapInfo, Group 1 Software, and **MapBasic** are trademarks of Pitney Bowes Software Inc. All other marks and trademarks are property of their respective holders.

United States:
Phone: 518.285.6000
Fax: 518.285.6070
Sales: 800.327.8627
Government Sales: 800.619.2333
Technical Support: 518.285.7283
Technical Support Fax: 518.285.6080
pbinsight.com

Canada:
Phone: 416.594.5200
Fax: 416.594.5201
Sales: 800.268.3282
Technical Support:.518.285.7283
Technical Support Fax: 518.285.6080
pbinsight.ca

Europe/United Kingdom:
Phone: +44.1753.848.200
Fax: +44.1753.621.140
Technical Support: +44.1753.848.229
pbinsight.co.uk

Asia Pacific/Australia:
Phone: +61.2.9437.6255
Fax: +61.2.9439.1773
Technical Support: 1.800.648.899
pbinsight.com.au

Contact information for all Pitney Bowes Software Inc. offices is located at: **http://www.pbinsight.com/about/contact-us**.

© 2011 Adobe Systems Incorporated. All rights reserved. Adobe, the Adobe logo, Acrobat and the Adobe PDF logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries.

© 2011 OpenStreetMap contributors, CC-BY-SA; see OpenStreetMap **http://www.openstreetmap.org** and CC-BY-SA **http://creativecommons.org/licenses/by-sa/2.0**

libtiff © 1988-1995 Sam Leffler, © 2011 Silicon Graphics International. All Rights Reserved.

libgeotiff © 2011 Niles D. Ritter.

Amigo, Portions © 1999 3D Graphics, Inc. All Rights Reserved.

Halo Image Library © 1993 Media Cybernetics Inc. All Rights Reserved

Portions thereof LEAD Technologies, Inc. © 1991-2011. All Rights Reserved.

Portions © 1993-2011 Ken Martin, Will Schroeder, Bill Lorensen. All Rights Reserved.

ECW by ERDAS © 1993-2011 ERDAS Inc. and/or its suppliers. All rights reserved.

Portions © 2011 ERDAS Inc. All Rights Reserved.

MrSID, MrSID Decompressor and the MrSID logo are trademarks of LizardTech, A Celartem Company. used under license. Portions of this computer program are copyright © 1995-1998 LizardTech, A Celartem Company, and/or the university of California or are protected by US patent nos. 5,710,835 or 5,467,110 and are used under license. All rights reserved. MrSID is protected under US and international patent & copyright treaties and foreign patent applications are pending. Unauthorized use or duplication prohibited.

Contains FME® Objects © 2005-2011 Safe Software Inc., All Rights Reserved.

© 2011 SAP AG, All Rights Reserved. Crystal Reports® and Business Objects™ are the trademark(s) or registered trademark(s) of SAP AG in Germany and in several other countries.

Amyuni PDF Converter © 2000-2011, AMYUNI Consultants – AMYUNI Technologies. All rights reserved.

Civic England - Public Sector Symbols Copyright © 2011 West London Alliance. The symbols may be used free of charge. For more information on these symbols, including how to obtain them for use in other applications, please visit the West London Alliance Web site at **http://www.westlondonalliance.org/**

© 2006-2011 Tele Atlas. All Rights Reserved. This material is proprietary and the subject of copyright protection and other intellectual property rights owned or licensed to Tele Atlas. The use of this material is subject to the terms of a license agreement. You will be held liable for any unauthorized copying or disclosure of this material.

Microsoft Bing: All contents of the Bing service are Copyright © 2011 Microsoft Corporation and/or its suppliers, One Microsoft Way, Redmond, WA 98052, USA. All rights reserved. Microsoft or its suppliers own the title, copyright, and other intellectual property rights in the Bing service and content. Microsoft, Windows, Windows Live, Windows logo, MSN, MSN logo (butterfly), Bing, and other Microsoft products and services may also be either trademarks or registered trademarks of Microsoft in the United States and/or other countries.

This product contains 7-Zip, which is licensed under GNU Lesser General Public License, Version 3, 29 June 2007 with the unRAR restriction. The license can be downloaded from **http://www.7-zip.org/license.txt**. The source code is available from **http://www.7-zip.org**.

Products named herein may be trademarks of their respective manufacturers and are hereby recognized. Trademarked names are used editorially, to the benefit of the trademark owner, with no intent to infringe on the trademark.

May 17, 2011

Table of Contents

Chapter 1: Introduction to MapBasic 19

Type Conventions 20
Language Overview 20
MapBasic Fundamentals 20
Variables 20
Looping and Branching 21
Output and Printing 21
Procedures (Main and Subs) 21
Error Handling 21
Functions 22
Custom Functions 22
Data-Conversion Functions 22
Date and Time Functions 22
Math Functions 23
String Functions 23
Working With Tables 24
Creating and Modifying Tables 24
Querying Tables 25
Working With Remote Data 25
Working With Files (Other Than Tables) 26
File Input/Output 26
File and Directory Names 27
Working With Maps and Graphical Objects 27
Creating Map Objects 27
Modifying Map Objects 27
Querying Map Objects 28
Working With Object Styles 29
Working With Map Windows 29
Creating the User Interface 30
ButtonPads (ToolBars) 30
Dialog Boxes 30

Menus 30
Windows. 31
System Event Handlers 31
Communicating With Other Applications 32
DDE (Dynamic Data Exchange; Windows Only) 32
Integrated Mapping 32
Special Statements and Functions 32
Chapter 2: New and Enhanced MapBasic Statements and Functions 34
New MapBasic Functions and Statements 35
Enhancements to MapBasic Functions and Statements. 35
Chapter 3: A – Z MapBasic Language Reference 38
Abs( ) function 39
Acos( ) function 39
Add Cartographic Frame statement 40
Add Column statement 42
Add Map statement 48
Alter Button statement 51
Alter ButtonPad statement 52
Alter Cartographic Frame statement 57
Alter Control statement 58
Alter MapInfoDialog statement 60
Alter Menu statement 62
Alter Menu Bar statement 67
Alter Menu Item statement 68
Alter Object statement 70
Alter Table statement 76
ApplicationDirectory$( ) function 78
ApplicationName$( ) function 79
Area( ) function 79
AreaOverlap( ) function 80
Asc( ) function 81
Asin( ) function 82
Ask( ) function 83
Atn( ) function 84
AutoLabel statement. 85
Beep statement 86
Browse statement 86
BrowserInfo function 88
Brush clause 89
Buffer( ) function 92

ButtonPadInfo( ) function 93
Call statement 94
CartesianArea( ) function 96
CartesianBuffer( ) function 97
CartesianConnectObjects( ) function 98
CartesianDistance( ) function 99
CartesianObjectDistance( ) function 100
CartesianObjectLen( ) function 100
CartesianOffset( ) function 101
CartesianOffsetXY( ) function 102
CartesianPerimeter( ) function 103
Centroid( ) function 104
CentroidX( ) function 105
CentroidY( ) function 106
CharSet clause 107
ChooseProjection$( ) function 110
Chr$( ) function 111
Close All statement 112
Close Connection statement 112
Close File statement 113
Close Table statement 114
Close Window statement 115
ColumnInfo( ) function 116
Combine( ) function 118
CommandInfo( ) function 119
Commit Table statement 124
ConnectObjects( ) function 129
Continue statement 130
Control Button / OKButton / CancelButton clause 132
Control CheckBox clause 133
Control DocumentWindow clause 134
Control EditText clause 135
Control GroupBox clause 136
Control ListBox / MultiListBox clause 137
Control PenPicker/BrushPicker/SymbolPicker/FontPicker clause 140
ControlPointInfo() function 141
Control PopupMenu clause 142
Control RadioGroup clause 143
Control StaticText clause 144
ConvertToPline( ) function 145
ConvertToRegion( ) function 146

ConvexHull( ) function **146**
CoordSys clause **147**
CoordSys Earth and NonEarth Projection 147
CoordSys Layout Units 150
CoordSys Table 150
CoordSys Window 151
CoordSysName$( ) function **151**
CoordSysStringToEPSG( ) function **152**
CoordSysStringToPRJ$( ) function **153**
CoordSysStringToWKT$( ) function **153**
Cos( ) function **154**
Create Adornment statement **155**
Create Arc statement **159**
Create ButtonPad statement **160**
Create ButtonPad As Default statement **164**
Create ButtonPads As Default statement **165**
Create Cartographic Legend statement **165**
CreateCircle( ) function **169**
Create Collection statement **171**
Create Cutter statement **172**
Create Ellipse statement **173**
Create Frame statement **174**
Create Grid statement **176**
Create Index statement **179**
Create Legend statement **179**
CreateLine( ) function **180**
Create Line statement **181**
Create Map statement **182**
Create Map3D statement **183**
Create Menu statement **185**
Create Menu Bar statement **190**
Create MultiPoint statement **192**
Create Object statement **194**
Create Pline statement **199**
CreatePoint( ) function **200**
Create Point statement **201**
Create PrismMap statement **202**
Create Ranges statement **204**
Create Rect statement **207**
Create Redistricter statement **208**
Create Region statement **209**

Create Report From Table statement 211
Create RoundRect statement 212
Create Styles statement 213
Create Table statement 214
CreateText( ) function 219
Create Text statement 221
CurDate( ) function 222
CurDateTime() function 223
CurrentBorderPen( ) function 223
CurrentBrush( ) function 224
CurrentFont( ) function 225
CurrentLinePen( ) function 226
CurrentPen( ) function 226
CurrentSymbol( ) function 227
CurTime() function 228
DateWindow( ) function 229
Day( ) function 229
DDEExecute statement 230
DDEInitiate( ) function 231
DDEPoke statement 234
DDERequest$( ) function 236
DDETerminate statement 238
DDETerminateAll statement 239
Declare Function statement 239
Declare Method statement 241
Declare Sub statement 244
Define statement 246
DeformatNumber$( ) function 247
Delete statement 248
Dialog statement 249
Dialog Preserve statement 255
Dialog Remove statement 256
Dim statement 257
Distance( ) function 261
Do Case...End Case statement 262
Do...Loop statement 264
Drop Index statement 266
Drop Map statement 266
Drop Table statement 267
End MapInfo statement 268
End Program statement 269

EndHandler procedure 270
EOF( ) function 270
EOT( ) function 271
EPSGToCoordSysString$( ) function 272
Erase( ) function 273
Err( ) function. 274
Error statement 275
Error$( ) function. 275
Exit Do statement 276
Exit For statement. 277
Exit Function statement 277
Exit Sub statement 278
Exp( ) function 279
Export statement. 279
ExtractNodes( ) function. 282
Farthest statement 283
Fetch statement. 286
FileAttr( ) function 288
FileExists( ) function. 289
FileOpenDlg( ) function. 289
FileSaveAsDlg( ) function. 291
Find statement. 292
Find Using statement 296
Fix( ) function. 297
Font clause 298
For…Next statement. 300
ForegroundTaskSwitchHandler procedure 302
Format$( ) function 303
FormatDate$( ) function 305
FormatNumber$( ) function 306
FormatTime$ function 307
FME Refresh Table statement 308
FrontWindow( ) function 309
Function…End Function statement. 309
Geocode statement. 312
GeocodeInfo( ) function 317
Get statement 320
GetCurrentPath( ) function. 322
GetDate( ) function 323
GetFolderPath$( ) function 323
GetGridCellValue( ) function 324

GetMetadata$( ) function 325
GetPreferencePath$( ) function 326
GetSeamlessSheet( ) function 327
GetTime() function 328
Global statement 328
Goto statement 329
Graph statement 330
GridTableInfo( ) function 331
GroupLayerInfo function 332
HomeDirectory$( ) function 334
HotlinkInfo function 334
Hour function 335
If…Then statement 336
Import statement 337
Include statement 343
Input # statement. 343
Insert statement. 344
InStr( ) function 346
Int( ) function 347
IntersectNodes( ) function 348
IsGridCellNull( ) function 348
IsogramInfo( ) function 349
IsPenWidthPixels( ) function 352
Kill statement. 352
LabelFindByID( ) function. 353
LabelFindFirst( ) function 354
LabelFindNext( ) function 355
LabelInfo( ) function 356
LabelOverrideInfo( ) function 359
LayerControlInfo( ) function. 363
LayerControlSelectionInfo( ) function. 363
LayerInfo( ) function 364
LayerListInfo function. 372
LayerStyleInfo( ) function 373
Layout statement. 374
LCase$( ) function 375
Left$( ) function 376
LegendFrameInfo( ) function 377
LegendInfo( ) function 378
LegendStyleInfo( ) function 379
Len( ) function 380

LibraryServiceInfo( ) function 381
Like( ) function. 382
Line Input statement 383
LocateFile$( ) function 384
LOF( ) function. 386
Log( ) function 386
LTrim$( ) function 387
Main procedure 388
MakeBrush( ) function 389
MakeCustomSymbol( ) function 390
MakeDateTime function 391
MakeFont( ) function 392
MakeFontSymbol( ) function 393
MakePen( ) function 394
MakeSymbol( ) function 395
Map statement 396
Map3DInfo( ) function 399
MapperInfo( ) function 402
Maximum( ) function 407
MBR( ) function 408
Menu Bar statement 409
MenuItemInfoByHandler( ) function 409
MenuItemInfoByID( ) function 411
Metadata statement. 412
MGRSToPoint( ) function 415
Mid$( ) function 416
MidByte$( ) function 417
Minimum( ) function 417
Minute function 418
Month( ) function 419
Nearest statement 420
Note statement 423
NumAllWindows( ) function 423
NumberToDate( ) function 424
NumberToDateTime function 425
NumberToTime function. 425
NumCols( ) function 426
NumTables( ) function 427
NumWindows( ) function 427
ObjectDistance( ) function 431
ObjectGeography( ) function 431

ObjectInfo( ) function 433
ObjectLen( ) function 438
ObjectNodeHasM( ) function 439
ObjectNodeHasZ( ) function 440
ObjectNodeM( ) function 441
ObjectNodeX( ) function 443
ObjectNodeY( ) function 444
ObjectNodeZ( ) function 445
Objects Check statement 446
Objects Clean statement 447
Objects Combine statement 449
Objects Disaggregate statement 450
Objects Enclose statement 452
Objects Erase statement 453
Objects Intersect statement 455
Objects Move statement 457
Objects Offset statement 458
Objects Overlay statement 459
Objects Pline statement 460
Objects Snap statement 461
Objects Split statement 463
Offset( ) function 465
OffsetXY( ) function 466
OnError statement 467
Open Connection statement 469
Open File statement 470
Open Report statement 472
Open Table statement 473
Open Window statement 475
Overlap( ) function 476
OverlayNodes( ) function 477
Pack Table statement 478
PathToDirectory$( ) function 479
PathToFileName$( ) function 480
PathToTableName$( ) function 480
Pen clause 481
PenWidthToPoints( ) function 484
Perimeter( ) function 485
PointsToPenWidth( ) function 486
PointToMGRS$( ) function 487
PointToUSNG$(obj, datumid) 488

Print statement **490**
Print # statement **491**
PrintWin statement **492**
PrismMapInfo( ) function **493**
ProgramDirectory$( ) function **496**
ProgressBar statement **496**
Proper$( ) function **499**
ProportionOverlap( ) function **499**
Put statement **500**
Randomize statement **501**
RasterTableInfo( ) function **502**
RegionInfo( ) function **504**
ReadControlValue( ) function **504**
ReDim statement **507**
Register Table statement **508**
Supporting Transaction Capabilities for WFS Layers 515
Relief Shade statement **515**
Reload Symbols statement **516**
RemoteMapGenHandler procedure **516**
RemoteMsgHandler procedure **517**
RemoteQueryHandler( ) function **518**
Remove Cartographic Frame statement **519**
Remove Map statement **520**
Rename File statement **521**
Rename Table statement **522**
Reproject statement **523**
Resume statement **523**
RGB( ) function **524**
Right$( ) function **525**
Rnd( ) function **526**
Rollback statement **527**
Rotate( ) function **528**
RotateAtPoint( ) function **529**
Round( ) function **530**
RTrim$( ) function **531**
Run Application statement **531**
Run Command statement **532**
Run Menu Command statement **534**
Run Program statement **536**
Save File statement **538**
Save MWS statement **538**

Save Window statement 540
Save Workspace statement 542
SearchInfo( ) function 543
SearchPoint( ) function 546
SearchRect( ) function 547
Second function 548
Seek( ) function 548
Seek statement 549
SelChangedHandler procedure 549
Select statement 551
SelectionInfo( ) function 560
Server Begin Transaction statement 561
Server Bind Column statement 562
Server Close statement 563
Server_ColumnInfo( ) function 564
Server Commit statement 566
Server_Connect( ) function 567
Server_ConnectInfo( ) function 575
Server Create Map statement 576
Server Create Style statement 578
Server Create Table statement 579
Server Create Workspace statement 581
Server Disconnect statement 582
Server_DriverInfo( ) function 583
Server_EOT( ) function 584
Server_Execute( ) function 585
Server Fetch statement 586
Server_GetODBCHConn( ) function 588
Server_GetODBCHStmt( ) function 589
Server Link Table statement 590
Server_NumCols( ) function 592
Server_NumDrivers( ) function 593
Server Refresh statement 594
Server Remove Workspace statement 595
Server Rollback statement 595
Server Set Map statement 596
Server Versioning statement 597
Server Workspace Merge statement 599
Server Workspace Refresh statement 601
SessionInfo( ) function 603
Set Adornment statement 604

Set Application Window statement **607**
Set Area Units statement **608**
Set Browse statement **609**
Set Buffer Version statement **610**
Set Cartographic Legend statement **611**
Set Combine Version statement **612**
Set Command Info statement **613**
Set Connection Geocode statement **614**
Set Connection Isogram statement **617**
Set CoordSys statement **619**
Set Date Window statement **620**
Set Datum Transform Version statement **621**
Set Digitizer statement **622**
Set Distance Units statement **624**
Set Drag Threshold statement **625**
Set Event Processing statement **625**
Set File Timeout statement **626**
Set Format statement **627**
Set Graph statement **628**
Set Handler statement **633**
Set Layout statement **634**
Set Legend statement **636**
Set LibraryServiceInfo statement **639**
Set Map statement **641**
Changing the Behavior of the Entire Map 642
Changing the Current View of the Map 644
Managing Individual Layer Properties and Appearance 646
Settings That Have a Permanent Effect on a Map Layer 649
Managing Individual Label Properties 650
Adding Style Overrides to a Layer 654
Modifying Style Overrides for a Layer 657
Enabling, Disabling, or Removing Overrides for a Layer 658
Adding Overrides for Layer Labels 659
Modifying Layer Label Overrides 662
Enabling, Disabling, or Removing Overrides for Layer Labels 665
Managing Group Layers 666
Ordering Layers 667
Managing the Coordinate System of the Map 669
Managing Image Properties 670
Managing Hotlinks 671
Adding New HotLink Definitions 674

Modifying Existing HotLink Definitions 675
Removing HotLink Definitions 676
Reordering HotLink Definitions 676
Set Map3D statement 677
Set Next Document statement 679
Set Paper Units statement 680
Set Path statement 681
Set PrismMap statement. 682
Set ProgressBars statement 684
Set Redistricter statement 684
Set Resolution statement 686
Set Shade statement 687
Set Style statement 688
Set Table Datum statement 689
Set Table statement 689
Set Target statement. 692
Set Window statement 692
Sgn( ) function 703
Shade statement 704
Shading by Ranges of Values 704
Shading by Individual Values 708
Dot Density. 710
Graduated Symbols 711
Pie Charts 712
Bar Charts 714
Sin( ) function 716
Space$( ) function 717
SphericalArea( ) function 718
SphericalConnectObjects( ) function 719
SphericalDistance( ) function. 719
SphericalObjectDistance( ) function 720
SphericalObjectLen( ) function 721
SphericalOffset( ) function 722
SphericalOffsetXY( ) function 723
SphericalPerimeter( ) function 724
Sqr( ) function 725
StatusBar statement 726
Stop statement 727
Str$( ) function. 728
String$( ) function 729
StringCompare( ) function 730

StringCompareIntl( ) function 731
StringToDate( ) function 731
StringToDateTime function 733
StringToTime function 733
StyleAttr( ) function 734
StyleOverrideInfo( ) function 736
Sub…End Sub statement 739
Symbol clause 740
SystemInfo( ) function 744
TableInfo( ) function 746
TableListInfo( ) function 751
TableListSelectionInfo( ) function 753
Tan( ) function 753
TempFileName$( ) function 754
Terminate Application statement. 755
TextSize( ) function 756
Time( ) function 756
Timer( ) function 757
ToolHandler procedure. 757
TriggerControl( ) function. 759
TrueFileName$( ) function 760
Type statement 761
UBound( ) function 762
UCase$( ) function. 763
UnDim statement. 763
UnitAbbr$( ) function 764
UnitName$( ) function 765
Unlink statement 766
Update statement 766
Update Window statement 767
URL clause. 768
USNGToPoint(string) 769
Val( ) function 770
Weekday( ) function 771
WFS Refresh Table statement 772
While…Wend statement 772
WinChangedHandler procedure 774
WinClosedHandler procedure 775
WindowID( ) function 776
WindowInfo( ) function 777
WinFocusChangedHandler procedure 784

Write # statement 785
Year( ) function 786
MICloseContent( ) procedure 794
MICloseFtpConnection( ) procedure 794
MICloseFtpFileFind( ) procedure 794
MICloseHttpConnection( ) procedure 795
MICloseHttpFile( ) procedure 795
MICloseSession( ) procedure 796
MICreateSession( ) function 796
MICreateSessionFull( ) function 797
MIErrorDlg( ) function 798
MIFindFtpFile( ) function 800
MIFindNextFtpFile( ) function 801
MIGetContent( ) function 801
MIGetContentBuffer( ) function 803
MIGetContentLen( ) function 803
MIGetContentString( ) function 804
MIGetContentToFile( ) function 804
MIGetContentType( ) function 805
MIGetCurrentFtpDirectory( ) function 806
MIGetErrorCode( ) function 806
MIGetErrorMessage( ) function 807
MIGetFileURL( ) function 807
MIGetFtpConnection( ) function 808
MIGetFtpFile( ) function 809
MIGetFtpFileFind( ) function 810
MIGetFtpFileName( ) procedure 811
MIGetHttpConnection( ) function 811
MIIsFtpDirectory( ) function 812
MIIsFtpDots( ) function 813
MIOpenRequest( ) function 813
MIOpenRequestFull( ) function 814
MIParseURL( ) function 816
MIPutFtpFile( ) function 817
MIQueryInfo( ) function 818
MIQueryInfoStatusCode( ) function 819
MISaveContent( ) function 820
MISendRequest( ) function 820
MISendSimpleRequest( ) function 821
MISetCurrentFtpDirectory( ) function 822
MISetSessionTimeout( ) function 822

MIXmlAttributeListDestroy( ) procedure 825
MIXmlDocumentCreate( ) function 825
MIXmlDocumentDestroy( ) procedure. 826
MIXmlDocumentGetNamespaces( ) function 826
MIXmlDocumentGetRootNode( ) function 827
MIXmlDocumentLoad( ) function. 827
MIXmlDocumentLoadXML( ) function 828
MIXmlDocumentLoadXMLString( ) function. 829
MIXmlDocumentSetProperty( ) function 830
MIXmlGetAttributeList( ) function 831
MIXmlGetChildList( ) function 831
MIXmlGetNextAttribute( ) function 832
MIXmlGetNextNode( ) function 833
MIXmlNodeDestroy( ) procedure 833
MIXmlNodeGetAttributeValue( ) function 834
MIXmlNodeGetFirstChild( ) function 834
MIXmlNodeGetName( ) function 835
MIXmlNodeGetParent( ) function. 836
MIXmlNodeGetText( ) function. 836
MIXmlNodeGetValue( ) function. 837
MIXmlNodeListDestroy( ) procedure. 837
MIXmlSCDestroy( ) procedure 838
MIXmlSCGetLength( ) function 838
MIXmlSCGetNamespace( ) function 839
MIXmlSelectNodes( ) function 839
MIXmlSelectSingleNode( ) function. 840
. 843
Numeric Operators 845
Comparison Operators 845
Logical Operators 846
Geographical Operators 846
Precedence 847
Automatic Type Conversions. 847
Wildcards 848
MapBasic.DEF File 850

1

Introduction to MapBasic

This manual describes every statement and function in the MapBasic Development Environment programming language. To learn about the concepts behind MapBasic programming, or to learn about using the MapBasic development environment, see the MapBasic *User Guide*.

Topics in this Section:

- Type Conventions 20
- Language Overview 20
- MapBasic Fundamentals 20
- Functions 22
- Working With Tables 24
- Working With Files (Other Than Tables) 26
- Working With Maps and Graphical Objects 27
- Creating the User Interface 30
- Communicating With Other Applications 32
- Special Statements and Functions 32

Type Conventions

This manual uses the following conventions to designate specific items in the text:

Convention	Meaning
If, **Call**, **Map**, **Browse**, **Area**	Bold words with the first letter capitalized are MapBasic keywords. Within this manual, the first letter of each keyword is capitalized; however, when you write MapBasic programs, you may enter keywords in upper-, lower-, or mixed-case.
Main, Pen, Object	Non-bold words with the first letter capitalized are usually special procedure names or variable types.
table, *handler*, *window_id*	Italicized words represent parameters to MapBasic statements. When you construct a MapBasic statement, you must supply an appropriate expression for each parameter.
[*window_id*], [**Interactive**]	Keywords or parameters which appear inside square brackets are optional.
{ **On** | **Off** }	When a syntax expression appears inside braces, the braces contain a list of keywords or parameters, separated by the vertical bar character (|). You must choose one of the options listed. For example, in the sample shown on the left ({ **On** | **Off** }), you should choose either **On** or **Off**.
`"Note "Hello,world!"`	Actual program samples are shown in Courier font.

Language Overview

The following pages provide an overview of the MapBasic language. Task descriptions appear on the left; corresponding statement names and function names appear on the right, in **bold**. Function names are followed by parentheses ( ).

MapBasic Fundamentals

Variables

Declare local or global variables:	**Dim**, **Global**
Resize array variables:	**ReDim**, **UBound( )**, **UnDim**
Declare custom data structure:	**Type**

Looping and Branching

Looping:	**For…Next**, **Exit For**, **Do…Loop**, **Exit Do**, **While…Wend**
Branching:	**If…Then**, **Do Case**, **GoTo**
Other flow control:	**End Program**, **Terminate Application**, **End MapInfo**

Output and Printing

Print a window's contents:	**PrintWin**
Print text to message window:	**Print**
Set up a Layout window:	**Layout**, **Create Frame**, **Set Window**
Export a window to a file:	**Save Window**
Controlling the Printer:	**Set Window**, **Window Info( )**

Procedures (Main and Subs)

Define a procedure:	**Declare Sub**, **Sub…End Sub**
Call a procedure:	**Call**
Exit a procedure:	**Exit Sub**
Main procedure:	**Main**

Error Handling

Set up an error handler:	**OnError**
Return current error information:	**Err( )**, **Error$( )**
Return from error handler:	**Resume**
Simulate an error:	**Error**

Functions

Custom Functions

Define a custom function:	**Declare Function**, **Function…End Function**
Exit a function:	**Exit Function**

Data-Conversion Functions

Convert strings to codes:	**Asc( )**
Convert codes to strings:	**Chr$( )**
Convert strings to numbers:	**Val( )**
Convert numbers to strings:	**Str$( )**, **Format$( )**
Convert a number or a string to a date:	**NumberToDate( )**, **StringToDate( )**
Converting to a 2-Digit Year:	**Set Date Window**, **DateWindow( )**
Convert object types:	**ConvertToRegion( )**, **ConvertToPline( )**
Convert labels to text:	**LabelInfo( )**
Convert a point object to a MGRS coordinate:	**PointToMGRS$( )**
Convert a MGRS coordinate to a point object:	**MGRSToPoint( )**
Convert a point object to a USNG coordinate:	**PointToUSNG$(obj, datumid)**
Convert a USNG coordinate to a point object:	**USNGToPoint(string)**

Date and Time Functions

Obtain the current date:	**CurDate( )**
Extract parts of a date:	**Day( )**, **Month( )**, **Weekday( )**, **Year( )**
Read system timer:	**Timer( )**
Convert a number or a string to a date:	**NumhmmberToDate( )**, **StringToDate( )**
Obtain the current time and date:	**CurTime(),CurDateTime()**

Obtain the date and time:	**GetDate(), GetTime()**
Convert a number to date and time:	**NumberToDateTime(), NumberToTime()**
Convert a string to date and time:	**StringToDateTime(), StringToTime()**
Convert date and time to a particular format:	**FormatDate(), FormatTime()**
Convert time to hour, minute and second:	**Hour(), Minute(), Second()**

Math Functions

Trigonometric functions:	**Cos( )**, **Sin( )**, **Tan( )**, **Acos( )**, **Asin( )**, **Atn( )**
Geographic functions:	**Area( )**, **Perimeter( )**, **Distance( )**, **ObjectLen( )**, **CartesianArea( )**, **CartesianPerimeter( )**, **CartesianDistance( )**, **CartesianObjectLen( )**, **SphericalArea( )**, **SphericalPerimeter( )**, **SphericalDistance( )**, **SphericalObjectLen( )**
Random numbers:	**Randomize**, **Rnd( )**
Sign-related functions:	**Abs( )**, **Sgn( )**
Truncating fractions:	**Fix( )**, **Int( )**, **Round( )**
Other math functions:	**Exp( )**, **Log( )**, **Minimum( )**, **Maximum( )**, **Sqr( )**

String Functions

Upper / lower case:	**UCase$( )**, **LCase$( )**, **Proper$( )**
Find a sub-string:	**InStr( )**
Extract part of a string:	**Left$( )**, **Right$( )**, **Mid$( )**, **MidByte$( )**
Trim blanks from a string:	**LTrim$( )**, **RTrim$( )**
Format numbers as strings:	**Format$( )**, **Str$( )**, **Set Format**, **FormatNumber$( )**, **DeformatNumber$( )**
Determine string length:	**Len( )**
Convert character codes:	**Chr$( )**, **Asc( )**

Compare strings:	**Like( )**, **StringCompare( )**, **StringCompareIntl( )**
Repeat a string sequence:	**Space$( )**, **String$( )**
Return unit name:	**UnitAbbr$( )**, **UnitName$( )**
Convert a point object to a MGRS coordinate:	**PointToMGRS$( )**
Convert a MGRS coordinate to a point object:	**MGRSToPoint( )**
Convert an EPSG string to a CoordSys clause:	**EPSGToCoordSysString$( )**
Convert a point object to a USNG coordinate:	**PointToUSNG$(obj, datumid)**
Convert a USNG coordinate to a point object:	**USNGToPoint(string)**

Working With Tables

Creating and Modifying Tables

Open an existing table:	**Open Table**
Close one or more tables:	**Close Table**, **Close All**
Create a new, empty table:	**Create Table**
Turn a file into a table:	**Register Table**
Import/export tables/files:	**Import**, **Export**
Modify a table's structure:	**Alter Table**, **Add Column**, **Create Index**, **Drop Index**, **Create Map**, **Drop Map**
Create a Crystal Reports file:	**Create Report From Table**
Load a Crystal Report:	**Open Report**
Add, edit, delete rows:	**Insert**, **Update**, **Delete**
Pack a table:	**Pack Table**
Control table settings:	**Set Table**
Save recent edits:	**Table**
Discard recent edits:	**Rollback**

Rename a table:	**Rename Table**
Delete a table:	**Drop Table**

Querying Tables

Position the row cursor:	**Fetch**, **EOT( )**
Select data, work with Selection:	**Select**, **SelectionInfo( )**
Find map objects by address:	**Find**, **Find Using**, **CommandInfo( )**
Find map objects at location:	**SearchPoint( )**, **SearchRect( )**, **SearchInfo( )**
Obtain table information:	**NumTables( )**, **TableInfo( )**
Obtain column information:	**NumCols( )**, **ColumnInfo( )**
Query a table's metadata:	**GetMetadata$( )**, **Metadata**
Query seamless tables:	**TableInfo( )**, **GetSeamlessSheet( )**

Working With Remote Data

Create a new table:	**Server Create Table**
Communicate with data server:	**Server_Connect( )**, **Server_ConnectInfo( )**
Begin work with remote server:	**Server Begin Transaction**
Assign local storage:	**Server Bind Column**
Obtain column information:	**Server_ColumnInfo( )**, **Server_NumCols( )**
Send an SQL statement:	**Server_Execute( )**
Position the row cursor:	**Server Fetch**, **Server_EOT( )**
Save changes:	**Server**
Discard changes:	**Server Rollback**
Free remote resources:	**Server Close**
Make remote data mappable:	**Server Create Map**
Change object styles:	**Server Set Map**

Synchronize a linked table:	**Server Refresh**
Create a linked table:	**Server Link Table**
Unlink a linked table:	**Unlink**
Disconnect from server:	**Server Disconnect**
Retrieve driver information:	**Server_DriverInfo( )**, **Server_NumDrivers( )**
Get ODBC connection handle:	**Server_GetODBCHConn( )**
Get ODBC statement handle:	**Server_GetODBCHStmt( )**
Set Object styles:	**Server Create Style**

Working With Files (Other Than Tables)

File Input/Output

Open or create a file:	**Open File**
Close a file:	**Close File**
Delete a file:	**Kill**
Rename a file:	**Rename File**
Copy a file:	**Save File**
Read from a file:	**Get**, **Seek**, **Input #**, **Line Input**
Write to a file:	**Put**, **Print #**, **Write #**
Determine file's status:	**EOF( )**, **LOF( )**, **Seek( )**, **FileAttr( )**, **FileExists( )**
Turn a file into a table:	**Register Table**
Retry on sharing error:	**Set File Timeout**

File and Directory Names

Return system directories:	**ProgramDirectory$( )**, **HomeDirectory$( )**, **ApplicationDirectory$( )**
Extract part of a filename:	**PathToTableName$( )**, **PathToDirectory$( )**, **PathToFileName$( )**
Return a full filename:	**TrueFileName$( )**
Let user choose a file:	**FileOpenDlg( )**, **FileSaveAsDlg( )**
Return temporary filename:	**TempFileName$( )**
Locate files:	**LocateFile$( )**, **GetFolderPath$( )**

Working With Maps and Graphical Objects

Creating Map Objects

Creation statements:	**Create Arc**, **Create Ellipse**, **Create Frame**, **Create Line**, **Create Object**, **Create PLine**, **Create Point**, **Create Rect**, **Create Region**, **Create RoundRect**, **Create Text**, **AutoLabel**, **Create Multipoint**, **Create Collection**
Creation functions:	**CreateCircle( )**, **CreateLine( )**, **CreatePoint( )**, **CreateText( )**
Advanced operations:	**Create Object**, **Buffer( )**, **CartesianBuffer( )**, **CartesianOffset( )**, **CartesianOffsetXY( )**, **ConvexHull( )**, **Offset( )**, **OffsetXY( )**, **SphericalOffset( )**, **SphericalOffsetXY( )**
Store object in table:	**Insert**, **Update**
Create regions:	**Objects Enclose**

Modifying Map Objects

Modify object attribute:	**Alter Object**
Change object type:	**ConvertToRegion( )**, **ConvertToPLine( )**
Offset objects:	**Objects Offset**, **Objects Move**
Set the editing target:	**Set Target**

Erase part of an object:	**CreateCutter**, **Objects Erase**, **Erase( )**, **Objects Intersect**, **Overlap( )**
Merge objects:	**Objects Combine**, **Combine( )**, **Create Object**
Rotate objects:	**Rotate( )**, **RotateAtPoint( )**
Split objects:	**Objects Pline**, **Objects Split**
Add nodes at intersections:	**Objects Overlay**, **OverlayNodes( )**
Control object resolution:	**Set Resolution**
Store an object in a table:	**Insert**, **Update**
Check Objects for bad data:	**Objects Check**
Object processing:	**Objects Disaggregate**, **Objects Snap**, **Objects Clean**

Querying Map Objects

Return calculated values:	**Area( )**, **Perimeter( )**, **Distance( )**, **ObjectLen( )**, **Overlap( )**, **AreaOverlap( )**, **ProportionOverlap( )**
Return coordinate values:	**ObjectGeography( )**, **MBR( )**, **ObjectNodeX( )**, **ObjectNodeY( )**, **ObjectNodeZ( )**, **Centroid( )**, **CentroidX( )**, **CentroidY( )**, **ExtractNodes( )**, **IntersectNodes( )**
Return settings for coordinates, distance, area and paper units:	**SessionInfo( )**
Configure units of measure:	**Set Area Units**, **Set Distance Units**, **Set Paper Units**, **UnitAbbr$( )**, **UnitName$( )**
Configure coordinate system:	**Set CoordSys**
Return style settings:	**ObjectInfo( )**
Query a map layer's labels:	**LabelFindByID( )**, **LabelFindFirst( )**, **LabelFindNext( )**, **LabelInfo( )**

Working With Object Styles

Return current styles:	**CurrentPen( )**, **CurrentBorderPen( )**, **CurrentBrush( )**, **CurrentFont( )**, **CurrentLinePen( )**, **CurrentSymbol( )**, **Set Style**, **TextSize( )**
Return part of a style:	**LayerStyleInfo( ) function**, **StyleAttr( )**
Create style values:	**MakePen( )**, **MakeBrush( )**, **MakeFont( )**, **MakeSymbol( )**, **MakeCustomSymbol( )**, **MakeFontSymbol( )**, **Set Style**, **RGB( )**
Query object's style:	**ObjectInfo( )**
Modify object's style:	**Alter Object**
Reload symbol styles:	**Reload Symbols**
Style clauses:	**Pen clause**, **Brush clause**, **Symbol clause**, **Font clause**

Working With Map Windows

Open a map window:	**Map**
Create/edit 3DMaps:	**Create Map3D**, **Set Map3D**, **Map3DInfo( )**, **Create PrismMap**, **Set PrismMap**, **PrismMapInfo( )**
Add a layer to a map:	**Add Map**
Remove a map layer:	**Remove Map**
Label objects in a layer:	**AutoLabel**
Query a map's settings:	**MapperInfo( )**, **LabelOverrideInfo( ) function**, **LayerInfo( )**, **StyleOverrideInfo( ) function**
Change a map's settings:	**Set Map**
Create or modify thematic layers:	**Shade**, **Set Shade**, **Create Ranges**, **Create Styles**, **Create Grid**, **Relief Shade**
Query a map layer's labels:	**LabelFindByID( )**, **LabelFindFirst( )**, **LabelFindNext( )**, **LabelInfo( )**, **LabelOverrideInfo( ) function**

Creating the User Interface

ButtonPads (ToolBars)

Create a new ButtonPad:	**Create ButtonPad**
Modify a ButtonPad:	**Alter ButtonPad**
Modify a button:	**Alter Button**
Query the status of a pad:	**ButtonPadInfo( )**
Respond to button use:	**CommandInfo( )**
Restore standard pads:	**Create ButtonPad As Default**, **Create ButtonPads As Default**

Dialog Boxes

Display a standard dialog box:	**Ask( )**, **Note**, **ProgressBar**, **FileOpenDlg( )**, **FileSaveAsDlg( )**, **GetSeamlessSheet( )**
Display a custom dialog box:	**Dialog**
Dialog handler operations:	**Alter Control**, **TriggerControl( )**, **ReadControlValue( )**, **Dialog Preserve**, **Dialog Remove**
Determine whether user clicked OK:	**CommandInfo(CMD_INFO_DLG_OK)**
Disable progress bars:	**Set ProgressBars**
Modify a standard MapInfo Professional dialog box:	**Alter MapInfoDialog**

Menus

Define a new menu:	**Create Menu**
Redefine the menu bar:	**Create Menu Bar**
Modify a menu:	**Alter Menu**, **Alter Menu Item**
Modify the menu bar:	**Alter Menu Bar**, **Menu Bar**

Invoke a menu command:	**Run Menu Command**
Query a menu item's status:	**MenuItemInfoByHandler( )**, **MenuItemInfoByID( )**

Windows

Show or hide a window:	**Open Window**, **Close Window**, **Set Window**
Open a new window:	**Map**, **Browse**, **Graph**, **Layout**, **Create Redistricter**, **Create Legend**, **Create Cartographic Legend**, **LegendFrameInfo**
Determine a window's ID:	**FrontWindow( )**, **WindowID( )**
Modify an existing window:	**Set Map**, **Shade**, **Add Map**, **Remove Map**, **Set Browse**, **Set Graph**, **Set Layout**, **Create Frame**, **Set Legend**, **Set Cartographic Legend**, **Set Redistricter**, **StatusBar**, **Alter Cartographic Frame**, **Add Cartographic Frame**, **Remove Cartographic Frame**
Return a window's settings:	**WindowInfo( )**, **MapperInfo( )**, **LayerInfo( )**
Print a window:	**PrintWin**
Control window redrawing:	**Set Event Processing**, **Update Window**, **Control DocumentWindow clause**
Count number of windows:	**NumWindows( )**, **NumAllWindows( )**

System Event Handlers

React to selection:	**SelChangedHandler**
React to window closing:	**WinClosedHandler**
React to map changes:	**WinChangedHandler**
React to window focus:	**WinFocusChangedHandler**
React to DDE request:	**RemoteMsgHandler**, **RemoteQueryHandler( )**
React to OLE Automation method:	**RemoteMapGenHandler**
Provide custom tool:	**ToolHandler**
React to termination of application:	**EndHandler**

React to MapInfo Professional getting or losing focus:	**ForegroundTaskSwitchHandler**
Disable event handlers:	**Set Handler**

Communicating With Other Applications

DDE (Dynamic Data Exchange; Windows Only)

Start a DDE conversation:	**DDEInitiate( )**
Send a DDE command:	**DDEExecute**
Send a value via DDE:	**DDEPoke**
Retrieve a value via DDE:	**DDERequest$( )**
Close a DDE conversation:	**DDETerminate**, **DDETerminateAll**
Respond to a request:	**RemoteMsgHandler**, **RemoteQueryHandler( )**, **CommandInfo(CMD_INFO_MSG)**

Integrated Mapping

Set MapInfo Professional 's parent window:	**Set Application Window**
Set a Map window's parent:	**Set Next Document**
Create a Legend window:	**Create Legend**

Special Statements and Functions

Defines the name and argument list of a method/function in a .Net assembly	**Declare Method( )**
Launch another program:	**Run Program**
Return information about the system:	**SystemInfo( )**
Run a string as an interpreted command:	**Run Command**
Save a workspace file:	**Save Workspace**

Load a workspace file or an MBX:	**Run Application**
Configure a digitizing tablet:	**Set Digitizer**
Send a sound to the speaker:	**Beep**
Set data to be read by CommandInfo:	**Set Command Info**
Set duration of the drag-object delay:	**Set Drag Threshold**

2

New and Enhanced MapBasic Statements and Functions

These are several new statements and functions added to the MapBasic API that assist you in working with grids and in developing with .Net to create your integrated mapping and other third-party solutions.

We have increased the size of text controls from 64K (65535) to 512K (524288), so that you can open larger files in MapBasic.

Topics in this Section:

- New MapBasic Functions and Statements 35
- Enhancements to MapBasic Functions and Statements 35

New MapBasic Functions and Statements

We have added the following statement and functions in this version of MapBasic:

BrowserInfo function

Returns information about a Browser window, such as: the total number of rows or columns in the Browser window; or the row number, column number, or value contained in the current cell.

LibraryServiceInfo( ) function

Returns information about the Library Services, such as the current mode of operation, version, or default URL for the Library Service. It also gives the list of CSW URL's exposed by the MapInfo Manager sever.

Set LibraryServiceInfo statement

Resets the current Library Service related attributes.

GetCurrentPath( ) function

Returns the path of a special MapInfo Professional directory defined initially in the Preferences dialog to access specific MapInfo files.

Set Path statement

Allows user to change programmatically the path of a special MapInfo Professional directory defined initially in the Preferences dialog to access specific MapInfo files.

URL clause

Specifies the default Library Service URL to use. It checks that the input is a valid Library Service URL, and displays an error message if it is not valid.

Enhancements to MapBasic Functions and Statements

This version of MapBasic provides more robust description for the **Shade statement**.

Additions to Existing Functions and Statements

This version of MapBasic adds functionality to the following functions and statements:

CommandInfo( ) function

The behavior of the ComandInfo( ) function is slightly modified to support the new Browser window. The ComandInfo( ) function ignores any clicks made in the top-left corner of the Browser window—above the select column and to the left of the column headers. Previously, it would return an (x, y) of (0, 0). It also ignores clicks made beyond the last column or row. Previously, it would return an x value for the last column and y value for the last row.

Create ButtonPad statement

You can create a menu item or a toolbar button to launch a preferences dialog box directly. The Create ButtonPad statement now includes an example that illustrates how to create a toolbar button to launch the Browser Window Preferences dialog box (to bypass selecting it from the Options menu and Preferences dialog box).

Pen clause

Includes extended descriptions for the width and pattern components, and an example for how to apply line styles to a layer in a map as a layer style override.

Component	Description
width	Integer value, usually from 1 to 7, representing the thickness of the line (in pixels). To create an invisible line style, specify a width of zero, and use a pattern value of 1 (one). To specify a width using points, calculate the pen width from a point size using the **PointsToPenWidth( ) function**. This calculation multiplies the point size by 10 and then adds 10 to the result, so the pen width is always larger than 10.
pattern	Integer value from 1 to 118; see table below. Pattern 1 is invisible. To specify an interleaved line style, add 128 to the pattern. However, not all patterns benefit from an interleaved line style.

Run Application statement

The syntax of the Run Application statement is slightly modified:

Syntax

`Run Application` *`file`*

file is the name of an application file or a workspace file.

If the statement includes the **NoMRU** clause, the application or workspace name would not be added to the Most recently Used list of files.

Set Browse statement

MapInfo Professional has an improved Browser window. As a result, the behavior of the Set Browse statement is slightly modified to distinguish between a Browser window and a Redistricter window. It also now includes an optional clause for automatic column resizing.

The optional **Window** clause lets you specify which document window to use. If a *`window_id`* is not specified, then it searches for the most recently used Browser window or Redistricter window. If neither Redistricter nor Browser is the front-most window, but both exist, then Set Browse finds the Browser window instead of the Redistricter window. To specify which window type to use, either include the *`window_id`* with the Set Browse statement or make the Redistricter window the front-most window before calling Set Browse.

The optional **Columns** clause lets you set column resizing based on the width of the column header (title) and the contents that are in view. On first display, the Browser window automatically resizes columns to completely contain the data that is visible. When scrolling vertically, the Browser window does not automatically adjust the column width for the new data in view. You must set the Columns clause to make this happen. After recalculating column width, the width does not change while scrolling—columns do not resize to the new data in view. If the user manually resizes a column, then its width does not change.

Set Window statement

MapInfo Professional ignores the Antialiasing option when exporting the contents of a Browser window.

TableInfo( ) function

Includes two new attributes to read the table ID and parent table ID value in .tab files, and a new attribute to return TRUE if a table is managed in a library service.

attribute code	ID	TableInfo( ) returns
TAB_INFO_TABLEID	39	String result: returns the unique table ID for a TAB file. If there is no Table ID in a TAB file, then it returns an empty string.
TAB_INFO_PARENTTABLEID	40	String result: returns the table ID from which this TAB file was copied. If this was not created from another TAB file, then it returns an empty string.
TAB_INFO_ISMANAGED	41	Logical result: TRUE if table is managed in a library service or FALSE if it is not.

3

A – Z MapBasic Language Reference

This section describes the MapBasic language in detail. You will find both statements and function descriptions arranged alphabetically. Each is described in the following format:

Purpose

Brief description of the function, clause, or statement.

Restrictions

Information about limitations (for example, “The DDEInitiate function is only available under Microsoft Windows,” “You cannot issue a **For…Next** statement through the MapBasic window”).

Syntax

The format in which you should use the function or statement and explanation of argument(s).

Return Value

The type of value returned by the function.

Description

Thorough explanation of the function or statement's role and any other pertinent information.

Example

A brief example.

Related functions or statements. Most MapBasic statements can be typed directly into MapInfo Professional, through the MapBasic window. If a statement may not be entered through the MapBasic window, the Restrictions section identifies the limitation. Generally, flow-control statements (such as looping and branching statements) cannot be entered through the MapBasic window.

Abs( ) function

Purpose

Returns the absolute value of a number. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Abs (*num_expr*)

num_expr is a numeric expression.

Return Value

Float

Description

The **Abs( )** function returns the absolute value of the expression specified by *num_expr*.

If *num_expr* has a value greater than or equal to zero, **Abs( )** returns a value equal to *num_expr*. If *num_expr* has a negative value, **Abs( )** returns a value equal to the value of *num_expr* multiplied by negative one (-1).

Example

```
Dim f_x, f_y As Float
f_x = -2.5
f_y = Abs(f_x)

' f_y now equals 2.5
```

See Also:

Sgn( ) function

Acos( ) function

Purpose

Returns the arc-cosine value of a number. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Acos (*num_expr*)

num_expr is a numeric expression between one and negative one, inclusive.

Return Value

Float

Description

The **Acos( )** function returns the arc-cosine of the numeric *num_expr* value. In other words, **Acos( )** returns the angle whose cosine is equal to *num_expr*.

The result returned from **Acos( )** represents an angle, expressed in radians. This angle will be somewhere between zero and Pi radians (given that Pi is equal to approximately 3.141593, and given that Pi/2 radians represents 90 degrees).

To convert a degree value to radians, multiply that value by DEG_2_RAD. To convert a radian value into degrees, multiply that value by RAD_2_DEG. Your program must Include MAPBASIC.DEF in order to reference DEG_2_RAD or RAD_2_DEG.

Since cosine values range between one and negative one, the expression *num_expr* should represent a value no larger than one and no smaller than negative one.

Example

```
Include "MAPBASIC.DEF"
Dim x, y As Float
x = 0.5
y = Acos(x) * RAD_2_DEG
' y will now be equal to 60,
' since the cosine of 60 degrees is 0.5
```

See Also:

Asin( ) function, **Atn( ) function**, **Cos( ) function**, **Sin( ) function**, **Tan( ) function**

Add Cartographic Frame statement

The **Add Cartographic Frame** statement allows you to add cartographic frames to an existing cartographic legend created with the **Create Cartographic Legend statement**. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Add Cartographic Frame
   [ Window legend_window_id ]
      [ Custom ]
      [ Default Frame Title { def_frame_title } [ Font... ] ]
      [ Default Frame Subtitle { def_frame_subtitle } [ Font... ] ]
      [ Default Frame Style { def_frame_style } [ Font... ] ]
      [ Default Frame Border Pen... pen_expr ]
      Frame From Layer { map_layer_id | map_layer_name }
         [ Position ( x , y ) [ Units paper_units ] ]
      [ Using
         [ Column { column | object [ FromMapCatalog { On | Off }]} ]
```

```
      [ Label { expression | default } ]
      [ Title [ frame_title ] [ Font... ] ]
      [ SubTitle [ frame_subtitle ] [ Font... ] ]
      [ Border Pen... ]
      [ Style [Font...] [ NoRefresh ]
      [ Text { style_name } { Line Pen...
         | Region Pen... Brush...
         | Symbol Symbol... } ]
         [ , ... ]
   ]
   [ , ... ]
```

legend_window_id is an integer window identifier which you can obtain by calling the **FrontWindow( ) function** and **WindowID( ) function**.

def_frame_title is a string which defines a default frame title. It can include the special character "#" which will be replaced by the current layer name.

def_frame_subtitle is a string which defines a default frame subtitle. It can include the special character "#" which will be replaced by the current layer name.

def_frame_style is a string that displays next to each symbol in each frame. The "#" character will be replaced with the layer name. The "%" character will be replaced by the text "Line", "Point, "Region", as appropriate for the symbol. For example, "% of #" will expand to "Region of States" for the STATES.TAB layer.

pen_expr is a Pen expression, for example, *MakePen(width, pattern, color)*. If a default border pen is defined, then it will be become the default for the frame. If a border pen clause exists at the frame level, then it is used instead of the default.

map_layer_id or *map_layer_name* identifies a map layer; can be a SmallInt (e.g., use 1 to specify the top map layer other than Cosmetic) or a string representing the name of a table displayed in the map. For a theme layer you must specify the *map_layer_id*.

paper_units is a string representing a paper unit name (for example, "cm" for centimeters).

frame_title is a string which defines a frame title. If a **Title** clause is defined here for a frame, then it will be used instead of the *def_frame_title*.

frame_subtitle is a string which defines a frame subtitle. If a **SubTitle** clause is defined here for a frame, then it will be used instead of the *def_frame_subtitle*.

column is an attribute column name from the frame layer's table, or the object column (meaning that legend styles are based on the unique styles in the mapfile). The default is 'object'.

style_name is a string which displays next to a symbol, line, or region in a custom frame.

Description

If the **Custom** keyword is included, then each frame section must include a **Position** clause. If **Custom** is omitted and the legend is laid out in portrait or landscape, then the frames will be added to the end.

The **Position** clause controls the frame's position on the legend window. The upper left corner of the legend window has the position 0, 0. **Position** values use paper units settings, such as “in” (inches) or “cm” (centimeters). MapBasic has a current paper units setting, which defaults to inches; a MapBasic program can change this setting through the **Set Paper Units statement**.You can override the current paper units by including the optional **Units** subclause within the **Position** clause.

The defaults in this statement apply only to the frames being created in this statement. They have no affect on existing frames. Frame defaults used in the **Create Cartographic Legend statement** have no affect on frames created in this statement.

When you save to a workspace, the **FromMapCatalog OFF** clause is written to the workspace when specified. This requires the workspace to bumped up to 800. If the **FromMapCatalog ON** clause is specified, we do not write it to the workspace since it is default behavior. This lets us avoid bumping up the workspace version in this case.

FromMapCatalog ON retrieves styles from the MapCatalog for a live access table. If the table is not a live access table, MapBasic reverts to the default behavior for a non-live access table instead of throwing an error. The default behavior for a non-access table is **FromMapCatalog Off** (for example, map styles).

FromMapCatalog OFF retrieves the unique map styles for the live table from the server. This table must be a live access table that supports per record styles for this to occur. If the live table does not support per record styles than the behavior is to revert to the default behavior for live tables, which is to get the default styles from the MapCatalog (**FromMapCatalog ON**).

Label is a valid expression or default (meaning that the default frame style pattern is used when creating each style's text, unless the style clause contains text). The default is default.

The **Style** clause and the **NoRefresh** keyword allow you to create a custom frame that will not be overwritten when the legend is refreshed. If the **NoRefresh** keyword is used in the **Style** clause, then the table is not scanned for styles. Instead, the **Style** clause must contain your custom list of definitions for the styles displayed in the frame. This is done with the **Text** and appropriate **Line**, **Region**, or **Symbol** clause.

See Also:

Create Cartographic Legend statement, **Set Cartographic Legend statement**, **Alter Cartographic Frame statement**, **Remove Cartographic Frame statement**

Add Column statement

Purpose

Adds a new, temporary column to an open table, or updates an existing column with data from another table. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Add Column table ( column [ datatype ] )
{ Values const [ , const ... ] |
   From source_table
```

```
Set To expression
[ Where { dest_column = source_column |
   Within | Contains | Intersects } ]
[ Dynamic ] }
```

table is the name of the table to which a column will be added.

column is the name of a new column to add to that table.

datatype is the data type of the column, defined as Char(*width*), Float, Integer, SmallInt, Decimal(*width*, *decimal_places*), Date or Logical, DateTime; if not specified, type defaults to Float.

source_table is the name of a second open table.

expression is the expression used to calculate values to store in the new column; this expression usually extracts data from the *source_table*, and it can include aggregate functions.

dest_column is the name of a column from the destination table (*table*).

source_column is the name of a column from the *source_table*.

Dynamic specifies a dynamic (hot) computed column that can be automatically update: if you include this keyword, then subsequent changes made to the source table are automatically applied to the destination table.

Description

The **Add Column** statement creates a temporary new column for an existing MapInfo Professional table. The new column will not be permanently saved to disk. However, if the temporary column is based on base tables, and if you save a workspace while the temporary column is in use, the workspace will include information about the temporary column, so that the temporary column will be rebuilt if the workspace is reloaded. To add a permanent column to a table, use the **Alter Table statement** and **Update statement**.

See Also:

Alter Table statement, **Update statement**

Filling the New Column with Explicit Values

Using the **Values** clause, you can specify a comma-separated list of explicit values to store in the new column.

The following example adds a temporary column to a table of “ward” regions. The values for the new column are explicitly specified, through the **Value** clause.

```
Open Table "wards"
Add Column wards(percent_dem)
Values 31,17,22,24,47,41,66,35,32,88
```

Filling the New Column with Values from Another Table

If you specify a **From** clause instead of a **Values** clause, MapBasic derives the values for the new column from a separate table (*source_table*). Both tables must already be open.

When you use a **From** clause, MapInfo Professional joins the two tables. To specify how the two tables are joined, include the optional **Where** clause. If you omit the **Where** clause, MapInfo Professional automatically tries to join the two tables using the most suitable method.

A **Where** clause of the form *Where column = column* joins the two tables by matching column values from the two tables. This method is appropriate if a column from one of your tables has values matching a column from the other table (e.g., you are adding a column to the States table, and your other table also has a column containing state names).

If both tables contain map objects, the **Where** clause can specify a geographic join. For example, if you specify the clause **Where Contains**, MapInfo Professional constructs a join by testing whether objects from the *source_table* contain objects from the table that is being modified.

The following example adds a “County” column to a “Stores” table. The new column will contain county names, which are extracted from a separate table of county regions:

```
Add Column
   stores(county char(20) 'add "county" column
   From counties 'derive data from counties table...
   Set to cname 'using the counties table's "cname" column
   Where Contains 'join: where a county contains a store site
```

The **Where Contains** method is appropriate when you add a column to a table of point objects, and the secondary table represents objects that contain the points.

The following example adds a temporary column to the States table. The new column values are derived from a second table (City_1K, a table of major U.S. cities). After the completion of the **Add Column** statement, each row in the States table will contain a count of how many major cities are in that state.

```
Open Table "states" Interactive
Open Table "city_1k" Interactive

Add Column states(num_cities)
   From city_1k 'derive values from other table
   Set To Count(*)'count cities in each state
   Where Within 'join: where cities fall within states
```

The **Set To** clause in this example specifies an aggregate function, **Count(*)**. Aggregate functions are described below.

Filling an Existing Column with Values from Another Table

To update an existing column instead of adding a new column, omit the datatype parameter and specify a **From** clause instead of a **Values** clause. When updating an existing column, MapBasic ignores the **Dynamic** clause.

Filling the New Column with Aggregate Data

If you specify a **From** clause, you can calculate values for the new column by aggregating data from the second table. To perform data aggregation, specify a **Set To** clause that includes an aggregate function.

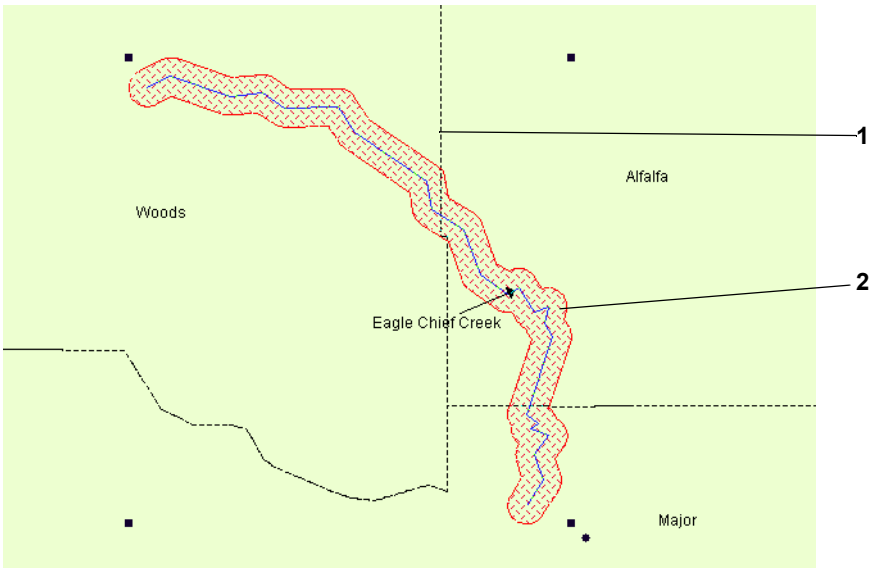
The following table lists the available aggregate functions.

Function	Value Stored In The New Column
Avg(*col*)	Average of values from rows in the source table.
Count(*)	Number of rows in the source table that correspond to the row in the table being updated.
Max(*col*)	Largest of the values from rows in the source table.
Min(*col*)	Smallest of the values from rows in the source table.
Sum(*col*)	Sum of the values from rows in the source table.
WtAvg(*col*, *weight_col*)	Weighted average of the values from the source table; the averaging is weighted so that rows having a large *weight_col* value have more of an impact than rows having a small *weight_col* value.
Proportion Avg(*col*)	Average calculation that makes adjustments based on how much of an object is within another object.
Proportion Sum(*col*)	Sum calculation that makes adjustments based on how much of an object is within another object.
Proportion WtAvg(*col* , *weight_col*)	Weighted average calculation that makes adjustments based on how much of an object is within another object.

ⓘ Count returns an integer value. All other functions return a float value. (No MapBasic function, aggregate or otherwise, returns a decimal value. A decimal field is only a way of storing the data. The arithmetic is done with floating point numbers.)

Most of the aggregate functions operate on data values only. The last three functions (Proportion Sum, Proportion Avg, Proportion WtAvg) perform calculations that take geographic relationships into account. This is best illustrated by example.

Suppose you have a Counties table, containing county boundary regions and demographic information (such as population) about each county. You also have a Risk table, which contains a region object. The object in the Risk table represents some sort of area that is at risk; perhaps the region object represents an area in danger of flooding due to proximity to a river.



1 County Boundaries 2 Risk Buffer Region

Given these two tables, you might want to calculate the population that lives within the risk region. If half of a county's area falls within the risk region, you will consider half of that county's population to be at risk; if a third of a county's area falls within the risk region, you will consider a third of that county's population to be at risk; etc.

The following example calculates the population at risk by using the **Proportion Sum** aggregate function, then stores the calculation in a new column (population_at_risk):

```
Add Column Risk(population_at_risk Integer)
   From counties
      Set To Proportion Sum(county_pop)
      Where Intersects
```

For each county that is at least partly within the risk region, MapInfo Professional adds some or all of the counties county_pop value to a running total.

The **Proportion Sum** function produces results based on an assumption—the assumption that the number being totalled is distributed evenly throughout the region. If you use **Proportion Sum** to process population statistics, and half of a region falls within another region, MapInfo Professional adds half of the region's population to the total. In reality, however, an area representing half of a region does not necessarily contain half of the region's population. For example, the population of New York State is not evenly distributed, because a large percentage of the population lives in New York City.

If you use **Proportion Sum** in cases where the data values are not evenly distributed, the results may not be realistic. To ensure accurate results, work with smaller region objects (for example, operate on county regions instead of state regions).

The **Proportion Avg** aggregate function performs an average calculation which takes into account the percentage of an object that is covered by another object. Continuing the previous example, suppose the County table contains a column, median_age, that indicates the median age in each county.

The following statement calculates the median age within the risk zone:

```
Add Column Risk(age Float)
   From Counties
      Set To Proportion Avg(median_age)
      Where Intersects
```

For each row in the County table, MapInfo Professional calculates the percentage of the risk region that is covered by the county; that calculation produces a number between zero and one, inclusive. MapInfo Professional multiplies that number by the county's median_age value, and adds the result to a running total. Thus, if a county has a median_age value of 50, and if the county region covers 10% of the risk region, MapInfo Professional adds 5 (five) to the running total, because 10% of 50 is 5.

Both **Proportion Sum** and **Proportion Avg** keep running totals. For example:

If half the county falls in the risk area, then you take half the value and add it to the running total. If it is 10%, then you add 10% of the value to the running total. However, Proportion Avg should be an average, so if 4 counties intersect the risk area, then you take the running total and divide by 4.

If county1 intersects the risk region, and 50% of county1 intersects the risk region, and the population of county1 is 66, then you add 33 to the running total.

If 30% of county2's area intersects the risk area and the population is 100, then add 30 to the running total.

If county3 has 20% overlap with the risk area and has a population of 50, then add 10 to the running total.

If county4 has 10% overlap with the risk area and has a population of 60, then add 6 to the running total.

Then the Proportion Sum is 33+30+10+6 = 82

Then the Proportion Avg is (33+30+10+6)/4 = 20 (or 21 depending on round off, but I think 20).

Proportion WtAvg is similar to **Proportion Avg**, but it also lets you specify a data column for weighting the average calculation; the weighting is also proportionate. For example:

Weighted Average should take a weighted value from another column; for the previous example there is another column called RuralPercent in the County table. If the risk is for flood and the rural areas are where it floods, then for risk you only want the population from the rural area.

If county1 has 50% overlap with the risk region, a population of 66, and a RuralPercent of 0.8, then add (0.5 * 66 * 0.8) = 26.

If county3, 4, and 5 are all 50% rural, then:

county3 0.3 * 100 * 0.5 = 15

county4 0.2 * 50 * 0.5 = 5

county5 0.1 * 60 * 0.5 = 3

Then the proportion weighted Avg is: (26 + 15 + 5 + 3)/4 = 12

Using Proportion… Functions with Non-Region Objects

When you use **Proportion** functions and the source table contains region objects, MapInfo Professional calculates percentages based on the overlap of regions. However, when the source table contains non-region objects, MapInfo Professional treats each object as if it were completely inside or completely outside of the destination region (depending on whether the non-region object's centroid is inside or outside of the destination region).

Dynamic Columns

If you include the optional **Dynamic** keyword, the new column becomes a dynamic computed column, meaning that subsequent changes made to the source table are automatically applied to the destination table.

If you create a dynamic column, and then close the source table used to calculate the dynamic column, the column values are frozen (the column is no longer updated dynamically).

Similarly, if a geographic join is used in the creation of a dynamic column, and you close either of the maps used for the geographic join, the column values are frozen.

Add Map statement

Purpose

Adds one or more graphic layers, or a group layer, to a Map window. You can also select a destination group layer and/or position to insert the new layers.

Syntax 1

```
Add Map
   [ Window window_id ] [ Auto ]
   Layer table [, table   [ Animate ] ...  ]
   [[ DestGroupLayer group_id ]  Position position ]
```

window_id is the window identifier of a Map window.

table is the name of a mappable open table to add to a Map window.

group_id is the identification for a group layer, either as a integer value (the group number) or as a string value (the group name).

position is the 1-based index within the destination group where to insert the new list of layers.

Syntax 2

```
Add Map
   [ Window window_id ] [ Auto ]
   GroupLayer ("friendly_name" [, item ...])
   [[ DestGroupLayer group_id ]  Position position ]
```

where:

```
   item = table | [GroupLayer ("friendly_name" [, item ...])
```

Description

The **Add Map** statement adds one or more open tables, or a group layer, to a Map window, but not both within the same statement. The group layer may contain any number of nested group layers. MapInfo Professional then automatically redraws the Map window, unless you have suppressed redraws through a **Set Event Processing statement** *Off* statement or **Set Map statement** *Redraw Off* statement.

The *window_id* parameter is an integer window identifier representing an open Map window; you can obtain a window identifier by calling the **FrontWindow( ) function** and **WindowID( ) function**. If the **Add Map** statement does not specify a *window_id* value, the statement affects the topmost Map window.

If you include the optional **Auto** keyword, MapInfo Professional tries to automatically position the map layer, or group layers at an appropriate place in the set of layers. A raster table or a map of region objects would be placed closer to the bottom of the map, while a map of point objects would be placed on top.

If you omit the **Auto** keyword, the specified table becomes the topmost layer in the window; in other words, when the map is redrawn, the new layer, or group layers will be drawn last. You can then use the **Set Map statement** to alter the order of layers in the Map window.

If a **DestGroupLayer** is specified the Auto keyword will be ignored and the list of layers, or the group layer will be inserted into the layer list, in the group specified, at the position specified. A group id of 0 is the top level list. If the **DestGroupLayer** is omitted the group ID defaults to 0.

The *position* is the 1-based index within the destination group of where to insert the new list of layers. If the position is omitted it is assumed to be the first position in the group (position = 1). If the position given exceeds the number of items in the destination group, the new layers and/or groups will be inserted at the end of the destination group.

Layer and group IDs may be the numeric ID or name. Group IDs range from 0 to the total number of groups in the list.

You cannot insert into the middle of a set of thematic layers. Thematic layers are inserted in a certain order as they are created and this order is maintained. If the destination position would cause the new layers to be inserted within a set of thematic layers, the final position will be adjusted to avoid that.

Adding Layers of Different Projections

If the layer added is a raster table, and the map does not already contain any raster map layers, the map adopts the coordinate system and projection of the raster image. If a Map window contains two or more raster layers, the window dynamically changes its projection, depending on which image occupies more of the window at the time.

If raster re-projection is turned on, then MapInfo Professional retains the coordinate system of the map even if you add a raster table to the map.

If the layer added is not a raster table, MapInfo Professional continues to display the Map window using whatever coordinate system and projection were used before the **Add Map** statement, even if the table specified is stored with a different native projection or coordinate system. When a table's native projection differs from the projection of the Map window, MapInfo Professional converts the table coordinates “on the fly” so that the entire Map window appears in the same projection.

When MapInfo Professional converts map layers in this fashion, map redraws take longer, since MapInfo Professional must perform mathematical transformations while drawing the map.

Using Animation Layers to Speed Up Map Redraws

If the **Add Map** statement includes the Animate keyword, the added layer becomes a special layer known as the animation layer. When an object in the animation layer is moved, the Map window redraws very quickly, because MapInfo Professional only redraws the one animation layer.

For an example of animation layers, see the sample program ANIMATOR.MB.

The animation layer is useful in real-time applications, where map features are updated frequently. For example, you can develop a fleet-management application that represents each vehicle as a point object. You can receive current vehicle coordinates by using GPS (Global Positioning Satellite) technology, and then update the point objects to show the current vehicle locations on the map. In this type of application, where map objects are constantly changing, the map redraws much more quickly if the objects being updated are stored in the animation layer instead of a conventional layer.

The following example opens a table (Vehicles) and makes the table an animation layer:

```
Open Table "vehicles" Interactive
Add Map Layer vehicles Animate
```

In general, the last table to be followed by the **Animate** keyword will be the animation layer. Only one layer at a time can be the Animation layer.

To terminate the animation layer processing, issue a **Remove Map statement** *Layer Animate* statement.

Animation layers have special restrictions. For example, users cannot use the Info tool to click on objects in an animation layer. Also, each Map window can have only one animation layer. For more information about animation layers, see the *MapBasic User's Guide*.

Example

```
Open Table "world"
Map From world
Open Table "cust1992" As customers
Open Table "lead1992" As leads
Add Map Auto Layer customers, leads
```

Add a group layer example:

```
Open Table world
Open Table worldcap
Add Map Auto GroupLayer("new group", worldcap, world)
```

```
Open Table ocean
Add Map Layer ocean DestGrouplayer "new group" position 3
```

See Also:

Map statement, **Remove Map statement**, **Set Map statement**

Alter Button statement

Purpose

Enables, disables, selects, or deselects a button from a ButtonPad (toolbar).

Syntax

```
Alter Button { handler | ID button_id }
   [ { Enable | Disable } ]
   [ { Check | Uncheck } ]
```

handler is the handler that is already assigned to an existing button. The *handler* can be the name of a MapBasic procedure, or a standard command code (e.g., M_TOOLS_RULER or M_WINDOW_LEGEND) from MENU.DEF.

button_id is a unique integer button identification number.

Description

If the **Alter Button** statement specifies a handler (e.g., a procedure name), MapInfo Professional modifies all buttons that call that handler. If the statement specifies a *button_id* number, MapInfo Professional modifies only the button that has that ID.

The **Disable** keyword changes the button to a grayed-out state, so that the user cannot select the button.

The **Enable** keyword enables a button that was previously disabled.

The **Check** and **Uncheck** keywords select and deselect ToggleButton type buttons, such as the Show Statistics Window button. The **Check** keyword has the effect of “pushing in” a ToggleButton control, and the **Uncheck** keyword has the effect of releasing the button. For example, the following statement selects the Show Statistics Window button:

```
Alter Button M_WINDOW_STATISTICS Check
```

Checking or unchecking a standard MapInfo Professional button does not automatically invoke that button's action; thus, checking the Show/Hide Statistics button does not actually show the Statistics window—it only affects the appearance of the button. To invoke an action as if the user had checked or unchecked the button, issue the appropriate statement; in this example, the appropriate statement is the **Open Window statement** *Statistics*.

Similarly, you can use the **Check** keyword to change the appearance of a ToolButton. However, checking a ToolButton does not actually select that tool, it only changes the appearance of the button. To make a standard tool the active tool, issue a **Run Menu Command statement**, such as the following:

```
Run Menu Command M_TOOLS_RULER
```

To make a custom tool the active tool, use the syntax *Run Menu Command ID* `IDnum`.

See Also:

Alter ButtonPad statement, **Create ButtonPad statement**, **Run Menu Command statement**

Alter ButtonPad statement

Purpose

Displays / hides a ButtonPad (toolbar), or adds / removes buttons.

Syntax

```
Alter ButtonPad { current_title | ID pad_num }
   [ Add button_definition [ button_definition ... ] ]
   [ Remove { handler_num | ID button_id } [ , ... ] ]
   [ Title new_title ]
   [ Width w ]
   [ Position ( x, y ) [ Units unit_name ] ]
   [ ToolbarPosition ( row, column ) ]
   [ { Show | Hide } ]
   [ { Fixed | Float | Top | Left | Right | Bottom } ]
   [ Destroy ]
```

current_title is the toolbar's title string (e.g., “Main”).

pad_num is the ID number for a standard toolbar:

- 1 for Main
- 2 for Drawing
- 3 for Tools
- 4 for Standard
- 5 for Database Management System (DBMS)
- 6 Web Services
- 7 Reserved

handler_num is an integer handler code, such as M_TOOLS_RULER (1710), from MENU.DEF.

button_id is a custom button's unique identification number.

new_title is a string that becomes the toolbar's new title; visible when toolbar is floating.

w is the pad width, in terms of the number of buttons across.

x, *y* specify the toolbar's position when floating; specified in paper units (e.g., inches).

unit_name is a string paper unit name (e.g., “in” for inches, “cm” for centimeters).

row, *column* specify the toolbar's position when docked (e.g., 0, 0 places the pad at the left edge of the top row of toolbars, and 0, 1 represents the second toolbar on the top row).

- *row* position starts at the top and increases in value going to the bottom. It is a relative value to the rows existing in the same position (top or bottom). When there is a menu bar in the same position, then the numbers become relative to the menu bar. When a toolbar is just below the menu bar, its row value is 0. If it is directly above the menu bar, then its row value is -1.
- *column* position starts at the left and increases in value going to the right. It is a relative value to the columns existing in the same position (left or right). For example, if a toolbar is docked to the left and the menu bar is docked to the left position, then the column number for the column left of the menu bar is -1. The column number for the column to the right of the menu bar is 0.

Each *button_definition* clause can consist of the keyword **Separator**, or it can have the following syntax:

```
{ PushButton | ToggleButton | ToolButton }
   Calling { procedure | menu_code | OLE methodname | DDE server, topic }
   [ ID button_id ]
   [ Icon icon_code [ File file_spec ] ]
   [ Cursor cursor_code [ File file_spec ] ]
   [ DrawMode dm_code ]
   [ HelpMsg msg ]
   [ ModifierKeys { On | Off } ]
   [ { Enable | Disable } ]
   [ { Check | Uncheck } ]
```

procedure is the handler procedure to call when a button is used.

menu_code is a standard MapInfo Professional menu code from MENU.DEF, such as M_FILE_OPEN (102); MapInfo Professional runs the menu command when the user uses the button.

methodname is a string specifying an OLE method name. For details on the **Calling OLE** clause syntax, see **Create ButtonPad statement**.

server and *topic* are strings specifying a DDE server and topic name. For details on the **Calling DDE** clause syntax, see **Create ButtonPad statement**.

button_id specifies the unique button number. This number can be used: as a tag in help; as a parameter to allow the handler to determine which button is in use (in situations where different buttons call the same handler); or as a parameter to be used with the **Alter Button statement**.

Icon *icon_code* specifies the icon to appear on the button; *icon_code* can be one of the standard MapInfo icon codes listed in ICONS.DEF, such as MI_ICON_RULER (11). If the File sub-clause specifies the name of a file containing icon resources, *icon_code* is an integer resource ID identifying a resource in the file.

Cursor *cursor_code* specifies the shape the mouse cursor should adopt whenever the user chooses a ToolButton tool; *cursor_code* is a code, such as MI_CURSOR_ARROW (0), from ICONS.DEF. This clause applies only to ToolButtons. If the File sub-clause specifies the name of a file containing icon resources, *cursor_code* is an integer resource ID identifying a resource in the file.

dm_code specifies whether the user can click and drag, or only click with the tool; *dm_code* is a code, such as DM_CUSTOM_LINE (33), from ICONS.DEF. Applies only to ToolButtons.

msg is a string that specifies the button's status bar help and, optionally, ToolTip help. The first part of *msg* is the status bar help message. If the *msg* string includes the letters *\n* then the text following the *\n* is used as the button's ToolTip help.

The **ModifierKeys** clause applies only to ToolButtons; it controls whether the shift and control keys affect “rubber-band” drawing if the user drags the mouse while using a ToolButton. Default is Off (modifier keys have no effect).

Description

Use the **Alter ButtonPad** statement to show, hide, modify, or destroy an existing ButtonPad. For an introduction to ButtonPads, see the *MapBasic User Guide*.

To show or hide a ButtonPad, include the **Show** or **Hide** keyword; see example below. The user also can show or hide ButtonPads by choosing the **Options > Toolbars** command.

To set whether the pad is fixed to the top of the screen (“docked”) or floating like a window, include the **Fixed** or the **Float** keyword. The user can also control whether the pad is docked or not by dragging the pad to or from the top of the screen. For more control over the location on the screen that the pad is docked to, use the **Top** (which is the same as using **Fixed**), **Left**, **Right**, or **Bottom** keywords.

When a pad is floating, its position is controlled by the **Position** clause; when a pad is docked, its position is controlled by the **ToolbarPosition** clause.

To destroy a ButtonPad, include the **Destroy** keyword. Once a ButtonPad is destroyed, it no longer appears in the **Options > Toolbars** dialog box.

The **Alter ButtonPad** statement can add buttons to existing ButtonPads, such as Main and Drawing. There are three types of button controls you can add: PushButton controls (which the user can click and release—for example, to display a dialog box); ToggleButton controls (which the user can select by clicking, then deselect by clicking again); and ToolButton controls (which the user can select, and then use for clicking on a Map or Layout window).

If you include the optional **Disable** keyword when adding a button, the button is disabled (grayed out) when it appears. Subsequent **Alter Button statement**s can enable the button. However, if the button's handler is a standard MapInfo Professional command, MapInfo Professional automatically enables or disables the button depending on whether the command is currently enabled.

If you include the optional **Check** keyword when adding a ToggleButton or a ToolButton, the button is automatically selected (“checked”) when it first appears.

If the user clicks while using a custom ToolButton tool, MapInfo Professional automatically calls the tool's handler, unless the user cancels (e.g., by pressing the Esc key while dragging the mouse). A handler procedure can call **CommandInfo( )** to determine where the user clicked. If two or more tools call the same handler procedure, the procedure can call **CommandInfo( )** to determine the ID of the button currently in use.

Custom Icons and Cursors

The **Icon** clause specifies the icon that appears on the button. If you omit the **File** clause, then the parameter *n* must refer to one of the icon codes listed in ICONS.DEF, such as MI_ICON_RULER (11).

> MapInfo Professional has many built-in icons that are not part of the normal user interface. To see a demonstration of these icons, run the sample program ICONDEMO.MBX. This sample program displays icons, and also lets you copy any icon's define code to the clipboard (so that you can then paste the code into your program).

The **File** *file_spec* sub-clause refers to a DLL file that contains bitmap resources; the *n* parameter refers to the ID of a bitmap resource. For more information on creating Windows icons, see the *MapBasic User Guide*.

A ToolButton definition also can include a **Cursor** clause, which controls the appearance of the mouse cursor while the user is using the custom tool. Available cursor codes are listed in ICONS.DEF, such as MI_CURSOR_CROSSHAIR (138) or MI_CURSOR_ARROW (0). The procedure for specifying a custom cursor is similar to the procedure for specifying a custom icon.

For custom icon size requirements for different MapInfo Professional versions, see **Create ButtonPad statement About Icon Size on page 162**.

Custom Drawing Modes

A ToolButton definition can include a **DrawMode** clause, which controls whether the user can drag with the tool (e.g., to draw a line) or only click (e.g., to draw a point). The following table lists the available drawing modes. Codes in the left column are defined in ICONS.DEF.

DrawMode parameter	ID	Description
DM_CUSTOM_POINT	34	The user cannot drag while using the custom tool.
DM_CUSTOM_LINE	33	As the user drags, a line connects the cursor with the location where the user clicked.
DM_CUSTOM_RECT	32	As the user drags, a rectangular marquee appears.
DM_CUSTOM_CIRCLE	30	As the user drags, a circular marquee appears.
DM_CUSTOM_ELLIPSE	31	As the user drags, an elliptical marquee appears; if you include the ModifierKeys clause, the user can force the marquee to a circular shape by holding down the Shift key.

DrawMode parameter	ID	Description
DM_CUSTOM_POLYGON	35	The user may draw a polygon. To retrieve the object drawn by the user, use the function call: **CommandInfo( ) function (CMD_INFO_CUSTOM_OBJ)**.
DM_CUSTOM_POLYLINE	36	The user may draw a polyline. To retrieve the object drawn by the user, use the function call: **CommandInfo( ) function (CMD_INFO_CUSTOM_OBJ)**.

All of the draw modes except for DM_CUSTOM_POINT (34) support the Autoscroll feature, which allows the user to scroll a Map or Layout by clicking and dragging to the edge of the window. To disable autoscroll, see **Set Window statement**.

ⓘ MapBasic supports an additional draw mode that is not available to MapInfo Professional users. If a custom ToolButton has the following **Calling** clause *Calling M_TOOLS_SEARCH_POLYGON* (*1733*) then the tool allows the user to draw a polygon. When the user double-clicks to close the polygon, MapInfo Professional selects all objects (from selectable map layers) within the polygon. The polygon is not saved.

Examples

The following example shows the Main ButtonPad and hides the Drawing ButtonPad:

```
Alter ButtonPad "Main" Show
Alter ButtonPad "Drawing" Hide
```

The next example docks the Main ButtonPad and sets its docked position to 0,0 (upper left):

```
Alter ButtonPad "Main" Fixed ToolbarPosition(0,0)
```

The next example moves the Main ButtonPad so that it is floating instead of docked, and sets its floating position to half an inch inside the upper-left corner of the screen.

```
Alter ButtonPad "Main" Float Position(0.5,0.5) Units "in"
```

The sample program, ScaleBar, contains the following **Alter ButtonPad** statement, which adds a custom ToolButton to the Tools ButtonPad. (Note that “ID 3" identifies the Tools ButtonPad.)

```
Alter ButtonPad ID 3
   Add
      Separator
      ToolButton
         Icon MI_ICON_CROSSHAIR
         HelpMsg "Draw a distance scale on a map\nScale Bar"
         Cursor MI_CURSOR_CROSSHAIR
         DrawMode DM_CUSTOM_POINT
```

```
        Calling custom_tool_routine
  Show
```

> ⓘ The **Separator** keyword inserts space between the last button on the Tools ButtonPad and the new MI_CURSOR_CROSSHAIR (138) button.

See Also:

Alter Button statement, **ButtonPadInfo( ) function**, **Create ButtonPad statement**, **Set Window statement**

Alter Cartographic Frame statement

Purpose

The **Alter Cartographic Frame** statement changes a frame(s) position, title, subtitle, border and style of an existing cartographic legend created with the **Create Cartographic Legend statement**. (To change the size, position or title of the legend window, use the **Set Window statement**.) You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Alter Cartographic Frame
[ Window legend_window_id ]
Id { frame_id }
   [ Position ( x, y ) [ Units paper_units ] ]
   [ Title [ frame_title ] [ Font... ] ]
   [ SubTitle [ frame_subtitle ] [ Font... ] ]
   [ Border Pen... ]
   [ Style [ Font... ]
      [ ID { id } Text { style_name } ]
      [Line Pen... | Region Pen... Brush... | Symbol Symbol... ] ]
      [ , ... ]
```

legend_window_id is an integer window identifier which you can obtain by calling the **FrontWindow( ) function** and **WindowID( ) function**.

frame_id is the ID of the frame on the legend. You cannot use a layer name. For example, three frames on a legend would have the successive ID's 1, 2, and 3.

frame_title is a string which defines a frame title.

frame_subtitle is a string which defines a frame subtitle.

id is the position within the style list for that frame. Currently there is no MapBasic function to get information about the number of styles in a frame.

style_name is a string that displays next to each symbol for the frame specified in ID. The “#” character will be replaced with the layer name. The “%” character will be replaced by the text “Line”, “Point, “Region”, as appropriate for the symbol. For example, “% of #” will expand to “Region of States” for the frame corresponding to the STATES.TAB layer.

Description

If a **Window** clause is not specified MapInfo Professional will use the topmost legend window.

The **Position** clause controls the frame's position on the legend window. The upper left corner of the legend window has the position 0, 0. Position values use paper units settings, such as “in” (inches) or “cm” (centimeters). MapBasic has a current paper units setting, which defaults to inches; a MapBasic program can change this setting through the **Set Paper Units statement**. An **Alter Cartographic Frame** statement can override the current paper units by including the optional **Units** subclause within the **Position** clause.

The **Title** and **SubTitle** clauses accept new text, new font or both.

The **Style** clause must contain a list of definitions for the styles displayed in frame. You can only update the Style type for a custom style. You can update the Text of any style. There is no way to add or remove styles from any type of frame.

See Also:

Create Cartographic Legend statement, **Set Cartographic Legend statement**, **Add Cartographic Frame statement**, **Remove Cartographic Frame statement**

Alter Control statement

Purpose

Changes the status of a control in the active custom dialog box.

Syntax

```
Alter Control id_num
   [ Title { title | From Variable array_name } ]
   [ Value value ]
   [ { Enable | Disable } ]
   [ { Show | Hide } ]
   [ Active ]
```

id_num is an integer identifying one of the controls in the active dialog box.

title is a string representing the new title to assign to the control.

array_name is the name of an array variable; used to reset the contents of ListBox, MultiListBox, and PopupMenu controls.

value is the new value to associate with the specified control.

Restrictions

You cannot issue this statement through the MapBasic window.

Description

The **Alter Control** statement modifies one or more attributes of a control in the active dialog box; accordingly, the **Alter Control** statement should only be issued while a dialog box is active (for example, from within a handler procedure that is called by one of the dialog box controls). If there are two or more nested dialog boxes on the screen, the **Alter Control** statement only affects controls within the topmost dialog box.

The *id_num* specifies which dialog box control should be modified; this corresponds to the *id_num* parameter specified within the **ID** clause of the **Dialog statement**.

Each of the optional clauses (**Title**, **Value**, **Enable**/**Disable**, **Hide**/**Show**, **Active**) modifies a different attribute of a dialog box control. Note that all of these clauses can be included in a single statement; thus, a single **Alter Control** statement could change the name, the value, and the enabled/disabled status of a dialog box control.

Some attributes do not apply to all types of controls. For example, a Button control may be enabled or disabled, but has no value attribute.

The **Title** clause resets the text that appears on most controls (except for Picker controls and EditText controls; to reset the contents of an EditText control, set its Value). If the control is a ListBox, MultiListBox, or PopupMenu control, the **Title** clause can read the control's new contents from an array of string variables, by specifying a **From Variable** clause.

The **Active** keyword applies only to EditText controls. An **Alter Control…Active** statement puts the keyboard focus on the specified EditText control.

Use the **Hide** and **Show** keywords to make controls disappear or reappear.

To de-select all items in a MultiListBox control, use a value setting of zero. To add a list item to the set of selected MultiListBox items, issue an **Alter Control** statement with a positive integer value corresponding to the number of the list item.

In this case, do not issue the **Alter Control** statement from within the MultiListBox control's handler.

You can use an **Alter Control** statement to modify the text that appears in a StaticText control. However, MapInfo Professional cannot increase the size of the StaticText control after it is created. Therefore, if you plan to alter the length of a StaticText control, you may want to pad it with spaces when you first define it. For example, your **Dialog statement** could include the following clause:

```
Control StaticText ID 1 Title "Message goes here" + Space$(30)
```

Example

The following example creates a dialog box containing two checkboxes, an OK button, and a Cancel button. Initially, the OK button is disabled (grayed out). The OK button is only enabled if the user selects one or both of the check boxes.

```
Include "mapbasic.def"
Declare Sub Main
Declare Sub checker
Sub Main
```

```
   Dim browse_it, map_it As Logical
   Dialog
      Title "Display a file"
      Control CheckBox
         Title "Display in a Browse window"
         Value 0
         Calling checker
         ID 1
         Into browse_it
      Control CheckBox
         Title "Display in a Map window"
         Value 0
         Calling checker
         ID 2
         Into map_it
      Control CancelButton
      Control OKButton
         ID 3
         Disable
   If CommandInfo(CMD_INFO_DLG_OK) Then
   ' ... then the user clicked OK...
   End If
End Sub
Sub checker
' If either check box is checked,
' enable the OK button; otherwise, Disable it.
   If ReadControlValue(1) Or ReadControlValue(2) Then
      Alter Control 3 Enable
   Else
      Alter Control 3 Disable
   End If
End Sub
```

Alter MapInfoDialog statement

Purpose

Disables, hides, or assigns new values to controls in MapInfo Professional's standard dialog boxes.

Restrictions

CAUTION: **The Alter MapInfoDialog statement may not be supported in future versions of MapInfo Professional. As a result, MapBasic programs that use this statement may not work correctly when run using future versions of MapInfo Professional. Use this statement with caution.**

Syntax 1 (assigning non-default settings)

```
Alter MapInfoDialog dialog_ID
   Control control_ID
      { Disable | Hide | Value new_value } [ , { Disable... } ]
   [ Control... ]
```

Syntax 2 (restoring default settings)

```
Alter MapInfoDialog dialog_ID Default
```

dialog_ID is an integer ID number, indicating which MapInfo Professional dialog box to alter.

control_ID is an integer ID number, 1 or larger, indicating which control to modify.

new_value is a new value assigned to the dialog box control.

Description

Use this statement if you need to disable, hide, or assign new values to controls—buttons, check boxes, etc.—in MapInfo Professional's standard dialog boxes.

Use this statement to modify only MapInfo Professional's standard dialog boxes. To modify custom dialog boxes that you create using the **Dialog statement**, use the **Alter Control statement**.

Determining ID Numbers

To determine a dialog box's ID number, run MapInfo Professional with this command line:

```
mapinfow.exe -helpdiag
```

After you run MapInfo Professional with the *-helpdiag* argument, display a MapInfo Professional dialog box and click the Help button. Ordinarily, the Help button launches Help, but because you used the *-helpdiag* argument, MapInfo Professional displays the ID number of the current dialog box.

There are different “common dialog boxes” (such as the Open and Save dialog boxes) for different versions of Windows. If you want to modify a common dialog box, and if your application will be used under different versions of Windows, you may need to issue two **Alter MapInfoDialog** statements, ne for each version of the common dialog box.

Each individual control has an ID number. For example, most OK buttons have an ID number of 1, and most Cancel buttons have an ID number of 2. To determine the ID number for a specific control, you must use a third-party developer's utility, such as the Spy++ utility that Microsoft provides with its C compiler. The MapBasic software does not provide a Spy++ utility.

Although the **Alter MapInfoDialog** statement changes the initial appearance of a dialog box, the changes do not have any effect unless the user clicks **OK**. For example, you can use **Alter MapInfoDialog** to store an address in the Find dialog box; however, MapInfo Professional will not perform the Find operation unless you display the dialog box and the user clicks **OK**.

Types of Changes Allowed

Use the **Disable** keyword to disable (gray out) the control.

Use the **Hide** keyword to make the control disappear.

Use the **Value** clause to change the setting of the control.

When you alter common dialog boxes (e.g., the Open dialog box), you may reset the item selected in a combo box control, or you may assign new text to static text, button, and edit box controls.

You can change the orientation control in the Page Setup dialog box. The Portrait and Landscape buttons are 1056 and 1057, respectively.

When you alter other MapInfo Professional dialog boxes, the following list summarizes the types of changes you may make.

- **Button**, **static text, edit box**, **editable combo box**: You may assign new text by using a text string in the *new_value* parameter.
- **List box**, **combo box**: You may set which item is selected by using a numeric *new_value*.
- **Checkbox**: You may set the checkbox (specify a value of 1) or clear it (value of zero).
- **Radio button**: Setting a button's value to 1 selects that button from the radio group.
- **Symbol style button**: You may assign a new symbol style (e.g., use the return value from the **MakeSymbol( ) function**).
- **Pen style button**: You may assign a new Pen value.
- **Brush style button**: You may assign a new Brush value.
- **Font style button**: You may assign a new Font value.
- **Combined Pen/Brush style** button: Specify a Pen value to reset the Pen style, or specify a Brush value to reset the Brush style. (For an example of this type of control, see MapInfo Professional's Region Style dialog box, which appears when you double-click an editable region.)

Example

The following example alters MapInfo Professional's Find dialog box by storing a text string (“23 Main St.”) in the first edit box and hiding the Respecify button.

```
If SystemInfo(SYS_INFO_MIVERSION) = 400 Then
   Alter MapInfoDialog 2202
      Control 5 Value "23 Main St."
      Control 12 Hide
End If
Run Menu Command M_ANALYZE_FIND
```

The ID number 2202 refers to the Find dialog box. Control 5 is the edit box where the user types an address. Control 12 is the Respecify button, which this example hides. All ID numbers are subject to change in future versions of MapInfo Professional; therefore, this example calls the **SystemInfo( ) function** to determine the MapInfo Professional version number.

See Also:

Alter Control statement, **SystemInfo( ) function**

Alter Menu statement

Purpose

Adds or removes items from an existing menu.

Syntax 1

```
Alter Menu { menuname | ID menu_id }
   Add menudef [ , menudef... ]
```

Where each *menudef* defines a menu item, according to the syntax:

```
   newmenuitem
   [ ID menu_item_id ]
   [ HelpMsg help ]
   [ { Calling handler | As menuname } ]
```

menuname is the name of an existing menu (for example, “File”).

menu_id is a standard integer menu ID from 1 to 64; 1 represents the File menu.

newmenuitem is a string, the name of an item to add to the specified menu.

menu_item_id is a custom integer menu item identifier, which can be used in subsequent **Alter Menu Item statement**s.

help is a string that will appear on the status bar while the menu item is highlighted.

handler is the name of a procedure, or a code for a standard menu command (e.g., M_FILE_NEW), or a special syntax for handling the menu event by calling OLE or DDE. If you specify a command code for a standard MapInfo Professional Show/Hide command (such as M_WINDOW_STATISTICS), the *newmenuitem* string must start with an exclamation point and include a caret (^), to preserve the item's Show/Hide behavior. For more details on the different types of handler syntax, see the **Create Menu statement**.

Syntax 2

```
Alter Menu { menuname | ID menu_id }
   Remove { handler | submenuname | ID menu_item_id }
      [ , { handler | submenuname | ID menu_item_id } ... ]
```

menuname is the name of an existing menu.

menu_id is an integer menu ID from 1 to 64; 1 represents the File menu.

handler is either the name of a sub procedure or the code for a standard MapInfo Professional command.

submenuname is the name of a hierarchical submenu to remove from the specified menu.

menu_item_id is a custom integer menu item identifier.

Description

The **Alter Menu** statement adds menu items to an existing menu or removes menu items from an existing menu.

The statement can identify the menu to be modified by specifying the name of the menu (e.g., “File”) through the *menuname* parameter.

If the menu to be modified is one of the standard MapInfo Professional menus, the **Alter Menu** statement can identify which menu to alter by using the **ID** clause. The **ID** clause identifies the menu by a number from 1 to 64. The following table lists the names and ID numbers of all standard MapInfo Professional menus.

ID Numbers for Menus

Menu Name	Define	ID	Description
File	M_FILE	1	File menu.
Edit	M_EDIT	2	Edit menu.
Search	M_SEARCH	3	Search menu.
Query	M_QUERY	3	Query menu.
Programs	M_PGM	4	Programs menu.
Tools	M_TOOLS	4	Tools menu.
Options	M_OPTIONS	5	Options menu.
Window	M_WINDOW	6	Window menu.
Help	M_HELP	7	Help menu.
Browse	M_BROWSE	8	Browse menu. Ordinarily, this only appears when a Browser window is the active window.
Map	M_MAP	9	Map menu. Ordinarily, this menu is only available when a Map window is active.
Layout	M_LAYOUT	10	Layout menu. Available when a Layout window is active.
Graph	M_GRAPH	11	Graph menu. Available when a Graph window is active.
MapBasic	M_MAPBASIC	12	MapBasic menu. Available when the MapBasic window is active.
Redistrict	M_REDISTRICT	13	Redistrict menu. Available when a Districts Browser is active.
Objects	M_OBJECTS	14	Objects menu.
Table	M_TABLE	15	Table menu.

Menus 16 through 36 are shortcut menus, which appear if the user clicks with the right mouse button.

ID Numbers for Shortcut Menus

Menu Name	Define	ID	Description
DefaultShortcut	M_SHORTCUT_DFLT	16	The default shortcut menu. This menu appears if the user right-clicks on a window that does not have its own shortcut menu defined.
MapperShortcut	M_SHORTCUT_MAPPER	17	The Map window shortcut menu.
BrowserShortcut	M_SHORTCUT_BROWSER	18	The Browse window shortcut menu.
LayoutShortcut	M_SHORTCUT_LAYOUT	19	The Layout window shortcut menu.
GrapherShortcut	M_SHORTCUT_GRAPHER	20	The Graph window shortcut menu. This menu contains options for creating graphs.
CmdShortcut	M_SHORTCUT_CMD	21	The MapBasic window shortcut menu.
RedistrictShortcut	M_SHORTCUT_REDISTRICTER	22	The Redistricting shortcut menu; available when the Districts Browser is active.
LegendShortcut	M_SHORTCUT_LEGEND	23	The Legend window shortcut menu.
GrapherShortcut	M_SHORTCUT_GRAPHTDG	24	The Graph window shortcut menu. This menu contains options for formatting graphs already created.
3DMapShortcut	M_SHORTCUT_3DMAP	25	The 3D Map window shortcut menu.
MessageWinShortcut	M_SHORTCUT_MSG_WIN	26	The Message window shortcut menu.
StatisticsWinShortcut	M_SHORTCUT_STAT_WIN	27	The Statistics window shortcut menu.
AdornmentShortcut	M_SHORTCUT_ADORNMENT	32	The Adornment window shortcut.
LcLayersShortcut	M_SHORTCUT_LC_LAYERS	33	The menu you get when you right-click a regular layer in the layer list
LcMapsShortcut	M_SHORTCUT_LC_MAPS	34	The menu you get when you right-click a Map node in the layer list

ID Numbers for Shortcut Menus

Menu Name	Define	ID	Description
LcGroupsShortcut	M_SHORTCUT_LC_GROUPS	35	The menu you get when you right-click a Group Layer in the layer list
TableAdornmentShortcut	M_SHORTCUT_TABLEADORNMENT	36	Reserved for future use.

ID Numbers for Non-Shortcut Menus

Menu Name	Define	ID	Description
3DWindow	M_3DMAP	28	3D Map window.
Graph	M_GRAPHTDG	29	Graph menu.
Legend	M_LEGEND	31	Legend menu.

When altering a Custom menu (even if you create it with an Custom ID, such as 999), you are required to use the Custom *menuname*, not the Custom *ID*, to alter it.

Examples

The following statement adds an item to the File menu.

```
Alter Menu "File" Add
   "Special" Calling sub_procedure_name
```

In the following example, the menu to be modified is identified by its number.

```
Alter Menu ID 1 Add
   "Special" Calling sub_procedure_name
```

In the following example, the menu item that is added contains an **ID** clause. The ID number (300) can be used in subsequent **Alter Menu Item statement**s.

```
Alter Menu ID 1 Add
   "Special" ID 300 Calling sub_procedure_name
```

The following example removes the custom item from the File menu.

```
Alter Menu ID 1 Remove sub_procedure_name
```

The sample program, TextBox, uses a **Create Menu statement** to create a menu called “TextBox,” and then issues the following **Alter Menu** statement to add the TextBox menu as a hierarchical menu located on the Tools menu:

```
Alter Menu "Tools" Add
   "(-",
   "TextBox" As "TextBox"
```

The following example adds a custom command to the Map window's shortcut menu (the menu that appears when an MapInfo Professional user right-clicks on a Map window).

```
Alter Menu ID 17 Add
   "Find Nearest Site" Calling sub_procedure_name
```

See Also:

Alter Menu Bar statement, **Alter Menu Item statement**, **Create Menu statement**, **Create Menu Bar statement**

Alter Menu Bar statement

Purpose

Adds or removes menus from the menu bar.

Syntax

```
Alter Menu Bar { Add | Remove }
   { menuname | ID menu_id }
      [ , { menuname | ID menu_id } ... ]
```

menuname is the name of an available menu (e.g., “File”)

menu_id is a standard menu ID from one to fifteen; one represents the File menu.

Description

The **Alter Menu Bar** statement adds or removes one or more menus from the current menu bar.

The *menuname* parameter is a string representing the name of a menu, such as “File” or “Edit”. The *menuname* parameter may also refer to the name of a custom menu created by a **Create Menu statement** (see example below)

If the application is running on a non-English language version of MapInfo, and if the menu names have been translated, the **Alter Menu Bar** statement must specify the translated version of the menu name. However, each of MapInfo Professional's standard menus (File, Edit, etc.) also has a menu ID, which you can use regardless of whether the menu names have been translated. For example, specifying ID 2 always refers to the Edit menu, regardless of whether the menu has been translated.

For a list of MapInfo Professional's standard menu names and their corresponding ID numbers, see **Alter Menu statement**.

Adding Menus to the Menu Bar

An **Alter Menu Bar Add** statement adds a menu to the right end of the menu bar. If you need to insert a menu at another position on the menu bar, use the **Create Menu Bar statement** to redefine the entire menu bar.

If you add enough menus to the menu bar, the menu bar wraps down onto a second line of menu names.

Removing Menus from the Menu Bar

An **Alter Menu Bar Remove…** statement removes a menu from the menu bar. However, the menu remains part of the "pool" of available menus. Thus, the following pair of statements would first remove the Query menu from the menu bar, and then add the Query menu back onto the menu bar (at the right end of the bar).

```
Alter Menu Bar Remove "Query"
Alter Menu Bar Add "Query"
```

After an **Alter Menu Bar Remove…** statement removes a menu, MapInfo Professional ignores any hotkey sequences corresponding to items that were on the removed menu. For example, a MapInfo Professional user might ordinarily press **Ctrl-O** to bring up the **File** menu's Open dialog box; however, if an **Alter Menu Bar Remove** statement removed the **File** menu, MapInfo Professional would ignore any **Ctrl-O** key-presses.

Example

The following example creates a custom menu, called DataEntry, then uses an **Alter Menu Bar Add** statement to add the DataEntry menu to MapInfo Professional's menu bar.

```
Declare Sub addsub
Declare Sub editsub
Declare Sub delsub
Create Menu "DataEntry" As
   "Add" Calling addsub,
   "Edit" Calling editsub,
   "Delete" Calling delsub
'Remove the Window menu and Help menu
Alter Menu Bar Remove ID 6, ID 7
'Add the custom menu, then the Window & Help menus
Alter Menu Bar Add "DataEntry", ID 6, ID 7
```

Before adding the custom menu to the menu bar, this program removes the **Help** menu (menu ID 7) and the **Window** menu (ID 6) from the menu bar. The program then adds the custom menu, the **Window** menu, and the **Help** menu to the menu bar. This technique guarantees that the last two menus will always be **Window** and **Help**.

See Also:

Alter Menu statement, **Alter Menu Item statement**, **Create Menu statement**, **Create Menu Bar statement**, **Menu Bar statement**

Alter Menu Item statement

Purpose

Alters the status of a specific menu item.

Syntax

```
Alter Menu Item { handler | ID menu_item_id }
   { [ Check | Uncheck ] |
    [ Enable | Disable ] |
    [ Text itemname ] |
    [ Calling handler | As menuname ] }
```

handler is either the name of a Sub procedure or the code for a standard MapInfo Professional command.

menu_item_id is an integer that identifies a menu item; this corresponds to the *menu_item_id* parameter specified in the statement that created the menu item (**Create Menu statement** or **Alter Menu statement**).

itemname is the new text for the menu item (may contain embedded codes).

menuname is the name of an existing menu.

Description

The **Alter Menu Item** statement alters one or more of the items that make up the available menus. For example, you could use the **Alter Menu Item** statement to check or disable (gray out) a menu item.

The statement must either specify a handler (e.g., the name of a procedure in the same program), or an **ID** clause to indicate which menu item(s) to modify. Note that it is possible for multiple, separate menu items to call the same handler procedure. If the **Alter Menu Item** statement includes the name of a handler procedure, MapInfo Professional alters all menu items that call that handler. If the statement includes an **ID** clause, MapInfo Professional alters only the menu item that was defined with that ID.

The **Alter Menu Item** statement can only refer to a menu item ID if the statement which defined the menu item included an **ID** clause. A MapBasic application cannot refer to menu item IDs created by other MapBasic applications.

The **Check** clause and the **Uncheck** clause affect whether the item appears with a checkmark on the menu. Note that a menu item may only be checked if it was defined as “checkable” (for example, if the **Create Map statement** included a “!” as the first character of the menu item name).

The **Disable** clause and the **Enable** clause control whether the item is disabled (grayed out) or enabled. Note that MapInfo Professional automatically enables and disables various menu items based on the current circumstances. For example, the **File > Close** command is disabled whenever there are no tables open. Therefore, MapBasic applications should not attempt to enable or disable standard MapInfo Professional menu items. Similarly, although you can treat specific tools as menu items (by referencing defines from MENU.DEF, such as M_TOOLS_RULER), you should not attempt to enable or disable tools through the **Alter Menu Item** statement.

The **Text** clause allows you to rename a menu item.

The **Calling** clause specifies a handler for the menu item. If the user chooses the menu item, MapInfo Professional calls the item's handler.

Examples

The following example creates a custom “DataEntry” menu.

```
Declare Sub addsub
Declare Sub editsub
Declare Sub delsub
Create Menu "DataEntry" As
   "Add" Calling addsub,
   "Edit" Calling editsub,
   "Delete" ID 100 Calling delsub,
   "Delete All" ID 101 Calling delsub
'Remove the Help menu
Alter Menu Bar Remove ID 7
'Add both the new menu and the Help menu
Alter Menu Bar Add "DataEntry" , ID 7
```

The following **Alter Menu Item statement** renames the “Edit” item to read “Edit…”

```
Alter Menu Item editsub Text "Edit…"
```

The following statement disables the “Delete All” menu item.

```
Alter Menu Item ID 101 Disable
```

The following statement disables both the “Delete” and the “Delete All” items, because it identifies the handler procedure *delsub*, which is the handler for both menu items.

```
Alter Menu Item delsub Disable
```

See Also:

Alter Menu statement, **Alter Menu Bar statement**, **Create Menu statement**

Alter Object statement

Purpose

Modifies the shape, position, or graphical style of an object. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Alter Object obj
   { Info object_info_code, new_info_value |
      Geography object_geo_code , new_geo_value |
      Node { Add [ Position polygon_num, node_num ] ( x, y ) |
         Set Position polygon_num, node_num ( x , y ) |
         Remove Position polygon_num, node_num
      }
   }
```

obj is an object variable.

object_info_code is an integer code relating to the **ObjectInfo( ) function** (e.g., OBJ_INFO_PEN).

new_info_value specifies the new *object_info_code* attribute to apply (e.g., a new Pen style).

object_geo_code is an integer code relating to the **ObjectGeography( ) function** (e.g., OBJ_GEO_POINTX).

new_geo_value specifies the new *object_geo_code* value to apply (e.g., the new x-coordinate).

polygon_num is a integer value (one or larger), identifying one polygon from a region object or one section from a polyline object.

node_num is a integer value (one or larger), identifying one node from a polyline or polygon.

x, *y* are x- and y-coordinates of a node.

Description

The **Alter Object** statement alters the shape, position, or graphical style of an object.

The effect of an **Alter Object** statement depends on whether the statement includes an **Info** clause, a **Node** clause, or a **Geography** clause. If the statement includes an **Info** clause, MapBasic alters the object's graphical style (e.g., the object's Pen and Brush styles). If the statement includes a **Node** clause, MapBasic adds, removes, or repositions a node (this applies only to polyline or region objects). If the statement includes a **Geography** clause, MapBasic alters a geographical attribute for objects other than polylines and regions (e.g., the x- or y-coordinate of a point object).

Info clause

By issuing an **Alter Object** statement with an **Info** clause, you can reset an object's style (e.g., the Pen or Brush). The **Info** clause lets you modify the same style attributes that you can query through the **ObjectInfo( ) function**.

For example, you can determine an object's current Brush style by calling the **ObjectInfo( ) function**:

```
Dim b_fillstyle As Brush
b_fillstyle = ObjectInfo(Selection.obj, OBJ_INFO_BRUSH)
```

Conversely, the following **Alter Object** statement allows you to reset the Brush style:

```
Alter Object obj_variable_name
   Info OBJ_INFO_BRUSH, b_fillstyle
```

Note that you use the same code (e.g., OBJ_INFO_BRUSH) in both the **ObjectInfo( ) function** and the **Alter Object** statement.

The table below summarizes the values you can specify in the **Info** clause to perform various types of style alterations. Note that the *obj_info_code* values are defined in the standard MapBasic definitions file, MAPBASIC.DEF. Accordingly, your program should Include “MAPBASIC.DEF” if you intend to use the **Alter Object…Info** statement.

***obj_info_code* Value**	**ID**	**Result of Alter Object**
OBJ_INFO_PEN	2	Resets object's Pen style; *new_info_value* must be a Pen expression.
OBJ_INFO_BRUSH	3	Resets object's Brush style; *new_info_value* must be a Brush expression.
OBJ_INFO_TEXTFONT	2	Resets a Text object's Font style; *new_info_value* must be a Font expression.
OBJ_INFO_SYMBOL	2	Resets a Point object's Symbol style; *new_info_value* must be a Symbol expression.
OBJ_INFO_SMOOTH	4	Resets a Polyline object's smoothed/unsmoothed setting; *new_info_value* must be a logical expression.
OBJ_INFO_FRAMEWIN	4	Changes which window is displayed in a Layout frame; *new_info_value* must be an integer window ID.
OBJ_INFO_FRAMETITLE	6	Changes the title of a Frame object; *new_info_value* must be a string.
OBJ_INFO_TEXTSTRING	3	Changes the text string that comprises a Text object; *new_info_value* must be a string expression.
OBJ_INFO_TEXTSPACING	4	Changes a Text object's line spacing; *new_info_value* must be a float value of 1, 1.5, or 2.
OBJ_INFO_TEXTJUSTIFY	5	Changes a Text object's alignment; *new_info_value* must be 0 for left-justified, 1 for center-justified, or 2 for right-justified.
OBJ_INFO_TEXTARROW	6	Changes a Text object's label line setting; *new_info_value* must be 0 for no line, 1 for simple line, or 2 for a line with an arrow.

Geography clause

By issuing an **Alter Object** statement with a **Geography** clause, you can alter an object's geographical coordinates. The **Geography** clause applies to all object types except for polylines and regions. To alter the coordinates of a polyline or region object, use the **Node** clause (described below) instead of the **Geography** clause.

The **Geography** clause lets you modify the same attributes that you can query through the **ObjectGeography( ) function**. For example, you can obtain a line object's end coordinates by calling the **ObjectGeography( ) function**:

```
Dim o_cable As Object
Dim x, y As Float
x = ObjectGeography(o_cable, OBJ_GEO_LINEENDX)
y = ObjectGeography(o_cable, OBJ_GEO_LINEENDY)
```

Conversely, the following **Alter Object** statements let you alter the line object's end coordinates:

```
Alter Object o_cable
   Geography OBJ_GEO_LINEENDX, x
Alter Object o_cable
   Geography OBJ_GEO_LINEENDY, y
```

You use the same codes (e.g., OBJ_GEO_LINEENDX) in both the **ObjectGeography( ) function** and the **Alter Object** statement.

The table below summarizes the values you can specify in the Geography clause in order to perform various types of geographic alterations. Note that the *obj_geo_code* values are defined in the standard MapBasic definitions file, MAPBASIC.DEF. Your program should Include "MAPBASIC.DEF" if you intend to use the **Alter Object…Geography** statement.

***obj_geo_code* Value**	**ID**	**Result of Alter Object**
OBJ_GEO_MINX	1	Alters object's minimum bounding rectangle.
OBJ_GEO_MINY	2	Alters object's MBR.
OBJ_GEO_MAXX	3	Alters object's MBR; does not apply to Point objects.
OBJ_GEO_MAXY	4	Alters object's MBR; does not apply to Point objects.
OBJ_GEO_ARCBEGANGLE	5	Alters beginning angle of an Arc object.
OBJ_GEO_ARCENDANGLE	6	Alters ending angle of an Arc object.
OBJ_GEO_LINEBEGX	1	Alters a Line object's starting node.
OBJ_GEO_LINEBEGY	2	Alters a Line object's starting node.
OBJ_GEO_LINEENDX	3	Alters a Line object's ending node.
OBJ_GEO_LINEENDY	4	Alters a Line object's ending node.
OBJ_GEO_POINTX	1	Alters a Point object's x coordinate.
OBJ_GEO_POINTY	2	Alters a Point object's y coordinate.

obj_geo_code Value	ID	Result of Alter Object
OBJ_GEO_ROUNDRADIUS	5	Alters the diameter of the circle that defines the rounded corner of a Rounded Rectangle object.
OBJ_GEO_TEXTLINEX	5	Alters x coordinate of the end of a Text object's label line.
OBJ_GEO_TEXTLINEY	6	Alters y coordinate of the end of a Text object's label line.
OBJ_GEO_TEXTANGLE	7	Alters rotation angle of a Text object.

Node clause

By issuing an **Alter Object** statement with a **Node** clause, you can add, remove, or reposition nodes in a polyline or region object.

If the **Node** clause includes an **Add** sub-clause, the **Alter Object** statement adds a node to **the** object. If the **Node** clause includes a **Remove** sub-clause, the statement removes a node. If the Node clause includes a **Set Position** sub-clause, the statement repositions a node.

The **Alter Object** statement's **Node** clause is often used in conjunction with the **Create Pline statement** and the **Create Region statement**. Create statements allow you to create new polyline and region objects. However, Create statements are somewhat restrictive, because they force you to state at compile time the number of nodes that will comprise the object. In some situations, you may not know how many nodes should go into an object until run-time.

If your program will not know until run-time how many nodes should comprise an object, you can issue a **Create Pline statement** or a **Create Region statement** which creates an “empty” object (an object with zero nodes). Your program can then issue an appropriate number of **Alter Object…Node Add** statements, to add nodes as needed.

Within the **Node** clause, the **Position** sub-clause includes two parameters, *polygon_num* and *node_num*, that let you specify exactly which node you want to reposition or remove. The **Position** sub-clause is optional when you are adding a node. The *polygon_num* and *node_num* parameters should always be 1 (one) or greater.

The *polygon_num* parameter specifies which polygon in a multiple-polygon region (or which section in a multiple-section polyline) should be modified.

Region Centroids

The Centroid of a Region can be set by using the **Alter Object** command with the syntax noted below:

```
Alter Object Obj Geography OBJ_GEO_CENTROID, PointObj
```

Note that *PointObj* is a point object. This differs from other values input by **Alter Object Geography**, which are all scalars. A point is needed in this instance because we need two values which define a point. The Point that is input is checked to make sure it is a valid Centroid (for example, it is inside the region). If the *Obj* is not a region, or if *PointObj* is not a point object, or if the point is not a valid centroid, then an error is returned.

An easy way to center an X and Y value for a centroid is as follows:

```
Alter Object Obj Geography OBJ_GEO_CENTROID, CreatePoint(X, Y)
```

The user can also query the centroid by using the **ObjectGeography( ) function** as follows:

```
PointObj = ObjectGeography(Obj, OBJ_GEO_CENTROID)
```

There are other ways to get the Centroid, including the **Centroid( ) function**, **CentroidX( ) function**, and **CentroidY( ) function**.

OBJ_GEO_CENTROID is defined in MAPBASIC.DEF.

Multipoint Objects and Collections

The **Alter Object** statement supports the following object types.

- **Multipoint**: sets a Multipoint symbol as shown in the following:

  ```
  Alter Object obj_variable_mpoint
     Info OBJ_INFO_SYMBOL, NewSymbol
  ```

- **Collection**: By issuing an **Alter Object** statement with an **Info** clause, you can reset collection parts (Region, Polyline or Multipoint) inside the collection object. The **Info** clause allows you to modify the same attributes that you can query through the **ObjectInfo( ) function**. For example, you can determine a collection object's region part by calling the **ObjectInfo( ) function**:

  ```
  Dim ObjRegion As Object
  ObjRegion = ObjectInfo(Selection.obj, OBJ_INFO_REGION)
  ```

  Also, the following **Alter Object** statement allows you to reset the region part of a collection object:

  ```
  Alter Object obj_variable_name
  Info OBJ_INFO_REGION, ObjRegion
  ```

You use the same code (e.g., OBJ_INFO_REGION) in both the **ObjectInfo( ) function** and the **Alter Object** statement.

The **Alter Object** statement inserts and deletes nodes to/from Multipoint objects.

```
Alter Object obj Node statement
```

To insert nodes within a Multipoint object:

```
Dim mpoint_obj as object
Create Multipoint Into Variable mpoint_obj 0
Alter Object mpoint_obj Node Add (0,1)
Alter Object mpoint_obj Node Add (2,1)
```

Nodes for Multipoint are always added at the end.

To delete nodes from a Multipoint object:

```
Alter Object mpoint_obj Node Remove Position polygon_num, node_num
```

mpoint_obj is a Multipoint object variable.

polygon_num is ignored for Multipoint, it is advisable to set it to 1.

node_num is the number of a node to be removed.

To set nodes inside a Multipoint object:

```
Alter Object mpoint_obj Node Set Position polygon_num, node_num (x,y)
```

mpoint_obj is a Multipoint object variable.

polygon_num is ignored for Multipoint, it is advisable to set it to 1.

node_num is the number of a node to be changed.

x and *y* are the new coordinates of the node *node_num*.

Example

```
Dim myobj As Object, i As Integer
Create Region Into Variable myobj 0
For i = 1 to 10
   Alter Object myobj
      Node Add (Rnd(1) * 100, Rnd(1) * 100)
   Next
```

After using the **Alter Object** statement to modify an object, use an **Insert statement** or an **Update statement** to store the object in a table.

See Also:

Create Pline statement, **Create Region statement**, **Insert statement**, **ObjectGeography( ) function**, **ObjectInfo( ) function**, **Update statement**

Alter Table statement

Purpose

Alters the structure of a table. Cannot be used on linked tables. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Alter Table table (
   [ Add columnname columntype [, ...] ]
   [ Modify columnname columntype [, ...] ]
   [ Drop columnname [, ...] ]
   [ Rename oldcolumnname newcolumnname [, ...] ]
   [ Order columnname, columnname [,...] ] )
   [ Interactive ]
```

table is the name of an open table.

columnname is the name of a column; column names can be up to 31 characters long, and can contain letters, numbers, and the underscore character, and column names cannot begin with numbers.

columntype indicates the datatype of a table column (including the field width if necessary).

oldcolumnname represents the previous name of a column to be renamed.

newcolumnname represents the intended new name of a column to be renamed.

Description

The **Alter Table** statement lets you modify the structure of an open table, allowing you to add columns, change column widths or datatypes, drop (delete) columns, rename columns, and change column ordering.

If you have edited a table, you must save or discard your edits before you can use the **Alter Table** statement.

Each *columntype* should be one of the following: integer, SmallInt, float, decimal(size, decplaces), char(size), date, or logical, DateTime.

By including an **Add** clause in an **Alter Table** statement, you can add new columns to your table. By including a **Modify** clause, you can change the datatypes of existing columns. A **Drop** clause lets you delete columns, while a **Rename** clause lets you change the names of existing columns. The **Order** clause lets you specify the order of the columns. Altogether, an **Alter Table** statement can have up to five clauses. Note that each of these five clauses can operate on a list of columns; thus, with a single **Alter Table** statement, you can make all of the structural changes that you need to make (see example below).

The **Order** clause affects the order of the columns, not the order of rows in the table. Column order dictates the relative positions of the columns when you browse the table; the first column appears at the left edge of a Browser window, and the last column appears at the right edge. Similarly, a table's first column appears at the top of an Info tool window.

If a MapBasic application issues an **Alter Table** statement affecting a table which has memo fields, the memo fields will be lost. No warning will be displayed.

An **Alter Table** statement may cause map layers to be removed from a Map window, possibly causing the loss of themes or cosmetic objects. If you include the **Interactive** keyword, MapInfo Professional prompts the user to save themes and/or cosmetic objects (if themes or cosmetic objects are about to be lost).

Example

In the following example, we have a hypothetical table, “gcpop.tab” which contains the following columns: pop_88, metsize, fipscode, and utmcode. The **Alter Table** statement below makes several changes to the gcpop table. First, a **Rename** clause changes the name of the pop_88 column to population. Then the **Drop** clause deletes the metsize, fipscode, and utmcode columns. An **Add** clause creates two new columns: a small (2-byte) integer column called schoolcode, and a floating point column called federalaid. Finally, an **Order** clause specifies the order for the new set of columns: the schoolcode column comes first, followed by the population column, etc.

```
Open Table "gcpop"
Alter Table gcpop
   (Rename pop_88 population
```

```
Drop metsize, fipscode, utmcode
Add schoolcode SmallInt, federalaid Float
Order schoolcode, population, federalaid)
```

See Also:

Add Column statement, **Create Index statement**, **Create Map statement**, **Create Table statement**

ApplicationDirectory$( ) function

Purpose

Returns a string containing the path from which the current MapBasic application is executing. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

ApplicationDirectory$( )

Return Value

String expression, representing a directory path.

Description

By calling the **ApplicationDirectory$( )** function from within a compiled MapBasic application, you can determine the directory or folder from which the application is running. If no application is running (e.g., if you call the function by typing into the MapBasic window), **ApplicationDirectory$( )** returns a null string.

To determine the directory or folder where the MapInfo Professional software is installed, call the **ProgramDirectory$( ) function**.

Example

```
Dim sAppPath As String
sAppPath = ApplicationDirectory$( )
' At this point, sAppPath might look like this:
'
' "C:\MAPBASIC\CODE\"
```

See Also:

ProgramDirectory$( ) function, **ApplicationName$( ) function**

ApplicationName$( ) function

Purpose

Returns a string containing the name of the current MapBasic application that is running. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

```
ApplicationName$( )
```

Return Value

String expression representing the name of the MapBasic program.

Description

By calling the ApplicationName$( ) function from within a compiled MapBasic application, you can determine the name of the running application. If no application is running (if you call the function by typing into the MapBasic window), then ApplicationName$( ) returns an empty string.

To determine the path from which the current MapBasic application is executing call the **ApplicationDirectory$( ) function**.

Example

```
Dim sAppName As String
sAppName = ApplicationName$( )
' At this point, sAppName might look like this:
'
' "Test.MBX"
```

See Also:

ApplicationDirectory$( ) function

Area( ) function

Purpose

Returns the geographical area of an Object. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Area(*obj_expr*, *unit_name*)

obj_expr is an object expression.

unit_name is a string representing the name of an area unit (e.g., “sq km”).

Return Value

Float

Description

The **Area( )** function returns the area of the geographical object specified by *obj_expr*.

The function returns the area measurement in the units specified by the *unit_name* parameter; for example, to obtain an area in acres, specify “acre” as the *unit_name* parameter. See **Set Area Units statement** for the list of available unit names.

Only regions, ellipses, rectangles, and rounded rectangles have any area. By definition, the area of a point, arc, text, line, or polyline object is zero. The **Area( )** function returns approximate results when used on rounded rectangles. MapBasic calculates the area of a rounded rectangle as if the object were a conventional rectangle.

For the most part, MapInfo Professional performs a Cartesian or Spherical operation. Generally, a spherical operation is performed unless the coordinate system is NonEarth, in which case, a Cartesian operation is performed.

Examples

The following example shows how the **Area( )** function can calculate the area of a single geographic object. Note that the expression *tablename.obj* (as in *states.obj*) represents the geographical object of the current row in the specified table.

```
Dim f_sq_miles As Float
Open Table "states"
Fetch First From states
f_sq_miles = Area(states.obj, "sq mi")
```

You can also use the **Area( )** function within the SQL Select statement, as shown in the following example.

```
Select state, Area(obj, "sq km")
   From states Into results
```

See Also:

ObjectLen( ) function, **Perimeter( ) function**, **CartesianArea( ) function**, **SphericalArea( ) function**, **Set Area Units statement**

AreaOverlap( ) function

Purpose

Returns the area resulting from the overlap of two closed objects. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

AreaOverlap(*object1*, *object2*)

object1 and *object2* are closed objects.

Return Value

A float value representing the area (in MapBasic's current area units) of the overlap of the two objects.

Restrictions

AreaOverlap( ) only works on closed objects. If both objects are not closed (such as points and lines), then you may see an error message. Closed objects are objects that can produce an area, such as regions (polygons).

See Also:

Overlap( ) function, **ProportionOverlap( ) function**, **Set Area Units statement**

Asc( ) function

Purpose

Returns the character code for the first character in a string expression. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Asc(*string_expr*)

string_expr is a string expression.

Return Value

Integer

Description

The **Asc( )** function returns the character code representing the first character in the string specified by *string_expr*.

If *string_expr* is a null string, the **Asc( )** function returns a value of zero.

All MapInfo Professional environments have common character codes within the range of 32 (space) to 126 (tilde).

On a system that supports double-byte character sets (e.g., Windows Japanese): if the first character of *string_expr* is a single-byte character, **Asc( )** returns a number in the range 0 - 255; if the first character of *string_expr* is a double-byte character, **Asc( )** returns a value in the range 256 - 65,535.

On systems that do not support double-byte character sets, **Asc( )** returns a number in the range 0 - 255.

Example

```
Dim code As SmallInt
code = Asc("Afghanistan")
' code will now be equal to 65,
' since 65 is the code for the letter A
```

See Also:

Chr$( ) function

Asin( ) function

Purpose

Returns the arc-sine value of a number. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Asin(*num_expr*)

num_expr is a numeric expression from one to negative one, inclusive.

Return Value

Float

Description

The **Asin( )** function returns the arc-sine of the numeric *num_expr* value. In other words, **Asin( )** returns the angle whose sine is equal to *num_expr*.

The result returned from **Asin( )** represents an angle, expressed in radians. This angle will be somewhere between -Pi/2 and Pi/2 radians (given that Pi is approximately equal to 3.141593, and given that Pi/2 radians represents 90 degrees).

To convert a degree value to radians, multiply that value by DEG_2_RAD. To convert a radian value into degrees, multiply that value by RAD_2_DEG. (Note that your program will need to Include “MAPBASIC.DEF” in order to reference DEG_2_RAD or RAD_2_DEG).

Since sine values range between one and negative one, the expression *num_expr* should represent a value no larger than one (1) and no smaller than negative one (-1).

Example

```
Include "MAPBASIC.DEF"
Dim x, y As Float
x = 0.5
y = Asin(x) * RAD_2_DEG

' y will now be equal to 30,
' since the sine of 30 degrees is 0.5
```

See Also:

Acos( ) function, **Atn( ) function**, **Cos( ) function**, **Sin( ) function**, **Tan( ) function**

Ask( ) function

Purpose

Displays a dialog box, asking the user a yes or no (**OK** or **Cancel**) question. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Ask(*prompt*, *ok_text*, *cancel_text*)

prompt is a string to appear as a prompt in the dialog box.

ok_text is a string (e.g., “OK”) that appears on the confirmation button.

cancel_text is a string (e.g., “Cancel”) that appears on the cancel button.

Return Value

Logical

Description

The **Ask( )** function displays a dialog box, asking the user a yes-or-no question. The prompt parameter specifies a message, such as “File already exists; do you want to continue?” The length of the prompt string passed to the **Ask( )** function can be approximately 2000 characters long, only the first 299 characters will be displayed in the dialog box.

The limit is based on the internal buffer size, the size of the dialog box that is used for the **Ask( )** function, and the size of the static text control in that dialog box.

The dialog box contains two buttons; the user can click one button to give a Yes answer to the prompt, or click the other button to give a No answer. The *ok_text* parameter specifies the name of the Yes-answer button (e.g., “OK” or “Continue”), and the *cancel_text* parameter specifies the name of the No-answer button (e.g., “Cancel” or “Stop”).

If the user selects the *ok_text* button, the **Ask( )** function returns TRUE. If the user clicks the *cancel_text* button or otherwise cancels the dialog box (e.g., by pressing the Esc key), the **Ask( )** function returns FALSE. Since the buttons are limited in size, the *ok_text* and *cancel_text* strings should be brief. If you need to display phrases that are too long to fit in small dialog box buttons, you can use the **Dialog statement** instead of calling the **Ask( )** function. The *ok_text* button is the default button (the button which will be selected if the user presses Enter instead of clicking with the mouse).

Example

```
Dim more As Logical
more = Ask("Do you want to continue?", "OK", "Stop")
```

See Also:

Dialog statement, **Note statement**, **Print statement**

Atn( ) function

Purpose

Returns the arc-tangent value of a number. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Atn(*num_expr*)

num_expr is a numeric expression.

Return Value

Float

Description

The **Atn( )** function returns the arc-tangent of the numeric *num_expr* value. In other words, **Atn( )** returns the angle whose tangent is equal to *num_expr*. The *num_expr* expression can have any numeric value.

The result returned from **Atn( )** represents an angle, expressed in radians, in the range -Pi/2 radians to Pi/2 radians.

To convert a degree value to radians, multiply that value by DEG_2_RAD. To convert a radian value into degrees, multiply that value by RAD_2_DEG. (Note that your program will need to Include “MAPBASIC.DEF” in order to reference DEG_2_RAD or RAD_2_DEG).

Example

```
Include "MAPBASIC.DEF"
Dim val As Float

val = Atn(1) * RAD_2_DEG
'val is now 45, since the
'Arc tangent of 1 is 45 degrees
```

See Also:

Acos( ) function, **Asin( ) function**, **Cos( ) function**, **Sin( ) function**, **Tan( ) function**

AutoLabel statement

Purpose

Draws labels in a Map window, and stores the labels in the Cosmetic layer. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
AutoLabel
   [ Window window_id ]
   [ { Selection | Layer layer_id } ]
   [ Overlap [ { On | Off } ] ]
   [ Duplicates [ { On | Off } ] ]
```

window_id is an integer window identifier for a Map window.

layer_id is a table name (e.g., World) or a SmallInt layer number (e.g., 1 to draw labels for the top layer).

Description

The **AutoLabel** statement draws labels (text objects) in a Map window. Only objects that are currently visible in the Map window are labeled. The **Window** clause controls which Map window is labeled. If you omit the **Window** clause, MapInfo Professional draws labels in the front-most Map window. If you specify **Selection**, only selected objects are labeled. If you omit both the **Selection** and the **Layer** clause, all layers are labeled.

The **Overlap** clause controls whether MapInfo Professional draws labels that overlap other labels. This setting defaults to **Off** (MapInfo Professional will not draw overlapping labels). To force MapInfo Professional to draw a label for every map object, regardless of whether the labels overlap, specify **Overlap On**. The **Duplicates** clause controls whether MapInfo Professional draws a new label for an object that has already been labeled. This setting defaults to **Off** (duplicates not allowed). The **AutoLabel** statement uses whatever font and position settings are in effect. Set label options by choosing **Map > Layer Control**. To control font and position settings through MapBasic, issue a **Set Map statement**.

Example

```
Open Table "world" Interactive
Open Table "worldcap" Interactive
Map From world, worldcap
AutoLabel
   Window FrontWindow( )
   Layer world
```

See Also:

Set Map statement

Beep statement

Purpose

Makes a beeping sound. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Beep
```

Description

The **Beep** statement sends a sound to the speaker.

Browse statement

Purpose

Opens a new Browser window. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Browse expression_list From table
   [ Position ( x, y ) [ Units paper_units ] ]
   [ Width window_width [ Units paper_units ] ]
   [ Height window_height [ Units paper_units ] ]
   [ Row n ]
   [ Column n ]
   [ Min | Max ]
```

expression_list is either an asterisk or a comma-separated list of column expressions.

table is a string representing the name of an open table.

x, *y* specifies the position of the upper left corner of the Browser, in *paper_units*.

paper_units is a string representing a paper unit name (for example, “cm” for centimeters).

window_width and *window_height* specify the size of the Browser, in *paper_units*.

n is a positive integer value.

Description

The **Browse** statement opens a Browse window to display a table.

If the *expression_list* is simply an asterisk (*), the new Browser includes all fields in the table. Alternately, the *expression_list* clause can consist of a comma-separated list of expressions, each of which defines one column that is to appear in the Browser. Expressions in the list can contain column names, operators, functions, and variables. Each column's name is derived from the

expression that defines the column. Thus, if a column is defined by the expression *population / area(obj, "acre")*, that expression will appear on the top row of the Browser, as the column name. To assign an alias to an expression, follow the expression with a string; see the example below.

An optional **Position** clause lets you specify where on the screen to display the Browser. The x coordinate specifies the distance (in paper units) from the left edge of the MapInfo Professional application window to the left edge of the Browser. The y coordinate specifies the distance from the top of the MapInfo Professional window down to the top of the Browser. The optional **Width** and **Height** clauses specify the size of the Browser window, in paper units. If no **Width** and **Height** clauses are provided, MapInfo Professional assigns the Browser window a default size which depends on the table in question: the Browser height will generally be one quarter of the screen height, unless the table does not have enough rows to fill a Browser window that large; and the Browser width will depend on the widths of the fields in the table.

If the **Browse** statement includes the optional **Max** keyword, the resultant Browser window is maximized, taking up all of the screen space available to MapInfo Professional. Conversely, if the **Browse** statement includes the **Min** keyword, the Browser window is minimized immediately.

The **Row** clause dictates which row of the table should appear at the top of the Browser. If the **Browse** statement does not include a **Row** clause, the first row of the table will be the top row in the Browser.

Similarly, the **Column** clause dictates which of the table's columns should appear at the left edge of the Browser. If the **Browse** statement does not include a **Column** clause, the table's first column will appear at the left edge of the Browser window.

Example

The following example opens the World table and displays all columns from the table in a Browser window.

```
Open Table "world"
Browse * From world
```

The next example specifies exactly which column expressions from the World table should be displayed in the Browser.

```
Open Table "world"
Browse
   country,
   population,
   population/area(obj, "sq km") "Density"
   From world
```

The resultant Browser has three columns. The first two columns represent data as it is stored in the World table, while the third column is derived. Through the third expression, MapBasic divides the population of each country record with the geographic area of the region associated with that record. The derived column expression has an alias ("Density") which appears on the top row of the Browse window.

See Also:

Set Browse statement, **Set Window statement**

BrowserInfo function

Purpose

Returns information about a Browser window, such as: the total number of rows or columns in the Browser window; or the row number, column number, or value contained in the current cell.

You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

BrowserInfo(*window_id*, *attribute*)

window_id is an integer window identifier.

attribute is an integer code indicating what type of information to return. For values, see the table later in this description.

Return Value

Float, logical, or string depending on the attribute parameter.

Description

The BrowserInfo( ) function returns information about a Browser window. The function does not apply to the Redistricter window.

The *window_id* parameter specifies which Browser window to query. To obtain a window identifier, call the **FrontWindow( ) function** immediately after opening a window, or call the **WindowID( ) function** at any time after the window's creation.

There are several attributes that BrowserInfo( ) returns about any given Browser window. The attribute parameter tells the BrowserInfo ( ) function what Browser window statistic to return. The attribute parameter should be one of the codes from the following table; codes are defined in MAPBASIC.DEF.

attribute parameter	ID	Return Value
BROWSER_INFO_NROWS	1	The total number of rows in the Browser window.
BROWSER_INFO_NCOLS	2	The total number of columns in the Browser window.
BROWSER_INFO_CURRENT_ROW	3	The row number of the current cell in the Browser window. Row numbers start at one (1).

attribute parameter	ID	Return Value
BROWSER_INFO_CURRENT_COLUMN	4	The column number of the current cell in the Browser window. Column numbers start at zero (0).
BROWSER_INFO_CURRENT_CELL_VALUE	5	The value contained in the current cell in the Browser window.

Error Conditions

ERR_BAD_WINDOW (590) error generated if parameter is not a valid window number.

ERR_FCN_ARG_RANGE (644) error generated if an argument is outside of the valid range.

ERR_WANT_BROWSER_WIN (312) error generated if window id is not a Browser window.

See Also:

FrontWindow( ) function, **WindowID( ) function**

Brush clause

Purpose

Specifies a fill style for graphic objects. You can use this clause in the MapBasic Window in MapInfo Professional.

Syntax

Brush *brush_expr*

brush_expr is a Brush expression, such as **MakeBrush**(*pattern*, *fgcolor*, *bgcolor*). (See **MakeBrush( ) function** for more information.) or a Brush variable.

Description

The **Brush** clause specifies a brush style—in other words, a set of color and pattern settings that dictate the appearance of a filled object, such as a circle or rectangle. **Brush** is a clause, not a complete MapBasic statement. Various object-related statements, such as **Create Ellipse statement**, allow you to specify a brush value. The keyword **Brush** may be followed by an expression which evaluates to a Brush value. This expression can be a Brush variable:

```
Brush br_var
```

or a call to a function which returns a Brush value:

```
Brush MakeBrush(64, CYAN, BLUE)
```

With some MapBasic statements (e.g., **Set Map statement**), the keyword **Brush** can be followed immediately by the three parameters that define a Brush style (pattern, foreground color, and background color) within parentheses:

```
Brush(64, CYAN, BLUE)
```

Some MapBasic statements take a Brush expression as a parameter (e.g., the name of a Brush variable), rather than a full **Brush** clause (the keyword **Brush** followed by the name of a Brush variable). The **Alter Object statement** is one example.

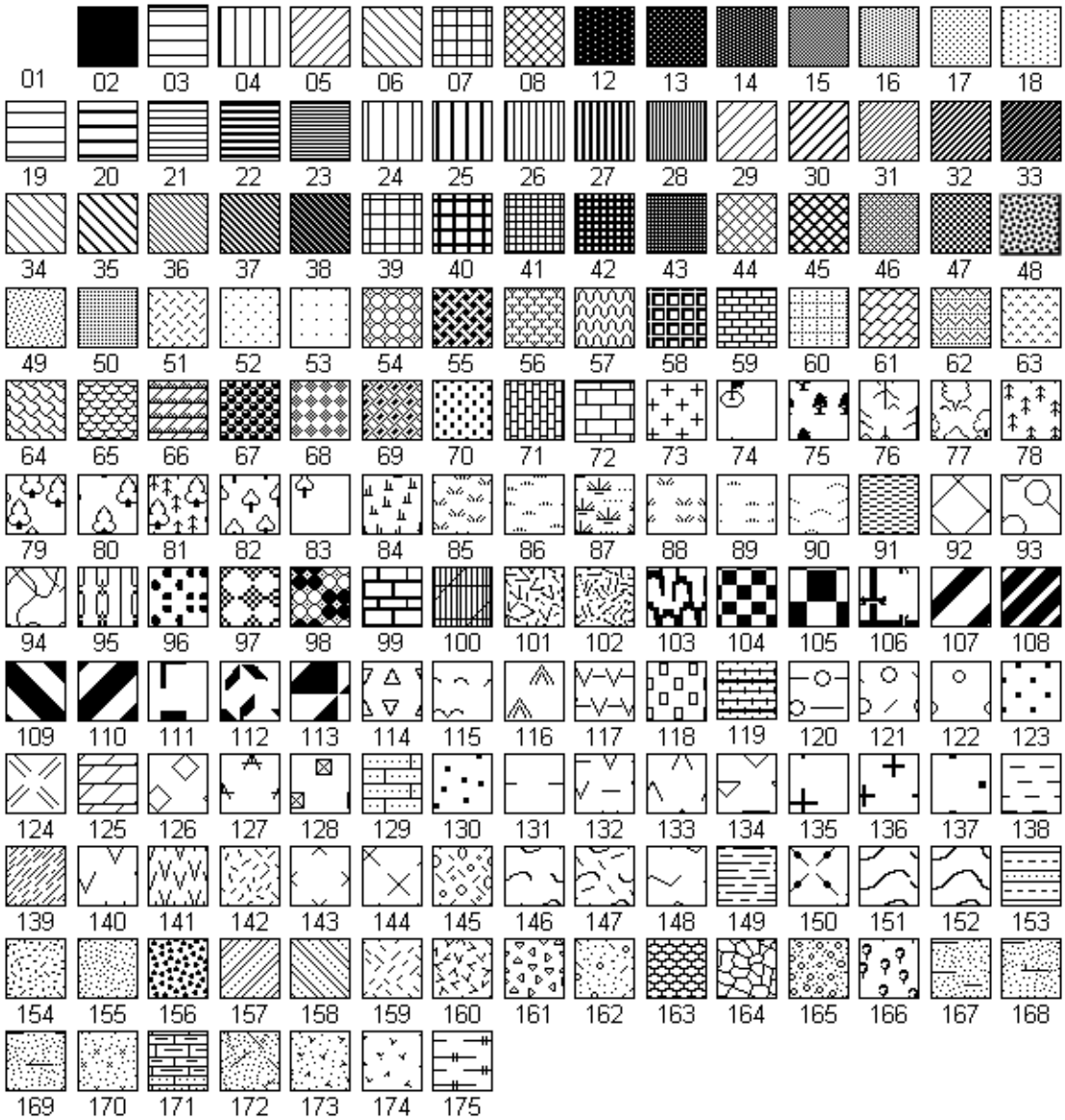
The following table summarizes the three components (pattern, foreground color, background color) that define a Brush:

Component	Description
pattern	Integer value from 1 to 8 or from 12 to 71; see table below.
foreground color	Integer RGB color value; see **RGB( ) function**. The definitions file, MAPBASIC.DEF, includes Define statements for BLACK, WHITE, RED, GREEN, BLUE, CYAN, MAGENTA, and YELLOW.
background color	Integer RGB color value.

To specify a transparent background, use pattern 3 or larger, and omit the background color from the **Brush** clause. For example, specify *Brush(5, BLUE)* to see thin blue stripes with no background fill color. Omitting the background parameter is like clearing the **Background** check box in MapInfo Professional's Region Style dialog box.

To specify a transparent background when calling the **MakeBrush( ) function** specify -1 as the background color.

The available patterns appear as follows. Pattern 2 produces a solid fill; pattern 1 produces no fill.



For a comprehensive list of fill patterns, see the MapInfo Professional *Help System*—launch MapInfo Professional and from the **Help** menu, select **MapInfo Professional Help Topics** and then search for *MapInfo Professional Fill Pattern Table*.

See Also:

CurrentBrush( ) function, **MakeBrush( ) function**, **Pen clause**, **Font clause**, **Symbol clause**

Buffer( ) function

Purpose

Returns a region object that represents a buffer region (the area within a specified buffer distance of an existing object). You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Buffer(*inputobject*, *resolution*, *width*, *unit_name*)

inputobject is an object expression.

resolution is a SmallInt value representing the number of nodes per circle at each corner.

width is a float value representing the radius of the buffer; if *width* is negative, and if *inputobject* is a closed object, the object returned represents an object smaller than the original object. If the *width* is negative, and the object is a linear object (line, polyline, arc) or a point, then the absolute value of *width* is used to produce a positive buffer.

unit_name is the name of the distance unit (e.g., “mi” for miles, “km” for kilometers) used by *width*.

Return Value

Returns a region object.

Description

The **Buffer( )** function returns a region representing a buffer.

The **Buffer( )** function operates on one single object at a time. To create a buffer around a set of objects, use the **Create Object statement** *As Buffer*. The object will be created using the current MapBasic coordinate system. The method used to calculate the buffer depends on the coordinate system. If it is NonEarth, then a Cartesian method will be used. Otherwise, a spherical method will be used.

Example

The following program creates a line object, then creates a buffer region surrounding the line. The buffer region extends ten miles in all directions from the line.

```
Dim o_line, o_region As Object
o_line = CreateLine(-73.5, 42.5, -73.6, 42.8)
o_region = Buffer( o_line, 20, 10, "mi")
```

See Also:

Create Object statement

ButtonPadInfo( ) function

Purpose

Returns information about a ButtonPad. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

ButtonPadInfo(*pad_name*, *attribute*)

pad_name is a string representing the name of an existing ButtonPad; use “Main”, “Drawing”, “Tools” or “Standard” to query the standard pads, or specify the name of a custom pad.

attribute is a code indicating which information to return; see table below.

Return Value

Depends on the *attribute* parameter specified.

Description

The *attribute* parameter specifies what to return. Codes are defined in MAPBASIC.DEF

***attribute* code**	**ID**	**ButtonPadInfo( ) returns:**
BTNPAD_INFO_FLOATING	1	Logical. TRUE means the pad is floating, FALSE means the pad is docked.
BTNPAD_INFO_WIDTH	2	SmallInt: The width of the pad, expressed as a number of buttons (not including separators).
BTNPAD_INFO_NBTNS	3	SmallInt. The number of buttons on the pad.
BTNPAD_INFO_X	4	A number indicating the x-position of the upper-left corner of the pad. If the pad is docked, this is an integer. The value is negative when a toolbar is docked to the left of the menu bar. If the pad is floating, this is a float value, in paper units such as inches.
BTNPAD_INFO_Y	5	A number indicating the y-position of the upper-left corner of the pad. This value is negative when a toolbar is docked above the menu bar.

attribute code	ID	ButtonPadInfo( ) returns:
BTNPAD_INFO_WINID	6	Integer. The window ID of the specified pad.
BTNPAD_INFO_DOCK_POSITION	7	Returns the position of the button pad as **Floating**, **Left**, **Top**, **Right** or **Bottom**. Use for floating toolbars as an alternative to BTNPAD_INFO_FLOATING. Returns: • BTNPAD_INFO_DOCK_NONE (0) • BTNPAD_INFO_DOCK_LEFT (1) • BTNPAD_INFO_DOCK_TOP (2) • BTNPAD_INFO_DOCK_RIGHT (3) • BTNPAD_INFO_DOCK_BOTTOM (4)

Example

```
Include "mapbasic.def"
If ButtonPadInfo("Main", BTNPAD_INFO_FLOATING) Then
   '...then the Main pad is floating; now let's dock it.
   Alter ButtonPad "Main" ToolbarPosition(0,0) Fixed
End If
```

See Also:

Alter ButtonPad statement

Call statement

Purpose

Calls a sub procedure or an external routine (DLL, XCMD).

Restrictions

You cannot issue a **Call** statement through the MapBasic window.

Syntax

Call *subproc* [([*parameter*] [, ...])]

subproc is the name of a sub procedure.

parameter is a parameter expression to pass to the sub procedure.

Description

The **Call** statement calls a procedure. The procedure is usually a conventional MapBasic sub procedure (defined through the **Sub…End Sub statement**). Alternately, a program running under MapInfo Professional can call a Windows Dynamic Link Library (DLL) routine through the **Call** statement.

When a **Call** statement calls a conventional MapBasic procedure, MapBasic begins executing the statements in the specified sub procedure, and continues until encountering an **End Sub** or an **Exit Sub statement**. At that time, MapBasic returns from the sub procedure, then executes the statements following the **Call** statement. The **Call** statement can only access sub procedures which are part of the same application.

A MapBasic program must issue a **Declare Sub statement** to define the name and parameter list of any procedure which is to be called. This requirement is independent of whether the procedure is a conventional MapBasic Sub procedure, a DLL procedure or an XCMD.

Parameter Passing

Sub procedures may be defined with no parameters. If a particular sub procedure has no parameters, then calls to that sub procedure may appear in either of the following forms:

Call *subroutine*

or

Call *subroutine***( )**

By default, each sub procedure parameter is defined “by reference.” When a sub procedure has a by-reference parameter, the caller must specify the name of a variable to pass as the parameter.

If the procedure then alters the contents of the by-reference parameter, the caller's variable is automatically updated to reflect the change. This allows the caller to examine the results returned by the sub procedure.

Alternately, any or all sub procedure parameters may be passed “by value” if the keyword **ByVal** appears before the parameter name in the **Sub** and **Declare Sub** declarations. When a parameter is passed by value, the sub procedure receives a copy of the value of the parameter expression; thus, the caller can pass any expression, rather than having to pass the name of a variable.

A sub procedure can take an entire array as a single parameter. When a sub procedure expects an array as a parameter, the caller should specify the name of an array variable, without parentheses.

Calling External Routines

When a **Call** statement calls a DLL routine, MapBasic executes the routine until the routine returns. The specified DLL routine is actually located in a separate file (e.g., “KERNEL.EXE”). The specified DLL file must be present at run-time for MapBasic to complete a DLL Call.

Similarly, if a **Call** statement calls an XCMD, the file containing the XCMD must be present at run-time. When calling XCMDs, you cannot specify array variables or variables of custom data Types as parameters.

Example

In the following example, the sub procedure Cube cubes a number (raises the number to the power of three), and returns the result. The sub procedure takes two parameters; the first parameter contains the number to be cubed, and the second parameter passes the results back to the caller.

```
Declare Sub Cube(ByVal original As Float, cubed As Float)
   Dim x, result As Float
```

```
   Call Cube( 2, result)
   ' result now contains the value: 8 (2 x 2 x 2)
   x = 1
   Call Cube( x + 2, result)
   ' result now contains the value: 27 (3 x 3 x 3)
End Program
Sub Cube (ByVal original As Float, cubed As Float)
   ' Cube the "original" parameter, and store
   ' the result in the "cubed" parameter.
   cubed = original ^ 3
End Sub
```

See Also:

Declare Sub statement, **Exit Sub statement**, **Global statement**, **Sub…End Sub statement**

CartesianArea( ) function

Purpose

Returns the area as calculated in a flat, projected coordinate system using a Cartesian algorithm. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

CartesianArea(*obj_expr*, *unit_name*)

obj_expr is an object expression.

unit_name is a string representing the name of an area unit (e.g., “sq km”).

Return Value

Float

Description

The **CartesianArea( )** function returns the Cartesian area of the geographical object specified by *obj_expr*.

The function returns the area measurement in the units specified by the *unit_name* parameter; for example, to obtain an area in acres, specify “acre” as the *unit_name* parameter. See the **Set Area Units statement** for the list of available unit names.

The **CartesianArea( )** function will always return the area using a cartesian algorithm. A value of -1 will be returned for data that is in a Latitude/Longitude since the data is not projected.

Only regions, ellipses, rectangles, and rounded rectangles have any area. By definition, the **CartesianArea( )** of a point, arc, text, line, or polyline object is zero. The **CartesianArea( )** function returns approximate results when used on rounded rectangles. MapBasic calculates the area of a rounded rectangle as if the object were a conventional rectangle.

Examples

The following example shows how the **CartesianArea( )** function can calculate the area of a single geographic object. Note that the expression *tablename*.obj (as in states.obj) represents the geographical object of the current row in the specified table.

```
Dim f_sq_miles As Float
Open Table "counties"
Fetch First From counties
f_sq_miles = CartesianArea(counties.obj, "sq mi")
```

You can also use the **CartesianArea( )** function within the **Select statement**, as shown in the following example.

```
Select lakes, CartesianArea(obj, "sq km")
   From lakes Into results
```

See Also:

Area( ) function, **SphericalArea( ) function**

CartesianBuffer( ) function

Purpose

Returns a region object that represents a buffer region (the area within a specified buffer distance of an existing object). You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

CartesianBuffer(*inputobject*, *resolution*, *width*, *unit_name*)

inputobject is an object expression.

resolution is a SmallInt value representing the number of nodes per circle at each corner.

width is a float value representing the radius of the buffer; if width is negative, and if inputobject is a closed object, the object returned represents an object smaller than the original object.

unit_name is the name of the distance unit (e.g., “mi” for miles, “km” for kilometers) used by *width*.

Return Value

Region Object

Description

The **CartesianBuffer( )** function returns a region representing a buffer and operates on one single object at a time.

To create a buffer around a set of objects, use the **Create Object statement** *As Buffer*. If *width* is negative, and the object is a linear object (line, polyline, arc) or a point, then the absolute value of *width* is used to produce a positive buffer.

The **CartesianBuffer( )** function calculates the buffer by assuming the object is in a flat projection and using the *width* to calculate a cartesian distance calculated buffer around the object.

If the *inputobject* is in a Latitude/Longitude Projection, then Spherical calculations will be used regardless of the Buffer function used.

Example

The following program creates a line object, then creates a buffer region that extends 10 miles surrounding the line.

```
Dim o_line, o_region As Object
o_line = CreateLine(-73.5, 42.5, -73.6, 42.8)
o_region = CartesianBuffer( o_line, 20, 10, "mi")
```

See Also:

Buffer( ) function, **Creating Map Objects**

CartesianConnectObjects( ) function

Purpose

Returns an object representing the shortest or longest distance between two objects. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

CartesianConnectObjects(*object1*, *object2*, *min*)

object1 and *object2* are object expressions.

min is a logical expression where TRUE calculates the minimum distance between the objects, and FALSE calculates the maximum distance between objects.

Return Value

This statement returns a single section, two-point Polyline object representing either the closest distance (*min* == *TRUE*) or farthest distance (*min* == *FALSE*) between *object1* and *object2*.

Description

One point of the resulting Polyline object is on *object1* and the other point is on *object2*. Note that the distance between the two input objects can be calculated using the **ObjectLen( ) function**. If there are multiple instances where the minimum or maximum distance exists (e.g., the two points returned are not uniquely the shortest distance and there are other points representing “ties”) then these functions return one of the instances. There is no way to determine if the object returned is uniquely the shortest distance.

CartesianConnectObjects( ) returns a Polyline object connecting *object1* and *object2* in the shortest (*min* == *TRUE*) or longest (*min* == *FALSE*) way using a cartesian calculation method. If the calculation cannot be done using a cartesian distance method (e.g., if the MapBasic Coordinate System is Lat/Long), then this function will produce an error.

CartesianDistance( ) function

Purpose

Returns the distance between two locations. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

CartesianDistance(*x1*, *y1*, *x2*, *y2*, *unit_name*)

x1 and *x2* are x-coordinates.

y1 and *y2* are y-coordinates.

unit_name is a string representing the name of a distance unit (e.g., “km”).

Return Value

Float

Description

The **CartesianDistance( )** function calculates the Cartesian distance between two locations. It returns the distance measurement in the units specified by the *unit_name* parameter; for example, to obtain a distance in miles, specify “mi” as the *unit_name* parameter. See **Set Distance Units statement** for the list of available unit names.

The **CartesianDistance( )** function always returns a value using a cartesian algorithm. A value of -1 is returned for data that is in a Latitude/Longitude coordinate system, since Latitude/Longitude data is not projected and not cartesian.

The x- and y-coordinate parameters must use MapBasic's current coordinate system. By default, MapInfo Professional expects coordinates to use a Latitude/Longitude coordinate system. You can reset MapBasic's coordinate system through the **Set CoordSys statement**.

Example

```
Dim dist, start_x, start_y, end_x, end_y As Float
Open Table "cities"
Fetch First From cities
start_x = CentroidX(cities.obj)
start_y = CentroidY(cities.obj)
Fetch Next From cities
end_x = CentroidX(cities.obj)
end_y = CentroidY(cities.obj)
dist = CartesianDistance(start_x,start_y,end_x,end_y,"mi")
```

See Also:

Math Functions, **CartesianDistance( ) function**, **Distance( ) function**

CartesianObjectDistance( ) function

Purpose

Returns the distance between two objects. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

CartesianObjectDistance(*object1, object2, unit_name*)

object1 and *object2* are object expressions.

unit_name is a string representing the name of a distance unit.

Return Value

Float

Description

CartesianObjectDistance( ) returns the minimum distance between *object1* and *object2* using a cartesian calculation method with the return value in *unit_name*. If the calculation cannot be done using a cartesian distance method (e.g., if the MapBasic Coordinate System is Lat/Long), then this function will produce an error.

CartesianObjectLen( ) function

Purpose

Returns the geographic length of a line or polyline object. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

CartesianObjectLen(*obj_expr, unit_name*)

obj_expr is an object expression.

unit_name is a string representing the name of a distance unit (e.g., “km”).

Return Value

Float

Description

The **CartesianObjectLen( )** function returns the length of an object expression. Note that only line and polyline objects have length values greater than zero; to measure the circumference of a rectangle, ellipse, or region, use the **Perimeter( ) function**.

The **CartesianObjectLen( )** function will always return a value using a cartesian algorithm. A value of -1 will be returned for data that is in a Latitude/Longitude coordinate system, since Latitude/Longitude data is not projected and not cartesian.

The **CartesianObjectLen( )** function returns a length measurement in the units specified by the *unit_name* parameter; for example, to obtain a length in miles, specify “mi” as the *unit_name* parameter. See the **Set Distance Units statement** for the list of valid unit names.

Example

```
Dim geogr_length As Float
Open Table "streets"
Fetch First From streets
geogr_length = CartesianObjectLen(streets.obj, "mi")
' geogr_length now represents the length of the
' street segment, in miles
```

See Also:

SphericalObjectLen( ) function, **CartesianObjectLen( ) function**, **ObjectLen( ) function**

CartesianOffset( ) function

Purpose

Returns a copy of the input object offset by the specified distance and angle using a Cartesian DistanceType. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

CartesianOffset(*object*, *angle*, *distance*, *units*)

object is the object being offset.

angle is the angle to offset the object.

distance is the distance to offset the object.

units is a string representing the unit in which to measure *distance*.

Return Value

Object

Description

This function produces a new object that is a copy of the input object offset by distance along angle (in degrees with horizontal in the positive X-axis being 0 and positive being counterclockwise). The unit string, similar to that used for **ObjectLen( ) function** or **Perimeter( ) function**, is the unit for the distance value. The DistanceType used is Cartesian. If the coordinate system of the input object is Lat/Long, an error will occur, since Cartesian DistanceTypes are not valid for Lat/Long. This is signified by returning a NULL object. The coordinate system used is the coordinate system of the input object.

There are some considerations for Spherical measurements that do not hold for Cartesian measurements. If you move an object that is in Lat/Long, the shape of the object remains the same, but the area of the object will change. This is because you are picking one offset delta in degrees, and the actual measured distance for a degree is different at different locations.

For the Offset functions, the actual offset delta is calculated at some fixed point on the object (e.g., the center of the bounding box), and then that value is converted from the input units into the coordinate system's units. If the coordinate system is Lat/Long, the conversion to degrees uses the fixed point. The actual converted distance measurement could vary at different locations on the object. The distance from the input object and the new offset object is only guaranteed to be exact at the single fixed point used.

Example

```
CartesianOffset(Rect, 45, 100, "mi")
```

See Also:

CartesianOffsetXY( ) function

CartesianOffsetXY( ) function

Purpose

Returns a copy of the input object offset by the specified X and Y offset values using a cartesian DistanceType. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

CartesianOffsetXY(*object*, *xoffset*, *yoffset*, *units*)

object is the object being offset.

xoffset and *yoffset* are the distance along the x and y axes to offset the object.

units is a string representing the unit in which to measure distance.

Return Value

Object

Description

This function produces a new object that is a copy of the input object offset by xoffset along the X-axis and yoffset along the Y-axis. The unit string, similar to that used for **ObjectLen( ) function** or **Perimeter( ) function**, is the unit for the distance values. The DistanceType used is Cartesian. If the coordinate system of the input object is Lat/Long, an error will occur, since Cartesian DistanceTypes are not valid for Lat/Long. This is signified by returning a NULL object. The coordinate system used is the coordinate system of the input object.

There are some considerations for Spherical measurements that do not hold for Cartesian measurements. If you move an object that is in Lat/Long, the shape of the object remains the same, but the area of the object will change. This is because you are picking one offset delta in degrees, and the actual measured distance for a degree is different at different locations.

For the Offset functions, the actual offset delta is calculated at some fixed point on the object (e.g., the center of the bounding box), and then that value is converted from the input units into the coordinate system's units. If the coordinate system is Lat/Long, the conversion to degrees uses the fixed point. The actual converted distance measurement could vary at different locations on the object. The distance from the input object and the new offset object is only guaranteed to be exact at the single fixed point used.

Example

```
CartesianOffset(Rect, 45, 100, "mi")
```

See Also:

CartesianOffset( ) function

CartesianPerimeter( ) function

Purpose

Returns the perimeter of a graphical object. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

CartesianPerimeter(*obj_expr* , *unit_name*)

obj_expr is an object expression.

unit_name is a string representing the name of a distance unit (e.g., “km”).

Return Value

Float

Description

The **CartesianPerimeter( )** function calculates the perimeter of the *obj_expr* object. The **Perimeter( ) function** is defined for the following object types: ellipses, rectangles, rounded rectangles, and polygons. Other types of objects have perimeter measurements of zero.

The **CartesianPerimeter( )** function will always return a value using a Cartesian algorithm. A value of -1 will be returned for data that is in a Latitude/Longitude coordinate system, since Latitude/Longitude data is not projected and not Cartesian.

The **CartesianPerimeter( )** function returns a length measurement in the units specified by the *unit_name* parameter; for example, to obtain a length in miles, specify “mi” as the *unit_name* parameter. See the **Set Distance Units statement** for the list of valid unit names. **CartesianPerimeter( )** returns approximate results when used on rounded rectangles. MapBasic calculates the perimeter of a rounded rectangle as if the object were a conventional rectangle.

Example

The following example shows how you can use the **CartesianPerimeter( )** function to determine the perimeter of a particular geographic object.

```
Dim perim As Float
Open Table "world"
Fetch First From world
perim = CartesianPerimeter(world.obj, "km")
' The variable perim now contains
' the perimeter of the polygon that's attached to
' the first record in the World table.
```

You can also use the **CartesianPerimeter( )** function within the **Select statement**. The following **Select statement** extracts information from the States table, and stores the results in a temporary table called Results. Because the **Select statement** includes the **CartesianPerimeter( )** function, the Results table will include a column showing each state's perimeter.

```
Open Table "states"
Select state, CartesianPerimeter(obj, "mi")
   From states
   Into results
```

See Also:

CartesianPerimeter( ) function, **SphericalPerimeter( ) function**, **Perimeter( ) function**

Centroid( ) function

Purpose

Returns the centroid (center point) of an object. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Centroid(*obj_expr*)

obj_expr is an object expression.

Return Value

Point object

Description

The **Centroid( )** function returns a point object, which is located at the centroid of the specified *obj_expr* object. A region's centroid does not represent its center of mass. Instead, it represents the location used for automatic labeling, geocoding, and placement of thematic pie and bar charts. If you edit a map in reshape mode, you can reposition region centroids by dragging them.

If the *obj_expr* parameter represents a point object, the **Centroid( )** function returns the position of the point. If the *obj_expr* parameter represents a line object, the **Centroid( )** function returns the point midway between the ends of the line.

If the *obj_expr* parameter represents a polyline object, the **Centroid( )** function returns a point located at the mid point of the middle segment of the polyline.

If the *obj_expr* parameter represents any other type of object, the **Centroid( )** function returns a point located at the true centroid of the original object. For rectangle, arc, text, and ellipse objects, the centroid position is halfway between the upper and lower extents of the object, and halfway between the left and right extents. For region objects, however, the centroid position is always on the object in question, and therefore may not be located halfway between the object's extents.

Example

```
Dim pos As Object
Open Table "world"
Fetch First From world
pos = Centroid(world.obj)
```

See Also:

Alter Object statement, **CentroidX( ) function**, **CentroidY( ) function**

CentroidX( ) function

Purpose

Returns the x-coordinate of the centroid of an object. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

CentroidX(*obj_expr*)

obj_expr is an object expression

Return Value

Float

Description

The **CentroidX( )** function returns the X coordinate (e.g., Longitude) component of the centroid of the specified object. See the **Centroid( ) function** for a discussion of what the concept of a centroid position means with respect to different types of graphical objects (lines vs. regions, etc.).

The coordinate information is returned in MapBasic's current coordinate system; by default, MapBasic uses a Longitude/Latitude coordinate system. The **Set CoordSys statement** allows you to change the coordinate system used.

Examples

The following example shows how the **CentroidX( )** function can calculate the longitude of a single geographic object.

```
Dim x As Float
Open Table "world"
Fetch First From world
x = CentroidX(world.obj)
```

You can also use the **CentroidX( )** function within the **Select statement**. The following **Select statement** extracts information from the World table, and stores the results in a temporary table called Results. Because the **Select statement** includes the **CentroidX( )** function and the **CentroidY( ) function**, the Results table will include columns which display the longitude and latitude of the centroid of each country.

```
Open Table "world"
Select country, CentroidX(obj), CentroidY(obj)
   From world Into results
```

See Also:

Centroid( ) function, **CentroidY( ) function**, **Set CoordSys statement**

CentroidY( ) function

Purpose

Returns the y-coordinate of the centroid of an object. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

CentroidY(*obj_expr*)

obj_expr is an object expression.

Return Value

Float

Description

The **CentroidY( )** function returns the Y-coordinate (e.g., latitude) component of the centroid of the specified object. See the **Centroid( ) function** for a discussion of what the concept of a centroid position means, with respect to different types of graphical objects (lines vs. regions, etc.).

The coordinate information is returned in MapBasic's current coordinate system; by default, MapBasic uses a Longitude/Latitude coordinate system. The **Set CoordSys statement** allows you to change the coordinate system used.

Example

```
Dim y As Float
Open Table "world"
Fetch First From world
y = CentroidY(world.obj)
```

See Also:

Centroid( ) function, **CentroidX( ) function**, **Set CoordSys statement**

CharSet clause

Purpose

Specifies which character set MapBasic uses for interpreting character codes.

See the MapInfo Professional *User Guide* documentation for changes affecting this clause.

Syntax

CharSet *char_set*

char_set is a string that identifies the name of a character set; see table below.

Description

The **CharSet** clause specifies which character set MapBasic should use when reading or writing files or tables. Note that **CharSet** is a clause, not a complete statement. Various file-related statements, such as the **Open File statement**, can incorporate optional **CharSet** clauses.

What Is A Character Set?

Every character on a computer keyboard corresponds to a numeric code. For example, the letter “A” corresponds to the character code 65. A character set is a set of characters that appear on a computer, and a set of numeric codes that correspond to those characters.

Different character sets are used in different countries. For example, in the version of Windows for North America and Western Europe, character code 176 corresponds to a degrees symbol; however, if Windows is configured to use a different character set, character code 176 may represent a different character.

Call *SystemInfo(SYS_INFO_CHARSET)* to determine the character set in use at run-time.

How Do Character Sets Affect MapBasic Programs?

If your files use only standard ASCII characters in the range of 32 (space) to 126 (tilde), you do not need to worry about character set conflicts, and you do not need to use the **CharSet** clause.

Even if your files include “special” characters (for example, characters outside the range 32 to 126), if you do all of your work within one environment (e.g., Windows) using only one character set, you do not need to use the **CharSet** clause.

If your program needs to read an existing file that contains “special” characters, and if the file was created in a character set that does not match the character set in use when you run your program, your program should use the **CharSet** clause. The **CharSet** clause should indicate what character set was in use when the file was created.

The **CharSet** clause takes one parameter: a string expression which identifies the name of the character set to use. The following table lists all character sets available.

Character Set	Comments
“Neutral”	No character conversions performed.
“ISO8859_1”	ISO 8859-1 (UNIX)
“ISO8859_2”	ISO 8859-2 (UNIX)
“ISO8859_3”	ISO 8859-3 (UNIX)
“ISO8859_4”	ISO 8859-4 (UNIX)
“ISO8859_5”	ISO 8859-5 (UNIX)
“ISO8859_6”	ISO 8859-6 (UNIX)
“ISO8859_7”	ISO 8859-7 (UNIX)
“ISO8859_8”	ISO 8859-8 (UNIX)
“ISO8859_9”	ISO 8859-9 (UNIX)
“PackedEUCJapanese”	UNIX, standard Japanese implementation.
“WindowsLatin2” “WindowsArabic” “WindowsCyrillic” “WindowsGreek” “WindowsHebrew” “WindowsTurkish”	Windows Eastern Europe
“WindowsTradChinese”	Windows Traditional Chinese
“WindowsSimpChinese”	Windows Simplified Chinese
“WindowsJapanese”	
“WindowsKorean”	
“CodePage437”	DOS Code Page 437 = IBM Extended ASCII
“CodePage850”	DOS Code Page 850 = Multilingual
“CodePage852”	DOS Code Page 852 = Eastern Europe
“CodePage855”	DOS Code Page 855 = Cyrillic
“CodePage857”	

Character Set	Comments
“CodePage860”	DOS Code Page 860 = Portuguese
“CodePage861”	DOS Code Page 861 = Icelandic
“CodePage863”	DOS Code Page 863 = French Canadian
“CodePage864”	DOS Code Page 864 = Arabic
“CodePage865”	DOS Code Page 865 = Nordic
“CodePage869”	DOS Code Page 869 = Modern Greek
“LICS”	Lotus worksheet release 1,2 character set
“LMBCS”	Lotus worksheet release 3,4 character set

You never need to specify a CharSet clause in an **Open Table statement**. Each table's .TAB file contains information about the character set used by the table. When opening a table, MapInfo Professional reads the character set information directly from the .TAB file, then automatically performs any necessary character translations.

To force MapInfo Professional to save a table in a specific character set, include a **CharSet** clause in the **Commit Table statement**.

See Also:

Commit Table statement, **Create Table statement**, **Export statement**, **Open File statement**, **Register Table statement**

ChooseProjection$( ) function

Purpose

Displays the Choose Projection dialog box and returns the coordinate system selected by the user. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

ChooseProjection$(*initial_coordsys*, *get_bounds*)

initial_coordsys is a string value in the form of a **CoordSys clause**. It is used to set which coordinate system is selected when the dialog box is first displayed. If *initial_coordsys* is empty or an invalid CoordSys clause, then the default Longitude/Latitude coordinate system is used as the initial selection.

get_bounds is a logical value that determines whether the users is prompted for boundary values when a non-earth projection is selected. If *get_bounds* is true then the boundary dialog box is displayed. If false, then the dialog box is not displayed and the default boundary is used.

Description

This function displays the Choose Projection dialog box and returns the selected coordinate system as a string. The returned string is in the same format as the CoordSys clause. Use this function if you wish to allow the user to set a projection within your application.

Example

```
Dim strNewCoordSys As String
strNewCoordSys = ChooseProjection$( "", True)
strNewCoordSys = "Set " + strNewCoordSys
Run Command strNewCoordSys
```

See Also:

MapperInfo( ) function

Chr$( ) function

Purpose

Returns a one-character string corresponding to a specified character code. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Chr$(*num_expr*)

num_expr is an integer value from 0 to 255 (or, if a double-byte character set is in use, from 0 to 65,535), inclusive.

Return Value

String

Description

The **Chr$( )** function returns a string, one character long, based on the character code specified in the num_expr parameter. On most systems, *num_expr* should be a positive integer value between 0 and 255. On systems that support double-byte character sets (e.g., Windows Japanese), *num_expr* can have a value from 0 to 65,535.

All MapInfo Professional environments have common character codes within the range of 32 (space) to 126 (tilde).

If the *num_expr* parameter is fractional, MapBasic rounds to the nearest integer.

Character 12 is the form-feed character. Thus, you can use the statement *Print Chr$(12)* to clear the Message window. Character 10 is the line-feed character; see example below.

Character 34 is the double-quotation mark ("). If a string expression includes the function call *Chr$(34)*, MapBasic embeds a double-quote character in the string.

Error Conditions

ERR_FCN_ARG_RANGE (644) error is generated if an argument is outside of the valid range.

Example

```
Dim s_letter As String * 1
s_letter = Chr$(65)
Note s_letter ' This displays the letter "A"
Note "This message spans" + Chr$(10) + "two lines."
```

See Also:

Asc( ) function

Close All statement

Purpose

Closes all open tables. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Close All [**Interactive**]

Description

If a MapBasic application issues a **Close All** statement, and the affected table has edits pending (the table has been modified but the modifications have not yet been saved to disk), the edits will be discarded before the table is closed. No warning will be displayed. If you do not want to discard pending edits, use the optional **Interactive** clause to prompt the user to save or discard changes.

See Also:

Close Table statement

Close Connection statement

Purpose

Closes a connection opened with the **Open Connection statement**. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Close Connection *connection_handle*

connection_handle is an integer expression representing the value returned from the **Open Connection statement**.

Description

The Close Connection statement closes the specified connection using the connection handle that is returned from an **Open Connection statement**. Any service specific properties associated with the connection are lost.

See Also:

Open Connection statement

Close File statement

Purpose

Closes an open file.

Syntax

Close File [**#**] *filenum*

filenum is an integer number identifying which file to close.

Description

The **Close File** statement closes a file which was opened through the **Open File statement**.

The **Open File statement** and **Close File statement** operate on files in general, not on MapInfo Professional tables. MapBasic provides a separate set of statements (e.g., **Open Table statement**) for manipulating MapInfo tables.

Example

```
Open File "cxdata.txt" For INPUT As #1
'
' read from the file... then, when done:
'
Close File #1
```

See Also:

Open File statement

Close Table statement

Purpose

Closes an open table. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Close Table *table* [**Interactive**]

table is the name of a table that is open

Description

The **Close Table** statement closes an open table. To close all tables, use the **Close All statement**.

If a table is displayed in one or more Grapher or Browser windows, those windows disappear automatically when the table is closed. If the **Close Table** statement closes the only table in a Map window, the window closes. If you use the **Close Table** statement to close a linked table that has edits pending, MapInfo Professional keeps the edits pending until a later session.

Saving Edits

If you omit the optional **Interactive** keyword, MapBasic closes the table regardless of whether the table has unsaved edits; any unsaved edits are discarded. If you include the **Interactive** keyword, and if the table has unsaved edits, MapBasic displays a dialog box allowing the user to save or discard the edits or cancel the close operation.

To guarantee that pending edits are discarded, omit the **Interactive** keyword or issue a **Rollback statement** before calling **Close Table**. To guarantee that pending edits are saved, issue a **Commit Table statement** before the **Close Table** statement. To determine whether a table has unsaved edits, call the **TableInfo( ) function**(*table, TAB_INFO_EDITED*) function.

Saving Themes and Cosmetic Objects

When you close the last table in a Map window, the window closes. However, the user may want to save thematic layers or cosmetic objects before closing the window. To prompt the user to save themes or cosmetic objects, include the **Interactive** keyword.

If you omit the **Interactive** keyword, the **Close Table** statement will not prompt the user to save themes or cosmetic objects. If you include the **Interactive** keyword, dialog boxes will prompt the user to save themes and/or cosmetic objects, if such prompts are appropriate. (The user is not prompted if the window has no themes or cosmetic objects.)

Examples

```
Open Table "world"
' ... when done using the WORLD table,
' close it by saying:
Close Table world
```

To deselect the selected rows, close the Selection table.

```
Close Table Selection
```

See Also:

Close All statement, **Commit Table statement**, **Open Table statement**, **Rollback statement**, **TableInfo( ) function**

Close Window statement

Purpose

Closes or hides a window. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Close Window *window_spec* [**Interactive**]

window_spec is a window name (e.g., Ruler), a window code (e.g., WIN_RULER), or an integer window identifier.

Description

The **Close Window** statement closes or hides a MapInfo Professional window.

To close a document window (Map, Browse, Graph, or Layout), specify an integer window identifier as the *window_spec* parameter. You can obtain integer window identifiers through the **FrontWindow( ) function** and the **WindowID( ) function**.

To close a special MapInfo Professional window, specify one of the window names from the table below as the *window_spec* parameter. You can identify a special window by name (e.g., Ruler) or by code (e.g., WIN_RULER).

To close an adornment window, specify the window ID of the adornment as determined by the **MapperInfo( ) function**.

The following table lists the available *window_spec* values:

window_spec value	Window description
MapBasic	The MapBasic window. You can also refer to this window by its define code: WIN_MAPBASIC.
Help	The Help window. Its define code: WIN_HELP.
Statistics	The Statistics window. Its define code: WIN_STATISTICS.
Legend	The Theme Legend window. Its define code: WIN_LEGEND.
Info	The Info tool window. Its define code: WIN_INFO.

window_spec value	Window description
Ruler	The Ruler tool window. Its define code: WIN_RULER.
Message	The Message window (which appears when you issue a **Print statement**). Its define code: WIN_MESSAGE.

Saving Themes and Cosmetic Objects

The user may want to save thematic layers or cosmetic objects before closing the window. To prompt the user to save themes or cosmetic objects, include the **Interactive** keyword.

If you omit the **Interactive** keyword, the **Close Window** statement will not prompt the user to save themes or cosmetic objects. If you include the **Interactive** keyword, dialog boxes will prompt the user to save themes and/or cosmetic objects, if such prompts are appropriate. (The user will not be prompted if the window has no themes or cosmetic objects.)

Example

```
Close Window Legend
```

See Also:

Open Window statement, **Print statement**, **Set Window statement**

ColumnInfo( ) function

Purpose

Returns information about a column in an open table. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

```
ColumnInfo( { tablename | tablenum } ,
   { columnname | "COLn"} , attribute )
```

tablename is a string representing the name of an open table.

tablenum is an integer representing the number of an open table.

columnname is the name of a column in that table.

n is the number of a column in the table.

attribute is a code indicating which aspect of the column to read.

Return Value

Depends on the *attribute* parameter specified.

Description

The **ColumnInfo( )** function returns information about one column in an open table.

The function's first parameter specifies either the name or the number of an open table. The second parameter specifies which column to query. The *attribute* parameter dictates which of the column's attributes the function should return. The *attribute* parameter can be any value from this table.

attribute setting	ID	ColumnInfo( ) returns:
COL_INFO_NAME	1	String identifying the column name.
COL_INFO_NUM	2	SmallInt indicating the number of the column.
COL_INFO_TYPE	3	SmallInt indicating the column type (see table below).
COL_INFO_WIDTH	4	SmallInt indicating the column width; applies to Character or Decimal columns only.
COL_INFO_DECPLACES	5	SmallInt indicating the number of decimal places in a Decimal column.
COL_INFO_INDEXED	6	Logical value indicating if column is indexed.
COL_INFO_EDITABLE	7	Logical value indicating if column is editable.

If the **ColumnInfo( )** function call specifies COL_INFO_TYPE as its *attribute* parameter, MapBasic returns one of the values from the table below:

ColumnInfo( ) returns:	ID	Type of column indicated:
COL_TYPE_CHAR	1	Character.
COL_TYPE_DECIMAL	2	Fixed-point decimal.
COL_TYPE_INTEGER	3	Integer (4-byte).
COL_TYPE_SMALLINT	4	Small integer (2-byte).
COL_TYPE_DATE	5	Date.
COL_TYPE_LOGICAL	6	Logical (TRUE or FALSE).
COL_TYPE_GRAPHIC	7	special column type Obj; this represents the graphical objects attached to the table.
COL_TYPE_FLOAT	8	Floating-point decimal.
COL_TYPE_TIME	37	Time.
COL_TYPE_DATETIME	38	DateTime.

The codes listed in both of the above tables are defined in the standard MapBasic definitions file, MAPBASIC.DEF. Your program must Include “MAPBASIC.DEF” if you intend to reference these codes.

Error Conditions

ERR_TABLE_NOT_FOUND (405) error generated if the specified table is not available.

ERR_FCN_ARG_RANGE (644) error generated if an argument is outside of the valid range.

Example

```
Include "MAPBASIC.DEF"
Dim s_col_name As String, i_col_type As SmallInt
Open Table "world"
s_col_name = ColumnInfo("world","col1",COL_INFO_NAME)
i_col_type = ColumnInfo("world","col1",COL_INFO_TYPE)
```

See Also:

NumCols( ) function, **TableInfo( ) function**

Combine( ) function

Purpose

Returns a region or polyline representing the union of two objects. The objects cannot be Text objects. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Combine(*object1*, *object2*)

object1, *object2* are two object expressions; both objects can be closed (e.g., a region and a circle), or both objects can be linear (e.g., a line and a polyline)

Return Value

An object that is the union of *object1* and *object2*.

Description

The **Combine( )** function returns an object representing the geographical union of two object expressions. The union of two objects represents the entire area that is covered by either object.

The **Combine( )** function has been updated to allow heterogeneous combines, and to allow Points, MultiPoints, and Collections as input objects. Previously, both objects had to be either linear objects (Lines, Polylines, or Arcs) and produce Polylines as output; or both input objects had to be closed (Regions, Rectangles, Rounded Rectangles, or Ellipses) and produce Regions as output. Heterogeneous combines are not allowed, as are combines containing Point, MultiPoint and Collection objects. Text objects are still not allowed as input to **Combine( )**.

MultiPoint and Collection objects, introduced in MapInfo Professional 6.5, extend the Combine operation. The following table details the possible combine options available and the output results:

Input Object Type	Input Object Type	OutputObject Type
Point or MultiPoint	Point or MultiPoint	MultiPoint
Linear (Line, Polyline, Arc)	Linear	Polyline
Closed (Region, Rectangle, Rounded Rectangle, Ellipse)	Closed	Region
Point, MultiPoint, Linear, Closed, Collection	Point, MultiPoint, Linear, Closed, Collection	Collection

The results returned by **Combine( )** are similar to the results obtained by choosing MapInfo Professional's **Objects > Combine** menu item, except that the **Combine** menu item modifies the original objects; the **Combine( )** function does not alter the *object1* or *object2* expressions. Also, the **Combine( )** function does not perform data aggregation.

The object returned by the **Combine( )** function retains the styles (e.g., color) of the *object1* parameter when possible. Collection objects produced as output will get those portions of style that are possible from *object1*, and the remaining portions of style from *objects2*. For example, if *object1* is a Region and *object2* is a Polyline, then the output collection will use the brush and boarder pen of *object1* for the Region style contained in the collection, and the pen from *object2* for the Polyline style in the collection.

See Also:

Objects Combine statement

CommandInfo( ) function

Purpose

Returns information about recent events. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

CommandInfo(*attribute*)

attribute is an integer code indicating what type of information to return.

Return Value

Logical, float, integer, or string, depending on circumstances.

Description

The **CommandInfo( )** function returns information about recent events that affect MapInfo Professional—for example, whether the "Selection" table has changed, where the user clicked with the mouse, or whether it was a simple **click** or a **SHIFT+click.**

After Displaying a Dialog Box

When you call **CommandInfo( )** after displaying a custom dialog box, the *attribute* parameter can be one of these codes:

***attribute* code**	**ID**	**CommandInfo(*attribute*) returns:**
CMD_INFO_DLG_OK	1	Logical value: TRUE if the user dismissed a custom dialog box by clicking **OK**; FALSE if user canceled by clicking **Cancel**, pressing **Esc**. (This call is only valid following a **Dialog statement**.)
CMD_INFO_STATUS	1	Logical value: TRUE if the user allowed a progress-bar operation to complete, or FALSE if the user pressed the **Cancel** button to halt.

Within a Custom Menu or Dialog Handler

When you call **CommandInfo( )** from within the handler procedure for a custom menu command or a custom dialog box, the *attribute* parameter can be one of these codes:

attribute code	**ID**	**CommandInfo(attribute) returns:**
CMD_INFO_MENUITEM	8	Integer value, representing the ID of the menu item the user chose. This call is only valid within the handler procedure of a custom menu item.
CMD_INFO_DLG_DBL	1	Logical value: TRUE if the user double-clicked on a ListBox or MultiListBox control within a custom dialog box. This call is only valid within the handler procedure of a custom dialog box.

Within a Standard Handler Procedure

When you call **CommandInfo( )** from within a standard system handler procedure (such as SelChangedHandler), the attribute parameter can be any of the codes from the following table. For details, see the separate discussions of SelChangedHandler, RemoteMsgHandler procedure, WinChangedHandler and WinClosedHandler. From within SelChangedHandler:

***attribute* code**	**ID**	**CommandInfo(*attribute*) returns:**
CMD_INFO_SELTYPE	1	1 if one row was added to the selection; 2 if one row was removed from the selection; 3 if multiple rows were added to the selection; 4 if multiple rows were de-selected.
CMD_INFO_ROWID	2	Integer value: The number of the row that was selected or de-selected (only applies if a single row was selected or de-selected).
CMD_INFO_INTERRUPT	3	Logical value: TRUE if the user interrupted a selection by pressing Esc, FALSE otherwise.

From within the **RemoteMsgHandler procedure**, the **RemoteQueryHandler( ) function**, or the **RemoteMapGenHandler procedure**:

***attribute* code**	**ID**	**CommandInfo(*attribute*) returns:**
CMD_INFO_MSG	1000	String value, representing the execute string or the item name sent to MapInfo Professional by a client program. For details, see **RemoteMsgHandler procedure**, **RemoteQueryHandler( ) function**, or **RemoteMapGenHandler procedure**.

From within **WinChangedHandler procedure** or **WinClosedHandler procedure**:

***attribute* code**	**ID**	**CommandInfo(*attribute*) returns:**
CMD_INFO_WIN	1	Integer value, representing the ID of the window that changed or the window that closed. For details, see **WinChangedHandler procedure** or **WinClosedHandler procedure**.

From within **ForegroundTaskSwitchHandler procedure**:

attribute code	ID	CommandInfo(*attribute*) returns:
CMD_INFO_TASK_SWITCH		Integer value, indicating whether MapInfo Professional just became the active application or just stopped being the active application. The return value matches one of these codes: SWITCHING_INTO_MI Pro (If MapInfo Professional received the focus) SWITCHING_OUT_OF_MapInfo Professional (If MapInfo Professional lost the focus).

After a Find Operation

Following a **Find statement**, the *attribute* parameter can be one of these codes:

attribute code	ID	CommandInfo(*attribute*) returns:
CMD_INFO_FIND_RC	3	Integer value, indicating whether the **Find statement** found a match.
CMD_INFO_FIND_ROWID	4	Integer value, indicating the Row ID number of the row that was found.
CMD_INFO_X or CMD_INFO_Y	1, 2	Floating-point number, indicating x- or y-coordinates of the location that was found.

Within a Custom ToolButton's Handler Procedure

Within a custom **ToolHandler procedure**, you can specify any of these codes:

attribute code	ID	CommandInfo(*attribute*) returns:
CMD_INFO_X	1	x coordinate of the spot where the user clicked: • If the user clicked on a Map, the return value represents a map coordinate (e.g., longitude), in the current coordinate system unit. • If the user clicked on a Browser, the value represents the number of a column in the Browser (e.g., one for the left most column, or zero for the select-box column).* • If the user clicked in a Layout, the value represents the distance from the left edge of the Layout (e.g., zero represents the left edge), in MapBasic's current paper units.

attribute code	ID	CommandInfo(*attribute*) returns:
CMD_INFO_Y	2	y-coordinate of the spot where the user clicked: • If the user clicked on a map, the value represents a map coordinate (e.g., Latitude). • If the user clicked on a Browser, the value represents a row number; a value of one represents the top row, and a value of zero represents the row of column headers at the top of the window.* • If the user clicked on a Layout, the value represents the distance from the top edge of the Layout.
CMD_INFO_X2	5	x-coordinate of the spot where the user released the mouse button. This only applies if the toolbutton was defined with a draw mode that allows dragging, e.g., DM_CUSTOM_LINE.
CMD_INFO_Y2	6	y-coordinate of the spot where the user released the mouse button.
CMD_INFO_SHIFT	3	Logical value: TRUE if the user held down the Shift key while clicking.
CMD_INFO_CTRL	4	Logical value: TRUE if the user held down the Ctrl key while clicking.
CMD_INFO_TOOLBTN	7	Integer value, representing the ID of the button the user clicked.
CMD_INFO_CUSTOM_OBJ	1	Object value: a polyline or polygon drawn by the user. Applies to drawing modes DM_CUSTOM_POLYLINE or DM_CUSTOM_POLYGON.

* The CommandInfo( ) function ignores any clicks made in the top-left corner of a Browser window—above the select column and to the left of the column headers. It also ignores clicks made beyond the last column or row.

Hotlink Support

MapBasic applications launched via the Hotlink Tool can use the **CommandInfo( )** function to obtain information about the object that was activated. The following is a table of the attributes that can be queried:

attribute code	ID	CommandInfo(*attribute*) returns:
CMD_INFO_HL_WINDOW_ID	17	ID of map or browser window.
CMD_INFO_HL_TABLE_NAME	18	Name of table associated with the map layer or browser.

attribute code	ID	CommandInfo(*attribute*) returns:
CMD_INFO_HL_ROWID	19	ID of the table row corresponding to the map object or browser row.
CMD_INFO_HL_LAYER_ID	20	Layer ID, if the program was launched from a map window.
CMD_INFO_HL_FILE_NAME	21	Name of file launched.

See Also:

FrontWindow( ) function, **SelectionInfo( ) function**, **Set Command Info statement**, **WindowInfo( ) function**

Commit Table statement

Purpose

Saves recent edits to disk, or saves a copy of a table. In the past, you were unable to save queries that contained indeterminate types, such as often occurred in ObjectInfo queries. We have added an Interactive parameter to allow you to specify indeterminate types in such a query. If you do not use the interactive parameter, the system uses a default type instead. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Commit Table  table
   [ As   filespec
      [ Type  { NATIVE  |
          DBF  [ Charset   char_set  ]  |
          Access Database database_filespec
               Version version
               Table tablename
               [ Password pwd  ]  [ Charset char_set  ] |
          QUERY |
          ODBC Connection ConnectionNumber  Table tablename
          [ Type SQLServerSpatial {Geometry | Geography} ]
          [ ConvertDateTime {ON | OFF | INTERACTIVE}]   }]
      [ CoordSys... ]
      [ Version  version ]  ]
      [ Interactive ]
   [ { Interactive | Automatic  commit_keyword } ]
   [ ConvertObjects {ON | OFF | INTERACTIVE}]
```

tableName is the name of the table as you want it to appear in database. The name can include a schema name, which specifies the schema that the table belongs to. If no schema name is provided, the table belongs to the default schema. The user is responsible for providing an eligible schema name and must know if the login user has the proper permissions on the given schema. This extension is for SQL Server 2005 only.

filespec is a file specification (optionally including directory path). This is where the MapInfo .TAB file is saved.

ConvertDateTime If the source table contains Time or Date type columns, these columns will be converted to DATETIME or TIMESTAMP depending on whether the server supports the data types. However, you can control this behavior using the clause *ConvertDateTime*. If the source table does not contain a Time or Date type, this clause is a non-operational. If *ConvertDateTime* is set to ON (which is the default setting), Time or Date type columns will be converted to DATETIME or TIMESTAMP. If *ConvertDateTime* is set to OFF, the conversion is not done and the operation will be cancelled if necessary. If *ConvertDateTime* is set to INTERACTIVE a dialog box will pop up to prompt the user and the operation will depend on the user's choice. If the user chooses to convert, then the operation will convert and continue; if the user chooses to cancel, the operation will be cancelled.

The Time type requires conversion for all supported servers (Oracle, PostGIS, SQL Server Spatial, MS SQL Server and Access) and the Date type requires conversion for MS SQL Server and Access database servers.

For MS SQL Server and Access database servers, this restriction could be an backward compatibility issue. In previous releases, we did the conversion without explaining it. In this release, we suggest you use the DateTime data type instead of Date data type. If you still use the Date data type, the conversion operation will fail.

version is an expression that specifies the version of the Microsoft Jet database format to be used by the new database. Acceptable values are 4.0 (for Access 2000) or 3.0 (for Access '95/'97). If omitted, the default version is 12.0. If the database in which the table is being created already exists, the specified database version is ignored.

ConvertObjects ON automatically converts any unsupported objects encountered in supported objects.

ConvertObjects OFF This does not convert any unsupported objects. If they are encountered, an error message is displayed saying the table can not be saved. (Before implementation of this feature this was the only behavior.)

ConvertObjects Interactive If any unsupported objects are encountered in a table, ask the user what she wants to do.

char_set is the name of a character set; see **CharSet clause**.

database_filespec is a string that identifies the name and path of a valid Access database. If the specified database does not exist, MapInfo Professional creates a new Access (.MDB or .ACCDB) file.

pwd is the database-level password for the database, to be specified when database security is turned on.

ODBC indicates a copy of the Table will be saved on the DBMS specified by *ConnectionNumber*.

ConnectionNumber is an integer value that identifies the specific connection to a database.

SQL Server Spatial, **SQL Server 2008** supports spatial data with GEOGRAPHY and GEOMETRY data types.

CoordSys is a coordinate system clause; see **CoordSys clause**.

version is 100 (to create a table that can be read by versions of MapInfo Professional) or 300 (MapInfo Professional 3.0 format) for non-Access tables. For Access tables, version is 410.

commit_keyword is one of the following keywords: **NoCollision**, **ApplyUpdates**, **DiscardUpdates**

ConvertDateTime Examples

Example 1

```
Commit Table DATETIME90 As "D:\MapInfo\Data\Remote\DATETIME90CPY.TAB"
Type ODBC Connection 1 Table """EAZYLOADER"".""DATETIME90CPY"""
ConvertDateTime Interactive
```

Example 2

```
Server 1 Create Table """EAZYLOADER"".""CITY_125AA""" (Field1
Char(10),Field2 Char(10),Field3 Char(10),MI_STYLE Char(254)) KeyColumn
SW_MEMBER ObjectColumn SW_GEOMETRY
or
Server 1 Create Table "EAZYLOADER.CITY_125AA" (Field1 Char(10),Field2
Char(10),Field3 Char(10),MI_STYLE Char(254)) KeyColumn SW_MEMBER
ObjectColumn SW_GEOMETRY


Commit Table City_125aa As
"C:\Projects\Data\TestScripts\English\remote\City_125aacpy.tab" Type ODBC
Connection 1 Table """EAZYLOADER"".""CITY_125AACPY"""
or
Commit Table City_125aa As
"C:\Projects\Data\TestScripts\English\remote\City_125aacpy.tab" Type ODBC
Connection 1 Table "EAZYLOADER.CITY_125AACPY"
```

Description

If no **As** clause is specified, the **Commit Table** statement saves any pending edits to the table. This is analogous to the user choosing **File > Save**.

A **Commit Table** statement that includes an **As** clause has the same effect as a user choosing **File > Save Copy As**. The **As** clause can be used to save the table with a different name, directory, file type, or projection.

To save the table under a new name, specify the new name in the *filespec* string. To save the table in a new directory path, specify the directory path at the start of the *filespec* string.

To save the table using a new file type, include a **Type** clause within the **As** clause.

The **CharSet** clause specifies a character set. The *char_set* parameter should be a string constant, such as “WindowsLatin1”. If no **CharSet** clause is specified, MapBasic uses the default character set for the hardware platform that is in use at runtime. See **CharSet clause** for more information.

To save the table using a different coordinate system or projection, include a **CoordSys** clause within the **As** clause. Note that only a mappable table may have a coordinate system or a projection.

To save a Query use the QUERY type for the table. Only queries made from the user interface and queries created from **Run Command statement**s in MapBasic can be saved. The **Commit Table** statement creates a .TAB file and a .QRY file.

The **Version** clause controls the table's format. If you specify Version 100, MapInfo Professional stores the table in a format readable by versions of MapInfo Professional. If you specify Version 300, MapInfo Professional stores the table in MapInfo Professional 3.0 format. Note that region and polyline objects having more than 8,000 nodes and multiple-segment polyline objects require version 300. If you omit the **Version** clause, the table is saved in the version 300 format.

If a MapBasic application issues a **Commit Table…As** statement affecting a table which has memo fields, the memo fields will not be retained in the new table. No warning will be displayed. If the table is saved to a new table through MapInfo Professional's user interface (by choosing **File > Save Copy As**), MapInfo Professional warns the user about the loss of the memo fields. However, when the table is saved to a new table name through a MapBasic program, no warning appears.

Saving Linked Tables

Saving a linked table can generate a conflict, when another user may have edits the same data in the same table MapInfo Professional will detect if there were any conflicts and allows the user to resolve them. The following clauses let you control what happens when there is a conflict. (These clauses have no effect on saving a conventional MapInfo table.)

Interactive

In the event of a conflict, MapInfo Professional displays the Conflict Resolution dialog box. After a successful **Commit Table Interactive** statement, MapInfo Professional displays a dialog box allowing the user to refresh.

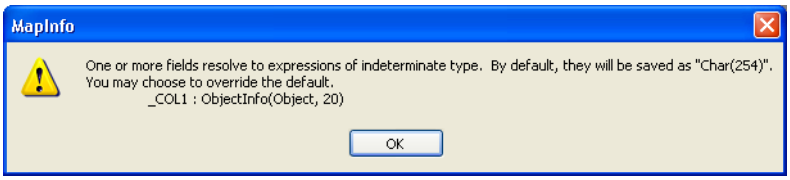
Interactive when invoked for Commit Table As, handles the case when a user is saving a query with one or more columns which are of indeterminate type. Using the Interactive parameter presents the user with a message indicating which column(s) contain the indeterminate type and allows the user to select new types and/or widths for these columns. If the Interactive parameter is not used, the system assigns a Char(254) type to the indeterminate type column(s) by default.

Example

Issue the following query in the SQL Select dialog box and click **OK** or type this query in the MapBasic window:

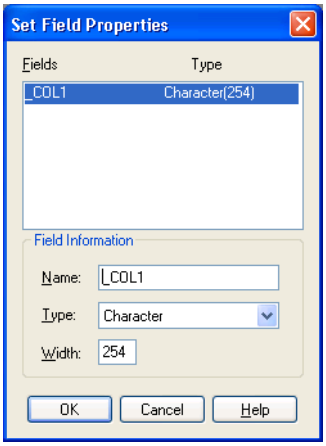
```
Select Highway, objectinfo(obj, 20) from US_HIWAY into Selection
```

When you select **File > Save Copy As**, select the current query, and click the Save As button, the following error message displays:



Typically this dialog box contains a list of all columns that contain indeterminate types. In this query, there is only one.

Click **OK** to display the Set Field Properties dialog box.



Use this dialog box to select the type information for this column. If there is more than one indeterminate type, you can set each of these types one at a time. If there are columns whose type is already defined, you will not be able to edit that information.

Click **OK** to save your query.

Automatic NoCollision

In the event of a conflict, MapInfo Professional does not perform the save. (This is the default behavior if the statement does not include an Interactive clause or an Automatic clause.)

Automatic ApplyUpdates

In the event of a conflict, MapInfo Professional saves the local updates. (This is analogous to ignoring conflicts entirely.)

Automatic DiscardUpdates

In the event of a conflict, MapInfo Professional saves the local updates already in the RDBMS (discards your local updates). You can copy a linked table by using the **As** clause; however, the new copy is not a linked table and no changes are updated to the server.

ODBC Connection

The length of *tablename* varies with databases. We recommend 14 or fewer characters for a table name in order to work correctly for all databases. The statement limits the length of the tablename to a maximum of 31 characters.

If the **As** clause is used and **ODBC** is the Type, a copy of the table will be saved on the database specified by *ConnectionNumber* and named as *tablename*. If the source table is mappable, three more columns, Key column, Object column, and Style column, may be added to the destination database table, *tablename*, whether or not the source table has those columns. If the source table is not mappable, one more column, Key column, may be added to the database table, *tablename*, even if the source table does not have a Key column. The Key column will be used to create a unique index.

A spatial index will be created on the Object column if one is present. The supported databases include Oracle, SQL Server, IIS (SQL Server Spatial, Universal Server), and Microsoft Access. However, to save a table with a spatial geometry/object, (including saving a point-only table) SpatialWare is required for SQL Server, in addition to the spatial option for Oracle. The XY schema is not supported in this statement.

Example

The following example opens the table STATES, then uses the **Commit Table** statement to make a copy of the states table under a new name (ALBERS). The optional **CoordSys clause** causes the ALBERS table to be saved using the Albers equal-area projection.

```
Open Table "STATES"
Commit Table STATES
   As "ALBERS"
   CoordSys Earth
   Projection 9,7, "m", -96.0, 23.0, 20.0, 60.0, 0.0, 0.0
```

The following example illustrates an ODBC connection:

```
dim hodbc as integer
hodbc = server_connect("ODBC", "dlg=1")
Open table "C:\MapInfo\USA"
Commit Table USA
as "c:\temp\as\USA"
   Type ODBC Connection hodbc Table "USA"
```

See Also:

Rollback statement

ConnectObjects( ) function

Purpose

Returns an object representing the shortest or longest distance between two objects. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

ConnectObjects(*object1*, *object2*, *min*)

object1 and *object2* are object expressions.

min is a logical expression where TRUE calculates the minimum distance between the objects, and FALSE calculates the maximum distance between objects.

Return Value

This statement returns a single section, two-point Polyline object representing either the closest distance (*min* == *TRUE*) or farthest distance (*min* == *FALSE*) between *object1* and *object2*.

Description

One point of the resulting Polyline object is on *object1* and the other point is on *object2*. Note that the distance between the two input objects can be calculated using the **ObjectLen( ) function**. If there are multiple instances where the minimum or maximum distance exists (e.g., the two points returned are not uniquely the shortest distance and there are other points representing “ties”) then these functions return one of the instances. There is no way to determine if the object returned is uniquely the shortest distance.

ConnectObjects( ) returns a Polyline object connecting *object1* and *object2* in the shortest (*min* == *TRUE*) or longest (*min* == *FALSE*) way using a spherical calculation method. If the calculation cannot be done using a spherical distance method (e.g., if the MapBasic coordinate system is NonEarth), then a Cartesian method will be used.

Continue statement

Purpose

Resumes the execution of a MapBasic program (following a **Stop statement**). You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Continue

Restrictions

The **Continue** statement may only be issued from the MapBasic window; it may not be included as part of a compiled program.

Description

The **Continue** statement resumes the execution of a MapBasic application which was suspended because of a **Stop statement**.

You can include **Stop statement**s in a program for debugging purposes. When a MapBasic program encounters a **Stop statement**, the program is suspended, and the **File** menu automatically changes to include a **Continue Program** option instead of a **Run** option. You can resume the suspended application by choosing **File > Continue Program**. Typing the **Continue** statement into the MapBasic window has the same effect as choosing **Continue Program**.

Control Button / OKButton / CancelButton clause

Purpose

Part of a **Dialog statement**; adds a push-button control to a dialog box.

Syntax

```
Control { Button | OKButton | CancelButton }
   [ Position x, y ] [ Width w ] [ Height h ]
   [ ID control_ID ]
   [ Calling handler ]
   [ Title title_string ]
   [ Disable ] [ Hide ]
```

x, *y* specifies the button's position in dialog box units.

w specifies the width of the button in dialog box units; default width is 40.

h specifies the height of the button in dialog box units; default height is 18.

control_ID is an integer; cannot be the same as the ID of another control in the dialog box.

handler is the name of a procedure to call if the user clicks on the button.

title_string is a text string to appear on the button.

Description

If a **Dialog statement** includes a **Control Button** clause, the dialog box includes a push-button control. If the **OKButton** keyword appears in place of the **Button** keyword, the control is a special type of button; the user chooses an **OKButton** control to “choose OK” and dismiss the dialog box. Similarly, the user chooses a **CancelButton** control to “choose Cancel” and dismiss the dialog box. Each dialog box should have no more than one **OKButton** control, and have no more than one **CancelButton** control. **Disable** makes the control disabled (grayed out) initially. **Hide** makes the control hidden initially.

Use the **Alter Control statement** to change a control's status (e.g., whether the control is enabled or hidden).

Example

```
Control Button
   Title "&Reset"
   Calling reset_sub
   Position 10, 190
```

See Also:

Alter Control statement, **Dialog statement**

Control CheckBox clause

Purpose

Part of a **Dialog statement**; adds a check box control to a dialog box

Syntax

```
Control CheckBox
   [ Position x, y ] [ Width w ]
   [ ID control_ID ]
   [ Calling handler ]
   [ Title title_string ]
   [ Value log_value ]
   [ Into log_variable ]
   [ Disable ] [ Hide ]
```

x, *y* specifies the control's position in dialog box units.

w specifies the width of the control in dialog box units.

control_ID is an integer; cannot be the same as the ID of another control in the dialog box.

handler is the name of a procedure to call if the user clicks on the control.

title_string is a text string to appear in the label to the right of the check-box.

log_value is a logical value: FALSE sets the control to appear un-checked initially.

log_variable is the name of a logical variable.

Description

If a **Dialog statement** includes a **Control CheckBox** clause, the dialog box includes a check-box control.

The **Value** clause controls the initial appearance. If the **Value** clause is omitted, or if it specifies a value of TRUE, the check-box is checked initially. If the **Value** clause specifies a FALSE value, check-box is clear initially. **Disable** makes the control disabled (grayed out) initially. **Hide** makes the control hidden initially.

Example

```
Control CheckBox
   Title "Include &Legend"
   Into showlegend
   ID 6
   Position 115, 155
```

See Also:

Alter Control statement, **Dialog statement**, **ReadControlValue( ) function**

Control DocumentWindow clause

Purpose

Part of a **Dialog statement**; adds a document window control to a dialog box which can be re-parented for integrated mapping.

Syntax

```
Control DocumentWindow
   [ Position x, y ]
   [ Width w ] [ Height h ]
   [ ID control_ID ]
   [ Disable ] [ Hide ]
```

x, y specifies the control's position in dialog box units.

w specifies the width of the control in dialog units; default width is 100.

h specifies the height of the control in dialog units; default height is 100.

control_ID is an integer; cannot be the same as the ID of another control in the dialog box.

Disable grays out the control initially.

Hide initially hides the control.

Description

If a **Dialog statement** includes a **Control DocumentWindow** clause, the dialog box includes a document window control that can be re-parented using the **Set Next Document statement**.

Example

The following example draws a legend in a dialog box:

```
Control DocumentWindow
      ID ID_LEGENDWINDOW
   Position 160, 20
   Width 120 Height 150
```

The dialog box handler will need to re-parent the window as in the following example:

```
Sub DialogHandler
      OnError Goto HandleError
      Dim iHwnd As Integer
      Alter Control ID_LEGENDWINDOW Enable Show
      ' draw the legend
      iHwnd = ReadControlValue(ID_LEGENDWINDOW)
      Set Next Document Parent iHwnd Style WIN_STYLE_CHILD
      Create Legend
   Exit Sub
   HandleError:
```

```
     Note "DialogHandler: " + Error$( )
End Sub
```

See Also:

Dialog statement

Control EditText clause

Purpose

Part of a **Dialog statement**; adds an EditText control box (input text) to a dialog box.

Syntax

```
Control EditText
   [ Position x, y ] [ Width w ] [ Height h ]
   [ ID control_ID ]
   [ Value initial_value ]
   [ Into variable ]
   [ Disable ] [ Hide ] [ Password ]
```

x, *y* specifies the control's position in dialog box units.

w specifies the width of the control in dialog box units.

h specifies the height of the control in dialog box units; if the height is greater than 20, the control becomes a multiple-line control, and text wraps down onto successive lines.

control_ID is an integer; cannot be the same as the ID of another control in the dialog box.

initial_value is a string or a numeric expression that initially appears in the dialog box.

variable is the name of a string variable or a numeric variable; MapInfo Professional stores the final value of the field in the variable if the user clicks **OK**.

The **Disable** keyword makes the control disabled (grayed out) initially.

The **Hide** keyword makes the control hidden initially.

The **Password** keyword creates a password field, which displays asterisks as the user types.

Description

If the user types more text than can fit in the box at one time, MapInfo Professional automatically scrolls the text to make room. An EditText control can hold up to 32,767 characters.

If the height is large enough to fit two or more lines of text (for example, if the height is larger than 20), MapInfo Professional automatically wraps text down to successive lines as the user types. If the user enters a line-feed into the EditText box (for example, on Windows, if the user presses **Ctrl-Enter** while in the EditText box), the string associated with the EditText control will contain a `Chr$(10)` value at the location of each line-feed. If the *initial_value* expression contains embedded `Chr$(10)` values, the text appears formatted when the dialog box appears.

To make an EditText control the active control, use an **Alter Control…Active statement**.

Example

```
Control EditText
   Value "Franchise Locations"
   Position 65, 8 Width 90
   ID 1
   Into s_map_title
```

See Also:

Alter Control statement, **Dialog statement**, **ReadControlValue( ) function**

Control GroupBox clause

Purpose

Part of a **Dialog statement**; adds a rectangle with a label to a dialog box.

Syntax

```
Control GroupBox
   [ Position x, y ] [ Width w ] [ Height h ]
   [ ID control_ID ]
   [ Title title_string ]
   [ Hide ]
```

x, *y* specifies the control's position in dialog box units.

w specifies the width of the control in dialog box units.

h specifies the height of the control in dialog box units.

control_ID is an integer; cannot be the same as the ID of another control in the dialog box.

title_string is a text string to appear at the upper-left corner of the box

The **Hide** keyword makes the control hidden initially.

Example

```
Control GroupBox
   Title "Level of Detail"
   Position 5, 30
   Height 40 Width 70
```

See Also:

Alter Control statement, **Dialog statement**

Control ListBox / MultiListBox clause

Purpose

Part of a **Dialog statement**; adds a list to a dialog box

Syntax

```
Control { ListBox | MultiListBox }
   [ Position x, y ] [ Width w ] [ Height h ]
   [ ID control_ID ]
   [ Calling handler ]
   [ Title { str_expr | From Variable str_array_var } ]
   [ Value i_selected ]
   [ Into i_variable ]
   [ Disable ] [ Hide ]
```

x, *y* specifies the control's position in dialog box units.

w specifies the width of the control in dialog box units; default width is 80.

h specifies the height of the control in dialog box units; default height is 70.

control_ID is an integer; cannot be the same as the ID of another control in the dialog box.

handler is the name of a procedure to call if the user clicks or double-clicks on the list.

str_expr is a string expression, containing a semicolon-delimited list of items to appear in the control.

str_array_var is the name of an array of string variables.

i_selected is a SmallInt value indicating which list item should appear selected when the dialog box first appears: a value of one selects the first list item; if the clause is omitted, no items are selected initially.

i_variable is the name of a SmallInt variable which stores the user's final selection.

The **Disable** keyword makes the control disabled (grayed out) initially.

The **Hide** keyword makes the control hidden initially.

Description

If a **Dialog statement** includes a **Control ListBox** clause, the dialog box includes a listbox control. If the list contains more items than can be shown in the control at one time, MapBasic automatically adds a scroll-bar at the right side of the control.

A MultiListBox control is identical to a ListBox control, except that the user can shift-click to select multiple items from a MultiListBox control.

The **Title** clause specifies the contents of the list. If the **Title** clause specifies a string expression containing a semicolon-delimited list of items, each item appears as one item in the list. The following sample **Title** clause demonstrates this syntax:

```
Title "1st Quarter;2nd Quarter;3rd Quarter;4th Quarter"
```

Alternately, if the **Title** clause specifies an array of string variables, each entry in the array appears as one item in the list. The following sample **Title** clause demonstrates this syntax:

```
Title From Variable s_optionlist
```

Processing a MultiListBox control

To read what items the user selected from a MultiListBox control, assign a handler procedure that is called when the user dismisses the dialog box (for example, assign a handler to the OKButton control). Within the handler procedure, set up a loop to call the **ReadControlValue( ) function** repeatedly.

The first call to the **ReadControlValue( ) function** returns the number of the first selected item; the second call to the **ReadControlValue( ) function** returns the number of the second selected item; etc. When the **ReadControlValue( ) function** returns zero, you have exhausted the list of selected items. If the first call to the **ReadControlValue( ) function** returns zero, there are no list items selected.

Processing Double-click events

If you assign a *handler* procedure to a list control, MapBasic calls the procedure every time the user clicks or double-clicks an item in the list. In some cases, you may want to provide special handling for double-click events. For example, when the user double-clicks a list item, you may want to dismiss the dialog box as if the user had clicked on a list item and then clicked **OK**.

To see an example, refer to the sample application NVIEWS.MB in <*Your MapBasic Installation Directory*>\SAMPLES\MAPBASIC\SNIPPETS.

To determine whether the user clicked or double-clicked, call the **CommandInfo( ) function** within the list control's handler procedure, as shown in the following sample handler procedure:

```
Sub lb_handler
   Dim i As SmallInt
   If CommandInfo(CMD_INFO_DLG_DBL) Then
   ' ... then the user double-clicked.
      i = ReadControlValue( TriggerControl( ) )
      Dialog Remove
      ' at this point, the variable i represents
      ' the selected list item...
   End If
End Sub
```

Example

```
Control ListBox
   Title "1st Quarter;2nd Quarter;3rd Quarter;4th Quarter"
   ID 3
   Value 1
   Into i_quarter
   Position 10, 92 Height 40
```

The NVIEWS.MB sample program demonstrates how to create a dialog box which provides special handling for when the user double-clicks. The NVIEWS program displays a dialog box with a ListBox control. To complete the dialog box, the user can click on a list item and then choose **OK**, or the user can double-click an item in the list.

The following **Control ListBox** clause adds a list to the Named Views dialog box. Note that the ListBox control has a handler routine, “listbox_handler.”

```
Control ListBox
   Title desc_list
   ID 1
   Position 10, 20 Width 245 Height  64
   Calling listbox_handler
```

If the user clicks or double-clicks on the ListBox control, MapBasic calls the sub procedure “listbox_handler.” The procedure calls the **CommandInfo( ) function** to determine whether the user clicked or double-clicked. If the user double-clicked, the procedure issues a **Dialog Remove statement** to dismiss the dialog box. If not for the **Dialog Remove statement**, the dialog box would remain on the screen until the user clicked **OK** or **Cancel**.

```
Sub listbox_handler
   Dim i As SmallInt
   ' First, since user clicked on the name of a view,
   ' we can enable the OK button and the Delete button.
   Alter Control 2 Enable
   Alter Control 3 Enable
   If CommandInfo(CMD_INFO_DLG_DBL) = TRUE Then
   ' ...then the user DOUBLE-clicked.
   ' see which list item the user clicked on.
      i = ReadControlValue(1) ' read user's choice.
      Dialog Remove
      Call go_to_view(i) ' act on user's choice.
   End If
End Sub
```

MapBasic calls the handler procedure whether the user clicks or double-clicks. The handler procedure must check to determine whether the event was a single- or double-click.

See Also:

Alter Control statement, **Dialog statement**, **ReadControlValue( ) function**, **CommandInfo( ) function**

Control PenPicker/BrushPicker/SymbolPicker/FontPicker clause

Purpose

Part of a **Dialog statement**; adds a button showing a pen (line), brush (fill), symbol (point), or font (text) style.

Syntax

```
Control { PenPicker | BrushPicker | SymbolPicker | FontPicker }
   [ Position x, y ] [ Width w ] [ Height h ]
   [ ID control_ID ]
   [ Calling handler ]
   [ Value style_expr ]
   [ Into style_var ]
   [ Disable ] [ Hide ]
```

x, *y* specifies the control's position, in dialog box units.

w specifies the control's width, in dialog box units; default width is 20.

h specifies the control's height, in dialog box units; default height is 20.

control_ID is an integer; cannot be the same as the ID of another control in the dialog box.

handler is the name of a handler procedure; if the user clicks on the Picker control, and then clicks **OK** on the style dialog box which appears, MapBasic calls the *handler* procedure.

style_expr is a Pen, Brush, Symbol, or Font expression, specifying what style will appear initially in the control; this expression type must match the type of control (for example, must be a Pen expression if the control is a PenPicker).

style_var is the name of a Pen, Brush, Symbol, or Font variable; this variable type must match the type of control (for example, must be a Pen variable if the control is a PenPicker control).

The **Disable** keyword makes the control disabled (grayed out) initially.

The **Hide** keyword makes the control hidden initially.

Description

A Picker control (PenPicker, BrushPicker, SymbolPicker, or FontPicker) is a button showing a pen, brush, symbol, or font style. If the user clicks on the button, a dialog box appears to allow the user to change the style.

Example

```
Control SymbolPicker
   Position 140,42
   Into sym_storemarker
```

See Also:

Alter Control statement, **Dialog statement**, **ReadControlValue( ) function**

ControlPointInfo() function

Purpose:

Returns raster and geographic control point coordinates for an image table. The geographic coordinates will be in the current MapBasic coordinate system.

Syntax:

```
ControlPointInfo( table_id, attribute, controlpoint_num )
```

table_id is a string representing a table name, a positive integer table number, or 0 (zero). The table must be a raster, grid or WMS table.

attribute is an integer code indicating which aspect of the control point to return.

controlpoint_num is the integer number of which control point to return. Control point numbers start at 1. The maximum control point number can be found by calling

```
RasterTableInfo(table_id, RASTER_TAB_INFO_NUM_CONTROL_POINTS)
```

Return Value

The X or Y raster coordinate is returned as an Integer. The X or Y geographic coordinate is returned as a Float. The return type depends upon the attribute flag, for the control point specified by controlpoint_num.

The attribute parameter can be any value from the table below. Codes in the left column (for example, RASTER_CONTROL_POINT_X) are defined in MAPBASIC.DEF.

attribute code	ID	ControlPointInfo() returns:
RASTER_CONTROL_POINT_X	1	Integer result, representing the X coordinate of the control point number specified by controlpoint_num
RASTER_CONTROL_POINT_Y	2	Integer result, representing the Y coordinate of the control point number specified by controlpoint_num
GEO_CONTROL_POINT_X	3	Float result, representing the X coordinate of the control point number specified by controlpoint_num
GEO_CONTROL_POINT_Y	4	Float result, representing the Y coordinate of the control point number specified by controlpoint_num
TAB_GEO_CONTROL_POINT_X	5	Float result, representing the X coordinate of the control point number specified by controlpoint_num stored in the raster image TAB file
TAB_GEO_CONTROL_POINT_Y	6	Float result, representing the Y coordinate of the control point number specified by controlpoint_num stored in the raster image TAB file

Control PopupMenu clause

Purpose

Part of a **Dialog statement**; adds a popup menu control to the dialog box.

Syntax

```
Control PopupMenu
   [ Position x, y ]
   [ Width w ]
   [ ID control_ID ]
   [ Calling handler ]
   [ Title { str_expr | From Variable str_array_var } ]
   [ Value i_selected ]
   [ Into i_variable ]
   [ Disable ]
```

x, *y* specifies the control's position in dialog box units.

w specifies the control's width, in dialog box units; default width is 80.

control_ID is an integer; cannot be the same as the ID of another control in the dialog box.

handler is the name of a procedure to call when the user chooses an item from the menu.

str_expr is a string expression, containing a semicolon-delimited list of items to appear in the control.

str_array_var is the name of an array of string variables.

i_selected is a SmallInt value indicating which item should appear selected when the dialog box first appears: a value of one selects the first item; if the clause is omitted, the first item appears selected.

i_variable is the name of a SmallInt variable which stores the user's final selection (one, if the first item selected, etc.).

The **Disable** keyword makes the control disabled (grayed out) initially.

Description

If a **Dialog statement** includes a **Control PopupMenu** clause, the dialog box includes a pop-up menu. A pop-up menu is a list of items, one of which is selected at one time. Initially, only the selected item appears on the dialog box.

If the user clicks on the control, the entire menu appears, and the user can choose a different item from the menu.

The **Title** clause specifies the list of items that appear in the menu. If the **Title** clause specifies a string expression containing a semicolon-delimited list of items, each item appears as one item in the menu. The following sample **Title** clause demonstrates this syntax:

```
Title "Town;County;Territory;Region;Entire state"
```

Alternately, the **Title** clause can specify an array of string variables, in which case each entry in the array appears as one item in the popup menu.

The following sample **Title** clause demonstrates this syntax:

```
Title From Variable s_optionlist
```

Example

```
Control PopupMenu
   Title "Town;County;Territory;Region;Entire state"
   Value 2
   ID 5
   Into i_map_scope
   Position 10, 150
```

See Also:

Alter Control statement, **Dialog statement**, **ReadControlValue( ) function**

Control RadioGroup clause

Purpose

Part of a **Dialog statement**; adds a list of radio buttons to the dialog box.

Syntax

```
Control RadioGroup
   [ Position x, y ]
   [ ID control_ID ]
   [ Calling handler ]
   [ Title { str_expr | From Variable str_array_var } ]
   [ Value i_selected ]
   [ Into i_variable ]
   [ Disable ] [ Hide ]
```

x, *y* specifies the control's position in dialog box units.

control_ID is an integer; cannot be the same as the ID of another control in the dialog box.

handler is the name of a procedure to call if the user clicks or double-clicks on any of the radio buttons.

str_expr is a string expression, containing a semicolon-delimited list of items to appear in the control.

str_array_var is the name of an array of string variables.

i_selected is a SmallInt value indicating which item should appear selected when the dialog box first appears: a value of one selects the first item; if the clause is omitted, the first item appears selected.

i_variable is the name of a SmallInt variable which stores the user's final selection (one, if the first item selected, etc.).

The **Disable** keyword makes the control disabled (grayed out) initially.

The **Hide** keyword makes the control hidden initially.

Description

If a **Dialog statement** includes a **Control RadioGroup** clause, the dialog box includes a group of radio buttons. Each radio button is a label to the right of a hollow or filled circle. The currently-selected item is indicated by a filled circle. Only one of the radio buttons may be selected at one time.

The **Title** clause specifies the list of labels that appear in the dialog box. If the **Title** clause specifies a string expression containing a semicolon-delimited list of items, each item appears as one item in the list.

The following sample **Title** clause demonstrates this syntax:

```
Title "&Full Details;&Partial Details"
```

Alternately, the **Title** clause can specify an array of string variables, in which case each entry in the array appears as one item in the list. The following sample **Title** clause demonstrates this syntax:

```
Title From Variable s_optionlist
```

Example

```
Control RadioGroup
   Title "&Full Details;&Partial Details"
   Value 2
   ID 2
   Into i_details
   Calling rg_handler
   Position 15, 42
```

See Also:

Alter Control statement, **Dialog statement**, **ReadControlValue( ) function**

Control StaticText clause

Purpose

Part of a **Dialog statement**; adds a label to a dialog box.

Syntax

```
Control StaticText
   [ Position x, y ]
   [ Width w ] [ Height h ]
   [ ID control_ID ]
   [ Title title_string ]
   [ Hide ]
```

x, *y* specifies the control's position, in dialog box units.

w specifies the control's width, in dialog box units.

h specifies the control's height, in dialog box units.

control_ID is an integer; cannot be the same as the ID of another control in the dialog box.

title_string is a text string to appear in the dialog box as a label.

The **Hide** keyword makes the control hidden initially.

Description

If you want the text string to wrap down onto multiple lines, include the optional **Width** and **Height** clauses. If you omit the **Width** and **Height** clauses, the static text control shows only one line of text.

Example

```
Control StaticText
   Title "Enter map title:"
   Position 5, 10
```

See Also:

Alter Control statement, **Dialog statement**

ConvertToPline( ) function

Purpose

Returns a polyline object that approximates the shape of another object. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

ConvertToPline(*object*)

object is the object to convert; may not be a point object or a text object.

Return Value

A polyline object

Description

The **ConvertToPline( )** function returns a polyline object which approximates the object parameter. Thus, if the object parameter represents a region object, **ConvertToPline( )** returns a polyline that has the same shape and same number of nodes as the region.

The results obtained by calling **ConvertToPline( )** are similar to the results obtained by choosing MapInfo Professional's **Objects > Convert To Polyline** command. However, the function **ConvertToPline( )** does not alter the original object.

See Also:

Objects Enclose statement

ConvertToRegion( ) function

Purpose

Returns a region object that approximates the shape of another object. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

ConvertToRegion(*object*)

object is the object to convert; may not be a point, line, or text object.

Return Value

A region object

Description

Retains most style attributes. Other attributes are determined by the current pens or brushes. A polyline whose first and last nodes are identical will not have the last node duplicated. Otherwise, MapInfo Professional adds a last node whose vertices are the same as the first node.

The **ConvertToRegion( )** function returns a region object which approximates the object parameter. Thus, if the object parameter represents a rectangle, **ConvertToRegion( )** returns a region that looks like a rectangle.

The results obtained by calling **ConvertToRegion( )** are similar to the results obtained by choosing MapInfo Professional's **Objects > Convert To Region** command. However, the **ConvertToRegion( )** function does not alter the original object.

See Also:

Objects Enclose statement

ConvexHull( ) function

Purpose

Returns a region object that represents the convex hull polygon based on the nodes from the input object. The convex hull polygon can be thought of as an operator that places a rubber band around all of the points. It will consist of the minimal set of points such that all other points lie on or inside the polygon. The polygon will be convex—no interior angle can be greater than 180 degrees. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

ConvexHull(*inputobject*)

inputobject is an object expression.

Return Value

Returns a region object.

Description

The **ConvexHull( )** function returns a region representing the convex hull of the set of points comprising the input object. The **ConvexHull( )** function operates on one single object at a time. To create a convex hull around a set of objects, use the Create Object As ConvexHull statement.

Example

The following program selects New York from the States file, then creates a ConvexHull surrounding the selection.

```
Dim Resulting_object as object
select * from States
where State_Name = "New York"
Resulting_object = ConvexHull(selection.obj)
Insert Into States(obj) Values (Resulting_object)
```

See Also:

Create Object statement

CoordSys clause

Purpose

Specifies a coordinate system. You can use this clause in the MapBasic Window in MapInfo Professional (see **CoordSys Earth and NonEarth Projection**, **CoordSys Layout Units**, **CoordSys Table**, and **CoordSys Window**).

CoordSys Earth and NonEarth Projection

Syntax 1 (Earth Projection)

```
CoordSys Earth
   [ Projection type, datum, unitname
      [ , origin_longitude ] [ , origin_latitude ]
      [ , standard_parallel_1 [ , standard_parallel_2 ] ]
      [ , azimuth ] [ , scale_factor ]
      [ , false_easting ] [ , false_northing ]
      [ , range ] ]
   [ Affine Units unitname, A, B, C, D, E, F ]
   [ Bounds ( minx, miny ) ( maxx, maxy ) ]
```

Syntax 2 (NonEarth Projection)

```
CoordSys Nonearth
   [ Affine Units unitname, A, B, C, D, E, F ]
```

```
Units unitname
[ Bounds ( minx, miny ) ( maxx, maxy ) ]
```

type is a positive integer value representing which coordinate system to use.

datum is a positive integer value identifying which datum to reference.

unitname is a string representing a distance unit of measure (for example, “m” for meters); for a list of unit names, see **Set Distance Units statement**.

origin_longitude is a float longitude value, in degrees.

origin_latitude is a float latitude value, in degrees.

standard_parallel_1 and *standard_parallel_2* are float latitude values, in degrees.

azimuth is a float angle measurement, in degrees.

scale_factor is a float scale factor.

range is a float value from 1 to 180, dictating how much of the Earth will be seen.

minx is a float specifying the minimum x value.

miny is a float specifying the minimum y value.

maxx is a float specifying the maximum x value.

maxy is a float specifying the maximum y value.

A performs scaling or stretching along the X axis.

B performs rotation or skewing along the X axis.

C performs shifting along the X axis.

D performs scaling or stretching along the Y axis.

E performs rotation or skewing along the Y axis.

F performs shifting along the Y axis.

Description

The **CoordSys** clause specifies a coordinate system, and, optionally, specifies a map projection to use in conjunction with the coordinate system. Note that **CoordSys** is a clause, not a complete MapBasic statement. Various statements may include the **CoordSys** clause; for example, a **Set Map statement** can include a **CoordSys** clause, in which case the **Set Map statement** will reset the map projection used by the corresponding Map window.

Use **CoordSys Earth** (syntax 1) to explicitly define a coordinate system for an Earth map (a map having coordinates which are specified with respect to a location on the surface of the Earth). The optional **Projection** parameters dictate what map projection, if any, should be used in conjunction with the coordinate system. If the **Projection** clause is omitted, MapBasic uses datum 0. The **Affine** clause describes the affine transformation for producing the derived coordinate system. If the **Projection** clause is omitted, the base coordinate system is Longitude/Latitude. Since the derived coordinates may be in different units than the base coordinates, the **Affine** clause requires you to specify the derived coordinate units.

Use **CoordSys Nonearth** (syntax 2) to explicitly define a non-Earth coordinate system, such as the coordinate system used in a floor plan or other CAD drawing. In the **CoordSys** Non-Earth case, the base coordinate system is an arbitrary Cartesian grid. The **Units** clause specifies the base coordinate units, and the **Affine** clause specifies the derived coordinate units.

When a **CoordSys** clause appears as part of a **Set Map statement** or **Set Digitizer statement**, the **Bounds** subclause is ignored. The **Bounds** subclause is required for non-Earth maps when the **CoordSys** clause appears in any other statement, but only for non-Earth maps.

The **Bounds** clause defines the map's limits; objects may not be created outside of those limits. When specifying an Earth coordinate system, you may omit the **Bounds** clause, in which case MapInfo Professional uses default bounds that encompass the entire Earth.

In a **Create Map statement**, you can increase the precision of the coordinates in the map by specifying narrower Bounds.

Every map projection is defined as an equation; and since the different projection equations have different sets of parameters, different **CoordSys** clauses may have varying numbers of parameters in the optional **Projection** clause. For example, the formula for a Robinson projection uses the *datum*, *unitname*, and *origin_latitude* parameters, while the formula for a Transverse Mercator projection uses the *datum*, *unitname*, *origin_longitude*, *origin_latitude*, *scale_factor*, *false_easting*, and *false_northing* parameters.

For more information on projections and coordinate systems, see the MapInfo Professional documentation.

Each MapBasic application has its own **CoordSys** setting that specifies the coordinate system used by the application. If a MapBasic application issues a **Set CoordSys statement**, other MapBasic applications which are also in use will not be affected.

Examples

The **Set Map statement** controls the settings of an existing Map window. The **Set Map statement** below tells MapInfo Professional to display the Map window using the Robinson projection:

```
Set Map CoordSys Earth Projection 12, 12, "m", 0.
```

The first 12 specifies the Robinson projection; the second 12 specifies the Sphere datum; the “m” specifies that the coordinate system should use meters; and the final zero specifies that the origin of the map should be at zero degrees longitude.

The following statement tells MapInfo Professional to display the Map window without any projection.

```
Set Map CoordSys Earth
```

The following example opens the table World, then uses a **Commit Table statement** to save a copy of World under the name RWorld. The new RWorld table will be saved with the Robinson projection.

```
Open Table "world" As World
 Table world As "RWORLD.TAB"
   CoordSys Earth Projection 12, 12, "m", 0.
```

The following example defines a coordinate system called DCS that is derived from UTM Zone 10 coordinate system using the affine transformation.

```
x1 = 1.57x - 0.21y + 84120.5
y1 = 0.19x + 2.81y - 20318.0
```

In this transformation, (x1, y1) represents the DCS derived coordinates, and (x, y) represents the UTM Zone 10 base coordinates. If the DCS coordinates are measured in feet, the **CoordSys** clause for DCS would be as follows:

```
CoordSys Earth
   Projection 8, 74, "m", -123, 0, 0.9996, 500000, 0
   Affine Units "ft", 1.57, -0.21, 84120.5, 0.19, 2.81, -20318.0
```

CoordSys Layout Units

Syntax

`CoordSys Layout Units` *`paperunitname`*

paperunitname is a string representing a paper unit of measure (for example, “in” for inches); for a list of unit names, see **Set Paper Units statement**.

Description

Use **CoordSys Layout** to define a coordinate system which represents a MapInfo Professional Layout window. A MapBasic program must issue a Set CoordSys Layout statement before querying, creating or otherwise manipulating Layout objects. The *unitname* parameter is the name of a paper unit, such as “in” for inches or “cm” for centimeters.

Examples

The following **Set CoordSys statement** assigns a Layout window's coordinate system, using inches as the unit of measure:

```
Set CoordSys Layout Units "in"
```

CoordSys Table

Syntax

`CoordSys Table` *`tablename`*

tablename is the name of an open table.

Description

Use **CoordSys Table** to refer to the coordinate system in which a table has been saved.

CoordSys Window

Syntax

CoordSys Window *window_id*

window_id is an integer window identifier corresponding to a Map or Layout window.

Description

Use **CoordSys Window** to refer to the coordinate system already in use in a window.

Examples

The following example sets one Map window's projection to match the projection of another Map window. This example assumes that two integer variables (*first_map_id* and *second_map_id*) already contain the window IDs of the two Map windows.

```
Set Map
   Window second_map_winid
   CoordSys Window first_map_winid
```

See Also:

Commit Table statement, **Set CoordSys statement**, **Set Map statement**

CoordSysName$( ) function

Purpose

Returns coordinate system name string from MapBasic Coordinate system clause. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

CoordSysName$ (*string*)

Return Value

String

Example

```
Note CoordSysName$("Coordsys Earth Projection 1, 62")
```

Returns this string in the MapInfo dialog box:

```
   Longitude / Latitude (NAD 27 for Continental US)
```

ⓘ If a coordinate system name does not exist in the MapInfow.prj file, such as when the map is in NonEarth system in Survey Feet, then function will return an empty string.

```
Note CoordSysName$("CoordSys NonEarth Units " + """survey ft""" +
"Bounds (0, 0) (10, 10)")
```

If an invalid CoordSys clause is passed such as this (using invalid units):

```
Note CoordSysName$("CoordSys Earth Projection 3, 74, " + """foo""" +
"-90, 42, 42.7333333333, 44.0666666667, 1968500, 0")
```

Then an Error regarding the Invalid Coordinate System should be returned (Error #727).

```
Invalid Coordinate System: CoordSys Earth Projection <content>
```

CoordSysStringToEPSG( ) function

Purpose

Converts a MapBasic Coordinate System clause into an EPSG integer value for use with any MapBasic function or statement. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

CoordSysStringToEPSG (*epsg_string*)

epsg_string is a MapBasic CoordSys clause. EPSG (European Petroleum Survey Group) value is an integer value; for example, CoordSys Clause of "Earth Projection 1, 104" will return an EPSG code of 4326. For a complete list of EPSG codes used with MapInfo Professional see the MAPINFOW.PRJ file in your MapInfo Professional installation. The EPSG codes are identified by a "\p" followed by a number.

Return Value

Integer. If no EPSG value is found, it returns -1.

Description

The CoordSysStringToEPSG( ) function is used to convert a MapBasic **CoordSys clause** into an integer EPSG value.

Example

The following example displays EPSG code Earth Projection 1, 104 Coordinate System.

```
print CoordSysStringToEPSG("Earth Projection 1, 104"))
```

See Also:

CoordSys clause

CoordSysStringToPRJ$( ) function

Purpose

Converts MapBasic Coordinate System clause into an PRJ string. PRJ string format is used to describe MapInfo Coordinate Systems in mapinfow.prj file. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

CoordSysStringToPRJ$(*prj_string*)

prj_string is a MapBasic CoordSys clause. PRJ string is an alternative definition of Coordinate System used in the mapinfow.prj file; for example, CoordSys Clause of "Earth Projection 1, 104" will return a PRJ string of "1,104".

Return Value

string

Description

The CoordSysStringToPRJ$( ) function is used to convert a MapBasic **CoordSys clause** into an integer EPSG value.

Example

The following example displays PRJ string for Earth Projection 1, 104 Coordinate System.

```
print CoordSysStringToPRJ$("Earth Projection 1, 104")
```

See Also:

CoordSys clause

CoordSysStringToWKT$( ) function

Purpose

Converts a MapBasic Coordinate System clause into a WKT string. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

CoordSysStringToWKT$(*wkt_string*)

wkt_string is a MapBasic CoordSys clause. WKT (Well-Known Text) value is a string value; for example, CoordSysStringToWKT$(“Coordsys Earth Projection 1, 62”) will produce following string:

```
GEOGCS["NAD27 Latitude/Longitude,
Degrees",DATUM["North_American_Datum_1927",SPHEROID["Clarke -
```

```
1866",6378206.4,294.9786982139006],AUTHORITY["EPSG","6267"]],PRIMEM["Gree
nwich",0],UNIT["degree",0.0174532925199433]]
```

Return Value

WKT string. If no WKT string value is found, it returns an empty string.

Description

The CoordSysStringToWKT$() function is used to convert a MapBasic **CoordSys clause** into an WKT string value.

Example

The following example:

```
Print coordsysstringtowkt$("CoordSys Earth Projection 8, 74, " + """m""" +
", -123, 0, 0.9996, 500000, 0 Affine Units " + """ft""" + ", 1.57, -0.21,
84120.5, 0.19, 2.81, -20318.0")
```

produces a WKT string:

```
PROJCS["_MI_0",GEOGCS[,DATUM["North_American_Datum_1983",SPHEROID["Geodet
ic Reference System of
1980",6378137,298.2572221009113],AUTHORITY["EPSG","6269"]],PRIMEM["Greenw
ich",0],UNIT["degree",0.0174532925199433]],PROJECTION["Transverse_Mercato
r"],PARAMETER["latitude_of_origin",0],PARAMETER["central_meridian",-
123],PARAMETER["scale_factor",0.9996],PARAMETER["false_easting",500000],P
ARAMETER["false_northing",0],UNIT["METER",1]]
```

See Also:

CoordSys clause

Cos( ) function

Purpose

Returns the cosine of a number. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

Cos (*num_expr*)

num_expr is a numeric expression representing an angle in radians.

Return Value

Float

Description

The **Cos( )** function returns the cosine of the numeric *num_expr* value, which represents an angle in radians. The result returned from **Cos( )** is between one (1) and negative one (-1).

To convert a degree value to radians, multiply that value by DEG_2_RAD. To convert a radian value into degrees, multiply that value by RAD_2_DEG.

Your program must Include “MAPBASIC.DEF” to reference DEG_2_RAD or RAD_2_DEG.

Example

```
Include "MAPBASIC.DEF"
Dim x, y As Float
x = 60 * DEG_2_RAD
y = Cos(x)

' y will now be equal to 0.5
' since the cosine of 60 degrees is 0.5
```

See Also:

Acos( ) function, **Asin( ) function**, **Atn( ) function**, **Sin( ) function**, **Tan( ) function**

Create Adornment statement

Purpose

Creates and displays Adornments, such as a scale bar, on mapper window. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
create adornment
   From Window map_window_id
   Type adornment_type
   [ Position {
      [ Fixed [ ( x, y ) [ Units paper_units ] ] ] |
      [ win_position [ Offset (x, y) ] [Units paper_units ] ]
   } ]
   [ Layout Fixed Position { Frame | Geographic } ]
   [ Size [ Width win_width ] [ Height win_height ] [ Units paper_units ] ]
   [ Background [ Brush ... ] [ Pen ... ] ]
   [ < SCALEBAR CLAUSE > ]
```

Where *SCALEBAR CLAUSE is:*

```
   [ BarType type ]
   [ Ground Units distance_units ]
   [ Display Units paper_units ]
   [ BarLength paper_length ]
```

```
[ BarHeight paper_height ]
[ BarStyle [ Pen .... ] [ Brush ... ] [ Font ... ] ]
[ Scale [ { On | Off } ] ]
```

adornment_type can be **scalebar**.

Position can be **Fixed** relative to the mapper upper left regardless of the size of the mapper, or relative to some anchor point on the mapper specified by *win_position*.

(*x, y*) in the **Fixed** clause is position measured from the upper left of the mapper window, which is (0, 0). Using this version of adornment placement, the adornment will be at that position in the mapper as the mapper resizes. For example, a position of (3, 3) inches would be toward the bottom right of a small sized mapper but in the middle of a large sized mapper. As the mapper changes size, the adornment will try to remain completely within the displayed mapper.

paper_units defaults to the MapBasic Paper Unit (see **Set Paper Units statement**).

win_position specify one of the following codes; codes are defined in MAPBASIC.DEF.

```
ADORNMENT_INFO_MAP_POS_TL (0)
ADORNMENT_INFO_MAP_POS_TC (1)
ADORNMENT_INFO_MAP_POS_TR (2)
ADORNMENT_INFO_MAP_POS_CL (3)
ADORNMENT_INFO_MAP_POS_CC (4)
ADORNMENT_INFO_MAP_POS_CR (5)
ADORNMENT_INFO_MAP_POS_BL (6)
ADORNMENT_INFO_MAP_POS_BC (7)
ADORNMENT_INFO_MAP_POS_BR (8)
```

Offset is the amount the adornment will be offset from the mapper when using one of the docked *win_positions*.

(x, y) in the **Offset** clause is measured from the anchor position. For example, if the *win_position* is ADORNMENT_INFO_MAP_POS_TL (top left), then the x is to the right and the y is down. If the *win_position* is ADORNMENT_INFO_MAP_POS_BR, then the x position is left and the y position is up. In the center left (ADORNMENT_INFO_MAP_POS_CL) and center right (ADORNMENT_INFO_MAP_POS_CR), the y offset is ignored. In the center position (ADORNMENT_INFO_MAP_POS_CC), the offset is ignored completely (both x and y). In the top center (ADORNMENT_INFO_MAP_POS_TC) and bottom center (ADORNMENT_INFO_MAP_POS_BC) positions, the x offset is ignored. For ADORNMENT_INFO_MAP_POS_ defines, see *win_position.*

Layout Fixed Position determines how an adornment is positioned in a layout when the adornment is using Fixed positioning. If this is set to **Geographic**, then the adornment is placed on the same geographic place on the map frame in the layout as it is in the mapper. If the layout frame changes size, then the adornment will move relative to the frame to match the geographic position. If this is set to **Frame**, then the adornment will remain at a fixed position relative to the frame, as designated in the **Position** clause. If the Position clause positions the adornment at (1.0, 1.0) inches, then the adornment will be placed 1 inch to the left and one inch down from the upper left corner of the frame. Changing the size of the frame will not change the position of the adornment. The default is **Geographic**.

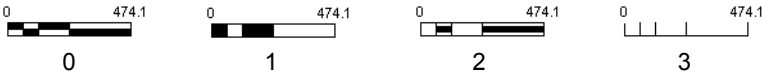
win_width and *win_height* define the size of the adornment. MapInfo Professional ignores these parameters if this is a scale bar adornment, because scale bar adornment size is determined by scale bar specific items, such as *BarLength*.

Brush is a valid **Brush clause**. Only Solid brushes are allowed. While values other than solid are allowed as input without error, the type is always forced to solid. This clause is used only to provide the background color for the adornment.

Pen is a valid **Pen clause**. Due to window clipping (the adornment is a window within the mapper), Pen widths other than 1 may not display correctly. Also, Pen styles other than solid may not display correctly. This clause is designed to turn on (solid) or off (hollow) and set the color of the border of the adornment.

type specify one of the following codes; codes are defined in MAPBASIC.DEF.

```
SCALEBAR_INFO_BARTYPE_CHECKEDBAR  (0)
SCALEBAR_INFO_BARTYPE_SOLIDBAR    (1)
SCALEBAR_INFO_BARTYPE_LINEBAR     (2)
SCALEBAR_INFO_BARTYPE_TICKBAR     (3)
```



***0** Check Bar, **1** Solid Bar, **2** Line Bar, or **3** Tick Bar*

distance_units a unit of measure that the scale bar is to represent:

distance value	Unit Represented
“ch”	chains
“cm”	centimeters
“ft”	feet (also called International Feet; one International Foot equals exactly 30.48 cm)
“in”	inches
“km”	kilometers
“li”	links
“m”	meters
“mi”	miles
“mm”	millimeters
“nmi”	nautical miles (1 nautical mile represents 1852 meters)
“rd”	rods

distance value	Unit Represented
“survey ft”	U.S. survey feet (used for 1927 State Plane coordinates; one U.S. Survey Foot equals exactly 12/39.37 meters, or approximately 30.48006 cm)
“yd”	yards

paper_units defaults to the MapBasic Paper Unit (see **Set Paper Units statement**).

paper_length a value in *paper_units* to specify how long the scale bar will be displayed. Specify the length of the scale bar to a maximum of 34 inches or 86.3 cm on the printed map.

paper_height a value in *paper_units* to specify how tall the scale bar will be displayed. Specify height of the adornment to a maximum of 44 inches or 111.76cm on the printed map.

Scale set to **On** to include a representative fraction (RF) with the scale bar. (In MapInfo Professional, a map scale that does not include distance units, such as 1:63,360 or 1:1,000,000, is called a **cartographic scale**.)

Font is a valid **Font clause**.

Description

The scale bar displays as a *paper_length* bar in the *paper_units*.

Example

If the *paper_length* is 1 and the *paper_unit* is inches, then the scale bar displays as 1 inch. It is labeled in the *distance_unit* for the current amount that *paper_unit* spans, and it dynamically updates as the map changes (e.g., zoom and pan). The default *distance_unit* is the current distance unit in the mapper. The *paper_height* determines how tall the scale bar displays. The **Pen** and **Brush** define the style to draw the scale bar with and **Font** defines the text style for scale bar labeling and annotation. The Scale parameter displays a cartographic scale.

The following example shows default settings:

```
create adornment
   from window 261763624
   type scalebar
   position 6 offset (0.000000, 0.000000) units "in"
   background Brush (2,16777215,16777215)  Pen (1,2,0)
   bartype 0 ground units "mi" display units "in"
   barlength 1.574803 barheight 0.078740
   barstyle Pen (1,2,0)  Brush (2,0,16777215)  Font ("Arial",0,8,0)
   scale on
```

See Also:

Set Adornment statement

Create Arc statement

Purpose

Creates an arc object. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Create Arc
   [ Into { Window window_id | Variable var_name } ]
   ( x1, y1 ) ( x2, y2 )
   start_angle end_angle
   [ Pen... ]
```

window_id is a window identifier.

var_name is the name of an existing object variable.

x1, *y1* specifies one corner of the minimum bounding rectangle (MBR) of an ellipse; the arc produced will be a section of this ellipse.

x2, *y2* specifies the opposite corner of the ellipse's MBR.

start_angle specifies the arc's starting angle, in degrees.

end_angle specifies the arc's ending angle, in degrees.

The **Pen clause** specifies a line style.

Description

The **Create Arc** statement creates an arc object.

If the statement includes the optional **Into Variable** clause, the object will be stored in the specified object variable. If the **Into** clause specifies a window identifier, the object will be stored in the appropriate place in the window (for example, in the editable layer of a Map window). If the **Into** clause is not provided, MapBasic will attempt to store the object in the topmost window; if objects may not be stored in the topmost window (for example, if the topmost window is a grapher) no object will be created.

The *x* and *y* parameters use whatever coordinate system MapBasic is currently using. By default, MapBasic uses a Longitude/Latitude coordinate system, although the **Set CoordSys statement** can re-configure MapBasic to use a different coordinate system. Note that MapBasic's coordinate system is independent of the coordinate system of any Map window. Objects created on a Layout window, however, are specified in paper units: each x-coordinate represents a distance from the left edge of the page, while each y-coordinate represents the distance from the top edge of the page. By default, MapBasic uses inches as the default paper unit. To use a different paper unit, use the **Set Paper Units statement**. Before creating objects on a Layout window, you must issue a Set CoordSys Layout statement.

The optional **Pen** clause specifies a line style. If no Pen clause is specified, the **Create Arc** statement uses the current MapInfo Professional line style (the style which appears in the **Options > Line Style** dialog box).

See Also:

Insert statement, **Pen clause**, **Update statement**, **Set CoordSys statement**

Create ButtonPad statement

Purpose

Creates a ButtonPad (toolbar). You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Create ButtonPad { title_string | ID pad_num } As
      button_definition [ button_definition ... ]
   [ Title title_string ]
   [ Width w ]
   [ Position ( x, y ) [ Units unit_name ] ]
   [ ToolbarPosition ( row, column ) ]
   [ { Show | Hide } ]
   [ { Fixed | Float | Top | Left | Right | Bottom } ]
```

title_string is the ButtonPad title (for example, “Drawing”).

pad_num is the ID number for the standard toolbar you want to re-define:

- 1 for Main
- 2 for Drawing
- 3 for Tools
- 4 for Standard
- 5 for Database Management System (DBMS)
- 6 Web Services
- 7 Reserved

w is the pad width, in terms of the number of buttons across.

x, *y* specify the pad's position when it is floating; specified in paper units (for example, inches).

unit_name is a string representing paper units name (for example, “in” for inches, “cm” for centimeters).

row, *column* specify the pad's position when it is docked as a toolbar (for example, 0, 0 places the pad at the left edge of the top row of toolbars, and 0, 1 represents the second pad on the top row).

- *row* position starts at the top and increases in value going to the bottom. It is a relative value to the rows existing in the same position (top or bottom). When there is a menu bar in the same position, then the numbers become relative to the menu bar. When a toolbar is just

below the menu bar, its row value is 0. If it is directly above the menu bar, then its row value is -1.

- *column* position starts at the left and increases in value going to the right. It is a relative value to the columns existing in the same position (left or right). For example, if a toolbar is docked to the left and the menu bar is docked to the left position, then the column number for the column left of the menu bar is -1. The column number for the column to the right of the menu bar is 0.

Each *button_definition* clause can consist of the keyword **Separator**, or it can have the following syntax:

```
{ PushButton | ToggleButton | ToolButton }
   Calling { procedure | menu_code | OLE methodname | DDE server, topic }
   [ ID button_id ]
   [ Icon n [ File file_spec ] ]
   [ Cursor n [ File file_spec ] ]
   [ DrawMode dm_code ]
   [ HelpMsg msg ]
   [ ModifierKeys { On | Off } ]
   [ Enable ] [ Disable ]
   [ Check ] [ Uncheck ]
```

procedure is the handler procedure to call when a button is used.

menu_code is a standard MapInfo Professional menu code from MENU.DEF (for example, M_FILE_OPEN); MapInfo Professional runs the menu command when the user uses the button.

methodname is a string specifying an OLE method name.

server, *topic* are strings specifying a DDE server and topic name.

ID *button_id* specifies a unique button number. This number can be used as a parameter to allow a handler to determine which button is in use (in situations where different buttons call the same handler) or as a parameter to be used with the **Alter Button statement**.

Icon *n* specifies the icon to appear on the button; *n* can be one of the standard MapInfo icon codes listed in ICONS.DEF (for example, MI_ICON_RULER). If the **File** sub-clause specifies the name of a file containing icon resources, *n* is an integer resource ID identifying a resource in the file. The size of the button can be defined with resource file id of n for small and n+1 for large sized buttons, with resource file ids of n and n+1 respectively.

Cursor *n* specifies the shape the mouse cursor should adopt whenever the user chooses a ToolButton tool; *n* is a cursor code (for example, MI_CURSOR_ARROW) from ICONS.DEF. This clause applies only to ToolButtons. If the **File** sub-clause specifies the name of a file containing icon resources, *n* is an integer resource ID identifying a resource in the file.

DrawMode *dm_code* specifies whether the user can click and drag, or only click with the tool; *dm_code* is a code (for example, DM_CUSTOM_LINE) from ICONS.DEF. The **DrawMode** clause applies only to ToolButtons.

HelpMsg *msg* specifies the button's status bar help and, optionally, ToolTip help. The first part of the *msg* string is the status bar help message. If the *msg* string includes the letters `\n` then the text following the `\n` is used as the button's ToolTip help.

ModifierKeys clause controls whether the shift and control keys affect “rubber-band” drawing if the user drags the mouse while using a ToolButton. Default is **Off**, meaning that the shift and control keys have no effect.

Description

Use the **Create ButtonPad** statement to create a custom ButtonPad. Once you have created a custom ButtonPad, you can modify it using **Alter Button statement** and **Alter ButtonPad statement**.

Each toolbar can be hidden. To create a toolbar in the hidden state, include the **Hide** keyword.

To set whether the pad is fixed to the top of the screen (“docked”) or floating like a window, include the **Fixed** or the **Float** keyword. The user can also control whether the pad is docked or not by dragging the pad to or from the top of the screen. For more control over the location on the screen that the pad is docked to, use the **Top** (which is the same as using **Fixed**), **Left**, **Right**, or **Bottom** keywords.

When a toolbar is floating, its position is controlled by the **Position** clause; when it is docked, its position is controlled by the **ToolbarPosition** clause.

For more information on ButtonPads, see the *MapBasic User Guide*. For additional information about the capabilities of ToolButtons, see **Alter ButtonPad statement**.

About Icon Size

Before MapInfo Professional 10.0, custom icons were rectangular in size: 18x16 for small icons and 26x24 for large icons. Toolbar and menu features introduced in MapInfo Professional 10.0 require square icons: 16x16 for small icons and 24x24 for large icons. MapInfo Professional 10.0 and later display custom icons in the square size distorting how they display. To correct the distortion MapInfo Professional removes the first and last column of pixels to not display them (cropping the image). As a result, there may be some loss of information in the icon image.

For best results, create your icons to 16x16 for small icons and 24x24 for large icons. Icons at these sizes are not compatible with versions before 10.0.1 and generate an error message in these versions.

For compatibility with versions before 10.0.1, use 18x16 for small icons and 26x24 for large icons. These icons appear distorted in version 10.0 and appear cropped in versions after 10.0.

About Icons for Menu Items

The following describe how MapInfo Professional 10.0 handles icons for menu items:

- Icons for menu items are dynamic based on icons in toolbars.

- If a toolbar button has an icon it may be used for a menu item if it meets one of the following requirements:
  - Toolbar button and menu item must be within the same MBX file.
  - The same handler/userId combination; any menu item that calls the same handler/userid from the same MBX file is assigned that icon).
  - The same userId calling different handlers.
  - The same handler, no userID.
  - ID must be <= 32767.
- For built in handlers, priority is given to find an icon from a built in toolbar. If none is found, then there is a search through the toolbar icons from the MBX files for a match.
- Small icons are shown next to menu items.
- When an MBX unloads, any of its associated icons are unloaded and are no longer in use for menu items.

The following describe how MapInfo Professional 10.0.1 and later handles icons for menu items:

- There is no icon for the Table > Drive Regions menu option, because there is no corresponding toolbar button for this option. There is an icon and toolbar button for the Objects > Driving Regions menu option.
- Other menu items can now show an icon if a MapBasic application creates a toolbar button and menu item that meet the previously listed requirements for menu item icons.

Calling Clause Options

The **Calling** clause specifies what should happen when the user acts on the custom button. The following table describes the available syntax.

Calling clause example	Description
`Calling M_FILE_NEW`	If **Calling** is followed by a numeric code from MENU.DEF, the event runs a standard MapInfo Professional menu command (the **File > New** command, in this example).
`Calling my_procedure`	If you specify a procedure name, the event calls the procedure. The procedure must be part of the same MapBasic program.
`Calling OLE "methodname"`	Makes a method call to the OLE Automation object set by MapInfo Professional's SetCallback method. See the *MapBasic User Guide*.
`Calling DDE "server","topic"`	Connects through DDE to “server|topic” and sending an Execute message to the DDE server.

In the last two cases, the string sent to OLE or DDE starts with the three letters “MI:” so that the server can detect that the message came from MapInfo. The remainder of the string contains a comma-separated list of the values returned from the function calls CommandInfo(1) through CommandInfo(8). For complete details on the string syntax, see the *MapBasic User Guide*.

Examples

Create a button pad of utilities:

```
Create ButtonPad "Utils" As
   PushButton
      HelpMsg "Choose this button to display query dialog"
      Calling button_sub_proc
      Icon MI_ICON_ZOOM_QUESTION
   ToolButton
      HelpMsg "Use this tool to draw a new route"
      Calling tool_sub_proc
      Icon MI_ICON_CROSSHAIR
      DrawMode DM_CUSTOM_LINE
   ToggleButton
      HelpMsg "Turn proximity checking on/off"
      Calling toggle_prox_check
      Icon MI_ICON_RULER
      Check
   Title "Utilities"
   Width 3
   Show
```

Create a toolbar button that launches the Browser Preferences dialog, which has a menu command ID of 222:

```
Create ButtonPad "Prefs" As
   PushButton
      HelpMsg "Browser Preferences.\nBrowser Prefs"
      Calling 222
      Icon 99
```

See Also:

Alter Button statement, **Alter ButtonPad statement**, **ButtonPadInfo( ) function**

Create ButtonPad As Default statement

Purpose

Restores one standard ButtonPad (for example, the Main ButtonPad) to its default state. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Create ButtonPad { *title_string* | **ID** *pad_num* } **As Default**

title_string is the ButtonPad title (for example, "Main", "Standard", or "Custom Tools").

pad_num is the ID number for the standard ButtonPad (toolbar) you want to re-define:

- 1 for Main
- 2 for Drawing

- 3 for Tools
- 4 for Standard
- 5 for Database Management System (DBMS)
- 6 Web Services
- 7 Reserved

Custom ButtonPads use only the *title_string*.

Description

This statement restores MapInfo Professional's standard ButtonPads (such as Main, Drawing, and Tools) to their default states. Custom ButtonPads will be destroyed.

See Also:

Alter Button statement,**Alter ButtonPad statement**, **Create ButtonPad statement**, **Create ButtonPads As Default statement**

Create ButtonPads As Default statement

Purpose

Restores all standard ButtonPads (for example, the Main ButtonPad) to their default state. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Create ButtonPads As Default
```

Description

This statement restores MapInfo Professional's standard ButtonPads (such as Main, Drawing, and Tools) to their default states. Custom ButtonPads will be destroyed.

Use this statement with caution. The **Create ButtonPads As Default** statement destroys all custom buttons, even buttons defined by other MapBasic applications.

See Also:

Alter Button statement,**Alter ButtonPad statement**, **Create ButtonPad statement**, **Create ButtonPad As Default statement**

Create Cartographic Legend statement

Purpose

Creates and displays cartographic style legends as well as theme legends for an active map window. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Create Cartographic Legend
   [ From Window map_window_id ]
   [ Behind ]
   [ Position ( x, y ) [ Units paper_units ] ]
   [ Width win_width [ Units paper_units ] ]
   [ Height win_height [ Units paper_units ] ]
   [ Window Title { legend_window_title }
   [ ScrollBars { On | Off } ]
   [ Portrait | Landscape | Custom ]
   [ Style Size { Small | Large }
   [ Default Frame Title { def_frame_title } [ Font... ] } ]
   [ Default Frame Subtitle { def_frame_subtitle } [ Font... ] } ]
   [ Default Frame Style { def_frame_style } [ Font... ] } ]
   [ Default Frame Border Pen [ [ pen_expr ]
   Frame From Layer { map_layer_id | map_layer_name
      [ Using
         [ Column { column | Object } [ FromMapCatalog { On | Off }]]
         [ Label { expression | Default } ]
      [ Position ( x, y ) [ Units paper_units ] ]
      [ Title { frame_title [ Font... ] }
      [ SubTitle { frame_subtitle [ Font... ] } ]
      [ Border Pen pen_expr ]
      [ Style [ Font...] [ Norefresh ] [ Text { style_name }
         { Line Pen... | Region Pen... Brush...| Symbol Symbol... } |
            Collection [ Symbol ... ]
      [ Line Pen... ] [ Region Pen... Brush ...] } ]
   [ , ... ]
```

map_window_id is an integer window identifier which you can obtain by calling the **FrontWindow( ) function** and **WindowID( ) function**.

x states the desired distance from the top of the workspace to the top edge of the window.

y states the desired distance from the left of the workspace to the left edge of the window.

Here workspace means the client area (which excludes the title bar, tool bar, and the status bar).

paper_units is a string representing a paper unit name (for example, “cm” for centimeters).

win_width is the desired width of the window.

win_height is the desired height of the window.

legend_window_title is a string expression representing a title for the window, defaults to “Legend of *xxx*” where *xxx* is the map window title.

def_frame_title is a string which defines a default frame title. It can include the special character “#” which will be replaced by the current layer name.

def_frame_subtitle is a string which defines a default frame subtitle. It can include the special character “#” which will be replaced by the current layer name.

def_frame_style is a string that displays next to each symbol in each frame. The "#" character will be replaced with the layer name. The % character will be replaced by the text "Line", "Point, "Region", as appropriate for the symbol. For example, "% of #" will expand to "Region of States" for the STATES.TAB layer.

pen_expr is a Pen expression, for example, MakePen(*width*, *pattern*, *color*). If a default border pen is defined, then it will be become the default for the frame. If a border pen clause exists at the frame level, then it is used instead of the default.

map_layer_id or *map_layer_name* identifies a map layer; can be a SmallInt (for example, use 1 to specify the top map layer other than Cosmetic) or a string representing the name of a table displayed in the map. For a theme layer you must specify the *map_layer_id*.

frame_title is a string which defines a frame title. If a **Title** clause is defined here for a frame, then it will be used instead of the *def_frame_title*.

frame_subtitle is a string which defines a frame subtitle. If a **Subtitle** clause is defined here for a frame, then it will be used instead of the *def_frame_subtitle*.

column is an attribute column name from the frame layer's table.

style_name is a string which displays next to a symbol, line, or region in a custom frame.

Description

The **Create Cartographic Legend** statement allows you to create and display cartographic style legends as well as theme legends for an active map window. Each cartographic and thematic styles legend will be connected to one, and only one, Map window so that there can be more than one Legend window open at a time.

You can create a frame for each cartographic or thematic map layer you want to include on the legend. The cartographic and thematic frames will include a legend title and subtitle. Cartographic frames display a map layer's styles; legend frames display the colors, symbols, and sizes represented by the theme. You can create frames that have styles based on the Map window's style or you can create your own custom frames.

At least one **Frame** clause is required.

All clauses pertaining to the entire legend (scrollbars, width, etc.) must proceed the first **Frame** clause.

The **From Layer** clause must be the first clause after the **Frame** clause.

The optional **Behind** clause places the legend behind the Thematic Map window.

The optional **Position** clause controls the window's position on MapInfo Professional's workspace. The upper left corner of MapInfo Professional's workspace has the position 0, 0. The optional **Width** and **Height** clauses control the window's size. Window position and size values use paper units settings, such as "in" (inches) or "cm" (centimeters). MapBasic has a current paper units setting, which defaults to inches; a MapBasic program can change this setting through the **Set Paper Units statement**. A **Create Cartographic Legend** statement can override the current paper units by including the optional **Units** subclause within the **Position**, **Width**, and/or **Height** clauses.

Use the **ScrollBars** clause to show or hide scroll-bars on a Map window.

Portrait or **Landscape** describes the orientation of the legend frames in the window. **Portrait** results in an orientation that is down and across. **Landscape** results in an orientation that is across and down.

If **Custom** is specified, you can specify a custom **Position** clause for a frame.

The **Position** clause at the frame level specifies the position of a frame if **Custom** is specified.The x coordinate measures from the left of the legend window, and the y coordinate measures from the bottom of the legend window (the origin (0,0) is in the bottom-left of the legend window).

The optional **Style Size** clause controls the size of the samples that appear in legend windows. If you specify **Style Size Small**, small-sized legend samples are used in Legend windows. If you specify **Style Size Large**, larger-sized legend samples are used.

The **Position**, **Title**, **SubTitle**, **Border Pen**, and **Style** clauses at the frame level are used only for map layers. They are not used for thematic layers. For a thematic layer, this information is gotten automatically from the theme.

The **Font** clause specifies a text style. If a default frame title, subtitle, or style name font is defined, then it will become the default for the frame. If a frame level **Title**, **Subtitle**, or **Style** clause exists and includes a **Font** clause, then the frame level font is used. If no font is specified at any level, then the current text style is used and the point sizes are 10, 9, and 8 for title, subtitle and style name.

The **Style** clause and the **NoRefresh** keyword allow you to create custom frames that are not overwritten when the legend is refreshed. If the **NoRefresh** keyword is used in the **Style** clause, then the table is not scanned for styles. Instead, the **Style** clause must contain your custom list of definitions for the styles displayed in the frame. This is done with the **Text** clause and appropriate **Line**, **Region**, or **Symbol** clause. Multipoint objects are treated as Point objects.

Collection objects are treated separately. When MapInfo Professional creates a Legend based on object types, it draws Point symbols first, then Lines, then Regions. Collection objects are drawn last. Inside collection objects the order of drawing is point, line, and then region samples.

If Column is defined, *column* is the name of an attribute column in the frame layer's table, or **Object** denotes the object column (meaning that legend styles are based on the unique styles in the map file). The default is **Object**.

FromMapCatalog ON retrieves styles from the MapCatalog for a live access table. If the table is not a live access table, MapBasic reverts to the default behavior for a non-live access table instead of throwing an error. The default behavior for a non-access table is **FromMapCatalog Off** (for example, map styles).

FromMapCatalog OFF retrieves the unique map styles for the live table from the server. This table must be a live access table that supports per record styles for this to occur. If the live table does not support per record styles than the behavior is to revert to the default behavior for live tables, which is to get the default styles from the MapCatalog (**FromMapCatalog ON**).

If a Label is defined, specify *expression* as a valid expression, or **Default** (meaning that the default frame style pattern is used when creating each style's text, unless the style clause contains text). The default is **Default**.

Initially, each frame layer's TAB file is searched for metadata values for the title, subtitle, column and label. If no metadata value exists for the column, the default is **Object**. If no metadata value exists for Label, the default is the default frame style pattern. If legend metadata keys exist and you want to override them, you must use the corresponding MapBasic syntax.

Example

The following example shows how to create a frame for a Map window's cartographic legend. Legend windows are a special case: To create a frame for a Legend window, you must use the **Title** clause instead of the **From Window** clause.

```
Dim i_layout_id, i_map_id As Integer
Dim s_title As String
' here, you would store the Map window's ID in i_map_id,
' and store the Layout window's ID in i_layout_id.
' To obtain an ID, call FrontWindow( ) or WindowID( ).
s_title = "Legend of " + WindowInfo(i_map_id, WIN_INFO_NAME)
Set CoordSys Layout Units "in"
Create Frame
   Into Window i_layout_id
   (1,2) (4, 5)
   Title s_title
```

This creates a frame for a Cartographic Legend window. To create a frame for a Thematic Legend window, change the title to the following.

```
S_title="Theme Legend of " + WindowInfo (I_map_id, WW_INFO_NAME)
```

See Also:

Set Cartographic Legend statement, **Alter Cartographic Frame statement**, **Add Cartographic Frame statement**, **Remove Cartographic Frame statement**, **Create Legend statement**, **Set Window statement**, **WindowInfo( ) function**

CreateCircle( ) function

Purpose

Returns an Object value representing a circle. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

CreateCircle(*x*, *y*, *radius*)

x is a float value, indicating the x-position (for example, Longitude) of the circle's center.

y is a float value, indicating the y-position (for example, Latitude) of the circle's center.

radius is a float value, indicating the circle radius.

Return Value

Object

Description

The **CreateCircle( )** function returns an Object value representing a circle.

The *x* and *y* parameters use whatever coordinate system MapBasic is currently using. By default, MapBasic uses a Longitude/Latitude coordinate system, although the **Set CoordSys statement** can re-configure MapBasic to use a different coordinate system.

MapBasic's coordinate system is independent of the coordinate system of any Map window.

The *radius* parameter specifies the circle radius, in whatever distance unit MapBasic is currently using. By default, MapBasic uses miles as the distance unit, although the **Set Distance Units statement** can re-configure MapBasic to use a different distance unit.

The circle uses whatever Brush style is currently selected. To create a circle object with a specific Brush, you can issue a **Set Style statement** before calling **CreateCircle( )**. Alternately, instead of calling **CreateCircle( )**, you can issue a **Create Ellipse statement**, which has optional **Pen clause** and **Brush clause**.

The circle object created through the **CreateCircle( )** function could be assigned to an Object variable, stored in an existing row of a table (through the **Update statement**), or inserted into a new row of a table (using an **Insert statement**).

Before creating objects on a Layout window, you must issue a **Set CoordSys Layout statement**.

Error Conditions

ERR_FCN_ARG_RANGE (644) is generated if an argument is outside of the valid range.

Examples

The following example uses the **Insert statement** to insert a new row into the table Sites. The **CreateCircle( )** function is used within the body of the Insert statement to specify the graphic object that is attached to the new row.

```
Open Table "sites"
Insert Into sites (obj)
   Values ( CreateCircle(-72.5, 42.4, 20) )
```

The following example assumes that the table Towers has three columns: Xcoord, Ycoord, and Radius. The Xcoord column contains longitude values, the Ycoord column contains latitude values, and the Radius column contains radius values. Each row in the table describes a radio broadcast tower, and the Radius column indicates each tower's broadcast area.

The **Update statement** uses the **CreateCircle( )** function to build a circle object for each row in the table. Following this Update statement, each row in the Towers table will have a circle object attached. Each circle object will have a radius derived from the Radius column, and each circle will be centered at the position indicated by the Xcoord and Ycoord columns.

```
Open Table "towers"
Update towers
   Set obj = CreateCircle(xcoord, ycoord, radius)
```

See Also:

Create Ellipse statement, **Insert statement**, **Update statement**

Create Collection statement

Purpose

Combines points, linear objects, and closed objects into a single object. The collection object displays in the Browser as a single record. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Create Collection [ num_parts ]
   [ Into { Window window_id | Variable var_name } ]
   Multipoint
      [ num_points ]
      ( x1, y1 ) ( x2, y2 ) [ ... ]
      [ Symbol... ]
   Region
      num_polygons
      [ num_points1 ( x1, y1 ) ( x2, y2 ) [ ... ] ]
      [ num_points2 ( x1, y1 ) ( x2, y2 ) [ ... ] ... ]
      [ Pen... ]
      [ Brush... ]
      [ Center ( center_x, center_y ) ]
   Pline
      [ Multiple num_sections ]
      num_points
      ( x1, y1 ) ( x2, y2 ) [ ... ]
      [ Pen... ]
      [ Smooth... ]
```

num_parts is the number of non-empty parts inside a collection. This number is from 0 to 3 and is optional for MapBasic code (it is mandatory for MIF files).

num_polygons is the number of polygons inside the Collection object.

num_sections specifies how many sections the multi-section polyline will contain.

Pen is a valid **Pen clause** to specify a line style.

Brush is a valid **Brush clause** to specify fill style.

Example

```
create collection multipoint 2 (0,0) (1,1) region 3 3 (1,1) (2,2) (3,4) 4
(11,11) (12,12) (13,14) (19,20) 3 (21,21) (22,22) (23,24) pline 3 (-1,1)
(3,-2) (4,3)
dim a as object
create collection into variable a multipoint 2 (0,0) (1,1) region 1 3
(1,1) (2,2) (3,4) pline 3 (-1,1) (3,-2) (4,3)
insert into test (obj) values (a)
create collection region 2 4 (-5,-5) (5,-5) (5,5) (-5,5) 4 (-3,-3) (3,-3)
(3,3) (-3,3) pline multiple 2 2 (-6,-6) (6,6) 2 (-6,6) (6,-6) multipoint 6
(2,2) (-2,-2) (2,-2) (-2,2) (4,1) (-1,-4)
```

See Also:

Create MultiPoint statement

Create Cutter statement

Purpose

Produces a Region object that can be used as a cutter for an Object Split operation, as well as a new set of Target objects which may be a subset of the original set of Target objects. You need to provide a set of Target objects, and a set of polylines as a selection object. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Create Cutter Into Target
```

Description

Before using Create Cutter, one or more Polyline objects must be selected, and an editable target must exist. This is set by choosing **Objects > Set Target**, or using the **Set Target statement**. The Polyline objects contained in the selection must represent a single, contiguous section. The Polyline selection must contain no breaks or self intersections.

The Polyline must intersect the Minimum Bounding Rectangle (MBR) of the Target in order for the Target to be a valid object to split. The Polyline, however, does not have to intersect the Target object itself. For example, the Target object could be a series of islands (for example, Hawaii), and the Polyline could be used to divide the islands into two sets without actually intersecting any of the islands. If the MBR of a Target does not intersect the Polyline, then that Target will be removed from the Target list.

Given this revised set of Target objects, a cumulative MBR of all of these objects is calculated and represents the overall space to be split. The polyline is then extended, if necessary, so that it covers the MBR. This is done by taking the direction of the last two points on each end of the polyline and extending the polyline in that Cartesian direction until it intersects with the MBR. The extended Polyline should divide the Target space into two portions. One Region object will be created and returned which represents one of these two portions.

This statement returns the revised set of Target objects (still set as the Target), as well as this new Region cutter object. This Region object will be inserted into the Target table (which must be an editable table). The original Polyline object(s) will remain, but will no longer be selected. The new Region object will now be the selected object. If the resulting Region object is suitable, then this operation can be immediately followed by an Object Split operation, as appropriate Target objects are set, and a suitable Region cutter object is selected.

The cutter object still remains in the target layer. You will have to delete the cutter object manually from your editable layer.

Example

```
Open Table "C:\MapInfo_data\TUT_USA\USA\STATES.TAB"
Open Table "C:\MapInfo_data\TUT_USA\USA\US_HIWAY.TAB"
Map from States, Us_hiway
select * from States where state = "NY"
Set target On
select * from Us_hiway where highway = "I 90"
Create Cutter Into Target
Objects Split Into Target
```

See Also:

Set Target statement

Create Ellipse statement

Purpose

Creates an ellipse or circle object. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Create Ellipse
   [ Into { Window window_id | Variable var_name } ]
   ( x1, y1 ) ( x2, y2 )
   [ Pen... ]
   [ Brush... ]
```

window_id is a window identifier.

var_name is the name of an existing object variable.

x1, *y1* specifies one corner of the rectangle which the ellipse will fill.

x2, *y2* specifies the opposite corner of the rectangle.

Pen is a valid **Pen clause** to specify a line style.

Brush is a valid **Brush clause** to specify fill style.

Description

The **Create Ellipse** statement creates an ellipse or circle object. If the object's Minimum Bounding Rectangle (MBR) is defined in such a way that the x-radius equals the y-radius, the object will be a circle; otherwise, the object will be an ellipse.

If the statement includes the optional **Into Variable** clause, the object will be stored in the specified object variable. If the **Into** clause specifies a window identifier, the object will be stored in the appropriate place in the window (for example, in the editable layer of a Map window). If the **Into** clause is not provided, MapBasic attempts to store the object in the topmost window; if objects may not be stored in the topmost window (for example, if the topmost window is a grapher) no object will be created.

The *x* and *y* parameters use whatever coordinate system MapBasic is currently using. By default, MapBasic uses a Latitude/Longitude coordinate system, although the **Set CoordSys statement** can re-configure MapBasic to use a different coordinate system. Note that MapBasic's coordinate system is independent of the coordinate system of any Map window. Objects created on a Layout window, however, are specified in paper units: each x-coordinate represents a distance from the left edge of the page, while each y-coordinate represents the distance from the top edge of the page. By default, MapBasic uses inches as the default paper unit. To use a different paper unit, use the **Set Paper Units statement**. Before creating objects on a Layout window, you must issue a Set CoordSys Layout statement.

The optional **Pen clause** specifies a line style. If no Pen clause is specified, the **Create Ellipse** statement uses the current MapInfo Professional line style (the style which appears in the **Options > Line Style** dialog box). Similarly, the optional **Brush clause** specifies a fill style.

See Also:

Brush clause, **CreateCircle( ) function**, **Insert statement**, **Pen clause**, **Update statement**

Create Frame statement

Purpose

Creates a new frame in a Layout window. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Create Frame
   [ Into { Window layout_win_id | Variable var_name } ]
   ( x1, y1 ) ( x2, y2 )
   [ Pen... ]
   [ Brush... ]
   [ Title title ]
   [ From Window contents_win_id ]
   [ FillFrame { On | Off } ]
```

x1, *y1* specifies one corner of the new frame to create.

x2, *y2* specifies the other corner.

layout_win_id is a Layout window's integer window identifier.

var_name is the name of an Object variable.

Pen is a valid **Pen clause** to specify a line style.

Brush is a valid **Brush clause** to specify fill style.

title is a string identifying the frame contents (for example, “WORLD Map”); not needed if the **From Window** clause is used.

contents_win_id is an integer window ID indicating which window will appear in the frame.

Description

The **Create Frame** statement creates a new frame within an existing Layout window. If no *layout_win_id* is specified, the new frame is added to the topmost Layout window. Before creating objects on a Layout window, you must issue a Set CoordSys Layout statement.

Between sessions, MapInfo Professional preserves Layout window settings by storing **Create Frame** statements in the workspace file. To see an example of the **Create Frame** statement, create a Layout, save the workspace, and examine the workspace file in a text editor.

The **Pen clause** dictates what line style will be used to display the frame, and the **Brush clause** dictates the fill style used to fill the frame window.

Use the **From Window** clause to specify which window should appear inside the frame. For example, to make a Map window appear inside the frame, specify **From Window** *i_map* (where *i_map* is an integer variable containing the Map's window identifier). A window must already be open before you can create a frame containing the window.

The **Title** clause provides an alternate syntax for specifying which window appears in the frame. For example, to identify a Map window which displays the table WORLD, the **Title** clause should read `Title "WORLD Map"`. If the *title* string does not refer to an existing window, or if *title* is an empty string (`""`), the frame will be empty. If you specify both the **Title** clause and the **From Window** clause, the latter clause takes effect.

The **FillFrame** clause controls how the window fills the frame. If you specify **FillFrame On**, the entire frame is filled with an image of the window. (This is analogous to checking the **Fill Frame With Contents** check box in MapInfo Professional's Frame Object dialog box, which appears if you double-click a frame.) If you specify **FillFrame Off** (or if you omit the **FillFrame** clause entirely), the aspect ratio of the window affects the appearance of the frame; in other words, re-sizing a Map window to be tall and thin causes the frame to appear tall and thin.

Example

The following examples show how to create a frame for a Map window's thematic legend, or Cartographic Legend window.

Theme Legend windows are a special case. To create a frame for a Theme Legend window, you must use the **Title** clause instead of the **From Window** clause.

```
Dim i_layout_id, i_map_id As Integer
Dim s_title As String
```

```
' here, you would store the Map window's ID in i_map_id,
' and store the Layout window's ID in i_layout_id.
' To obtain an ID, call FrontWindow( ) or WindowID( ).

s_title = "Theme Legend of " + WindowInfo(i_map_id, WIN_INFO_NAME)
Set CoordSys Layout Units "in"
Create Frame
   Into Window i_layout_id
   (1,2) (4, 5)
   Title s_title
```

To create a frame for a Map window's cartographic legend, you should use the **From Window** clause since there may be more than one cartographic legend window per map.

```
Dim i_cartlgnd_id As Integer

' here, you would store the Cartographic Legend window's ID
' in i_cartlgnd _id,
' To obtain an ID, call FrontWindow( ) or WindowID( ).

Create Frame
   Into Window i_layout_id
   (1,2) (4, 5)
   From Window i_cartlgnd_id
```

See Also:

Brush clause, **Insert statement**, **Layout statement**, **Pen clause**, **Set CoordSys statement**, **Set Layout statement**, **Update statement**

Create Grid statement

Purpose

Produces a raster grid file, which MapBasic displays as a raster table in a Map window. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Create Grid
   From tablename
   With expression [ Ignore value_to_ignore ]
   Into filespec [ Type grid_type ]
      [ Coordsys... ]
   [ Clipping { Object obj } | { Table tablename } ]
   Inflect num_inflections By Percent at
      color : inflection_value [ color : inflection_value ...]
   [ Round rounding_factor ]
   {[ Cell Size cell_size [ Units distance_unit ]] | [ Cell Min n_cells ]}
       [ Border numcells ]
   Interpolate With interpolator_name Version version_string
```

```
Using num_parameters parameter_name : parameter_value
[ parameter_name : parameter_value ... ]
```

tablename is the “alias” name of an open table from which to get data points.

expression is the expression by which the table will be shaded, such as a column name.

value_to_ignore is a value to be ignored; this is usually zero. No grid theme will be created for a row if the row's value matches the value to be ignored.

filespec specifies the fully qualified path and name of the new grid file. It will have a .MIG extension.

grid_type is a string expression that specifies the type of grid file to create. By default, .MIG files are created.

Coordsys is an optional CoordSys clause which is the coordinate system that the grid will be created in. If not provided, the grid will be created in the same coordinate system as the source table. Refer to **CoordSys clause** for more information.

obj is an object to clip grid cells to. Only the portion of the grid theme within the object will display. If a grid cell is not within the object, that cell value will not be written out and a null cell is written in its place.

tablename is the name of a table of region objects which will be combined into a single region object and then used for clipping grid cells.

num_inflections is a numeric expression, specifying the number of *color*:*inflection_value* pairs.

color is a color expression of, part of a *color:value inflection* pair.

inflection_value is a numeric expression, specifying the value of a *color*:*inflection_value* pair.

cell_size is a numeric expression, specifying the size of a grid cell in distance units.

n_cells is a numeric expression that specifies the height or width of the grid in cells.

numcells defines the number of cells to be added around the edge of the original grid bounds. *numcells* will be added to the left, right, top, and bottom of the original grid dimensions.

distance_unit is a string expression, specifying the units for the preceding cell size. This is an optional parameter. If not present, the distance units from the table's coordinate system are used.

interpolator_name is a string expression, specifying the name of the interpolator to use to create the grid.

version_string is a string expression, specifying the version of the interpolator that the parameters are meant for.

num_parameters is a numeric expression, specifying the number of interpolator parameter name:value pairs.

parameter_name is a string expression, specifying the name part of a *parameter_name*:*parameter_value* pair.

parameter_value is a numeric expression, specifying the value part of a *parameter_name*:*parameter_value* pair.

By Percent at specifies that the subsequent *color:Inflection_value* pairs represent a color value and percentage value.

Round is a numeric expression, specifying the rounding factor applied to the inflection values.

Description

A grid surface theme is a continuous raster grid produced by an interpolation of point data. The **Create Grid** statement takes a data column from a table of points, and passes those points and their data values to an interpolator. The interpolator produces a raster grid file, which MapBasic displays as a raster table in a Map window.

The **Create Grid** statement reads (*x*, *y*, *z*) values from the table specified in the **From** clause. It gets the z values by evaluating the expression specified in the **With** clause with respect to the table.

The dimensions of the grid can be specified in two ways. One is by specifying the size of a grid cell in distance units, such as miles. The other is by specifying a minimum height or width of the grid in terms of grid cells. For example, if you wanted the grid to be at least 200 cells wide by 200 cells high, you would specify “cell min 200". Depending on the aspect ratio of the area covered by the grid, the actual grid dimensions would not be 200 by 200, but it would be at least that wide and high.

Example

```
Open Table "C:\States.tab" Interactive
Map From States
Open Table "C:\Us_elev.tab" Interactive
Add Map Auto Layer Us_elev
set map redraw off
Set Map Layer 1 Display Off
set map redraw on

create grid
   from Us_elev
   with Elevation_FT
   into "C:\Us_elev_grid"
   clipping table States
   inflect 5 at
      RGB(0, 0, 255) : 13
      RGB(0, 255, 255) : 3632.5
      RGB(0, 255, 0) : 7252
      RGB(255, 255, 0) : 10871.5
      RGB(255, 0, 0) : 14491
   cell min 200
   interpolate
      with "IDW" version "100"
      using 4
         "EXPONENT": "2"
         "MAX POINTS": "25"
         "MIN POINTS": "1"
         "SEARCH RADIUS": "100"
```

See Also:

Set Map statement

Create Index statement

Purpose

Creates an index for a column in an open table. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Create Index On *table* (*column*)

table is the name of an open table.

column is the name of a column in the open table.

Description

The **Create Index** statement creates an index on the specified column. MapInfo Professional uses Indexes in operations such as **Query > Find**. Indexes also improve the performance of queries in general.

MapInfo Professional cannot create an index if the table has unsaved edits. Use the **Commit Table statement** to save edits.

Example

The following example creates an index for the “Capital” field of the World table.

```
Open Table "world" Interactive
Create Index on World(Capital)
```

See Also:

Alter Table statement, **Create Table statement**, **Drop Index statement**, **Commit Table statement**

Create Legend statement

Purpose

Creates a new Theme Legend window tied to the specified Map window. You can issue this statement from the MapBasic Window in MapInfo Professional.

For MapInfo Professional versions 5.0 and later, the Create Cartographic Legend statement allows you to create and display cartographic style legends. Refer to the **Create Cartographic Legend statement** for more information.

Syntax

```
Create Legend
   [ From Window window_ID ]
   [ { Show | Hide } ]
```

window_ID is an integer, representing a MapInfo Professional window ID for a Map window.

Description

This statement creates a special floating, Thematic Legend window, in addition to the standard MapInfo Professional Legend window. (To open MapInfo Professional's standard Legend window, use the Open Window Legend statement.)

The **Create Legend** statement is useful if you want the Legend of a Map window to always be visible, even when the Map window is not active. Also, this statement is useful in "Integrated Mapping" applications, where MapInfo Professional windows are integrated into another application, such as a Visual Basic application. For information about Integrated Mapping, see the *MapBasic User Guide*.

If you include the **From Window** clause, the new Theme Legend window is tied to the window that you specify; otherwise, the new window is tied to the most recently used Map.

If you include the optional **Hide** keyword, the window is created in a hidden state. You can then show the hidden window by using the **Set Window…Show** statement.

After you issue the **Create Legend** statement, determine the new window's integer ID by calling `WindowID( 0 )`. Use that window ID in subsequent statements (such as the **Set Window statement**).

The new Theme Legend window is created according to the parent and style settings that you specify through the **Set Next Document statement**.

See Also:

Create Cartographic Legend statement, **Open Window statement**, **Set Next Document statement**, **Set Window statement**

CreateLine( ) function

Purpose

Returns an Object value representing a line. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

```
CreateLine( x1, y1, x2, y2 )
```

x1 is a float value, indicating the x-position (for example,) of the line's starting point.

y1 is a float value, indicating the y-position (for example, Latitude) of the line's starting point.

x2 is a float value, indicating the x-position of the line's ending point.

y2 is a float value, indicating the y-position of the line's ending point.

Return Value

Object

Description

The **CreateLine( )** function returns an Object value representing a line. The *x* and *y* parameters use the current coordinate system. By default, MapBasic uses a Longitude/Latitude coordinate system. Use the **Set CoordSys statement** to choose a new system.

The line object will use whatever Pen style is currently selected. To create a line object with a specific Pen style, you could issue the **Set Style statement** before calling **CreateLine( )** or you could issue a **Create Line statement**, with an optional **Pen clause**.

The line object created through the **CreateLine( )** function could be assigned to an Object variable, stored in an existing row of a table (through the **Update statement**), or inserted into a new row of a table (through an **Insert statement**). If you need to create objects on a Layout window, you must first issue a **Set CoordSys Layout statement**.

Example

The following example uses the **Insert statement** to insert a new row into the table Routes. The **CreateLine( )** function is used within the body of the Insert statement.

```
Open Table "Routes"
Insert Into routes (obj)
   Values (CreateLine(-72.55, 42.431, -72.568, 42.435))
```

See Also:

Create Line statement, **Insert statement**, **Update statement**

Create Line statement

Purpose

Creates a line object. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Create Line
   [ Into { Window window_id | Variable var_name } ]
   ( x1, y1) ( x2, y2)
   [ Pen... ]
```

window_id is a window identifier.

var_name is the name of an existing object variable.

x1, *y1* specifies the starting point of a line.

x2, *y2* specifies the ending point of the line.

The **Pen clause** specifies a line style.

Description

The **Create Line** statement creates a line object.

If the statement includes the optional **Into Variable** clause, the object will be stored in the specified object variable. If the **Into** clause specifies a window identifier, the object will be stored in the appropriate place in the window (for example, in the editable layer of a Map window). If the **Into** clause is not provided, MapBasic will attempt to store the object in the topmost window; if objects may not be stored in the topmost window (for example, if the topmost window is a grapher) no object will be created.

The *x* and *y* parameters use whatever coordinate system MapBasic is currently using. By default, MapBasic uses a Longitude/Latitude coordinate system, although the **Set CoordSys statement** can re-configure MapBasic to use a different coordinate system. Note that MapBasic's coordinate system is independent of the coordinate system of any Map window. Objects created on a Layout window, however, are specified in paper units: each x-coordinate represents a distance from the left edge of the page, while each y-coordinate represents the distance from the top edge of the page. By default, MapBasic uses inches as the default paper unit. To use a different paper unit, use the **Set Paper Units statement**.

If you need to create objects on a Layout window, you must first issue a **Set CoordSys Layout statement**.

The optional **Pen clause** specifies a line style; see **Pen clause** for more details. If no Pen clause is specified, the **Create Line** statement will use the current MapInfo Professional line style.

See Also:

CreateLine( ) function, **Insert statement**, **Pen clause**, **Update statement**

Create Map statement

Purpose

Modifies the structure of a table, making the table mappable. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Create Map
   For table
   [ CoordSys... ] Using from_table]
```

table is the name of an open table.

from_table is the name of an open table from where to copy a coordinate system.

Description

The **Create Map** statement makes an open table mappable, so that it can be displayed in a Map window. This statement does not open a new Map window. To open a new Map window, use the **Map statement**.

You should not perform a **Create Map** statement on a table that is already mappable; doing so will delete all map objects from the table. If a table already has a map attached, and you wish to permanently change the projection of the map, use a **Table As** statement. Alternately, if you wish to temporarily change the projection in which a map is displayed, issue a **Set Map statement** with a **CoordSys clause**. The **Create Map** statement does not work on linked tables. To make a linked table mappable, use the **Server Create Map statement**.

Specifying the Coordinate System

Use one of the following two methods to specify a coordinate system:

- Provide the name of an already open mappable table as the *from_table* portion of the **Using** clause. In this case, the coordinate system used will be identical to that used in the *from_table*. The *from_table* must be a currently open table, and must be mappable or an error will occur.
- Explicitly supply the coordinate system information through a CoordSys clause (set in preferences). If you omit both the CoordSys clause and the Using clause, the table will use the current MapBasic coordinate system.

Note that the **CoordSys clause** affects the precision of the map. The **CoordSys clause** includes a **Bounds** clause, which sets limits on the minimum and maximum coordinates that can be stored in the map. If you omit the **Bounds** clause, MapInfo Professional uses default bounds that encompass the entire Earth (in which case, coordinates are precise to one millionth of a degree, or approximately 4 inches). If you know in advance that the map you are creating is limited to a finite area (for example, a specific metropolitan area), you can increase the precision of the map's coordinates by specifying bounds that confine the map to that area. For a complete listing of the **CoordSys** syntax, see **CoordSys clause**.

See Also:

Commit Table statement, **CoordSys clause**, **Create Table statement**, **Drop Map statement**, **Map statement**, **Server Create Map statement**, **Set Map statement**

Create Map3D statement

Purpose

Creates a 3DMap with the desired parameters. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Create Map3D
   [ From Window window_id | MapString mapper_creation_string ]
   [ Camera [ Pitch angle | Roll angle | Yaw angle | Elevation angle ] |
   [ Position ( x, y, z ) | FocalPoint ( x, y, z ) ] |
   [ Orientation ( vu_1, vu_2, vu_3, vpn_1, vpn_2, vpn_3,
```

```
   clip_near, clip_far )]]
[ Light [ Position ( x, y, z ) | Color lightcolor ] ]
[ Resolution ( res_x, res_y ) ]
[ Scale grid_scale ]
[ Background backgroundcolor ]
[ Units unit_name ]
```

window_id is a window identifier a for a Map window which contains a Grid layer. An error message is displayed if a Grid layer is not found.

mapper_creation_string specifies a command string that creates the mapper textured on the grid.

Camera specifies the camera position and orientation.

angle is an angle measurement in degrees. The horizontal angle in the dialog box ranges from 0-360 degrees and rotates the maps around the center point of the grid. The vertical angle in the dialog box ranges from 0-90 and measures the rotation in elevation from the start point directly over the map.

Pitch adjusts the camera's current rotation about the x axis centered at the camera's origin.

Roll adjusts the camera's current rotation about the z axis centered at the camera's origin.

Yaw adjusts the camera's current rotation about the y axis centered at the camera's origin.

Elevation adjusts the current camera's rotation about the x axis centered at the camera's focal point.

Position indicates the camera/light position.

FocalPoint indicates the camera/light focal point.

Orientation specifies the cameras ViewUp (*vu_1*, *vu_2*, *vu_3*), ViewPlane Normal (*vpn_1*, *vpn_2*, *vpn_3*), and Clipping Range (*clip_near*, *clip_far*) (used specifically for persistence of view).

Resolution is the number of samples to take in the x and y directions. These values can increase to a maximum of the grid resolution. The resolution values can increase to a maximum of the grid *x*, *y* dimension. If the grid is 200x200 then the resolution values will be clamped to a maximum of 200x200. You cannot increase the grid resolution, only specify a subsample value.

grid_scale is the amount to scale the grid in the z direction. A value >1 will exaggerate the topology in the z direction, a value <1 will scale down the topological features in the z direction.

backgroundcolor is a color to be used to set the background and is specified using the **RGB( ) function**.

unit_name specifies the units the grid values are in. Do not specify this for unit-less grids (for example, grids generated using temperature or density). This option needs to be specified at creation time. You cannot change them later with the **Set Map3D statement** or the Properties dialog box.

Description

Once it is created, the 3DMap window is a standalone window. Since it is based on the same tables as the original Map window, if these tables are changed and the 3DMap window is manually "refreshed" or re-created from a workspace, these changes are displayed on the grid. The creation fails if the *window_id* is not a Map window or if the Map window does not contain a Grid layer. If there are multiple grids in the Map window, each will be represented in the 3DMap window.

A 3DMap keeps a Mapper creation string as its texture generator. This string will also be prevalent in the workspace when the 3DMap window is persisted. The initialization will read in the grid layer to create 3D geometry and topology objects.

Example

```
Create Map3D Resolution(75,75)
```

Creates a 3DMap window of the most recent Map window. It will fail if the window does not contain any Continuous Grid layers. Another example is:

```
Create Map3D From Window FrontWindow( ) Resolution(100,100) Scale 2
Background RGB(255,0,0) Units "ft".
```

Creates a 3DMap window with a Red background, the z units set to feet, a Z scale factor of 2, and the grid resolution set to 100x100.

See Also:

Set Map3D statement

Create Menu statement

Purpose

Creates a new menu, or redefines an existing menu. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax 1

```
Create Menu newmenuname [ ID menu_id ] [ Context ] As
   menuitem [ ID menu_item_id ] [ HelpMsg help ]
   { Calling handler | As menuname }
   [ , menuitem ... ]
```

Syntax 2

```
Create Menu newmenuname As Default
```

newmenuname is a string representing the name of the menu to define or redefine.

menuitem is a string representing the name of an item to include on the new menu.

Context is reserved for internal use only; not usable in MapBasic programs.

menu_id is a SmallInt ID number from one to fifteen, identifying a standard menu.

menu_item_id is an integer ID number that identifies a custom menu item.

help is a string that appears on the status bar whenever the menu item is highlighted.

handler is the name of a procedure, or a code for a standard menu command, or a special syntax for handling the menu event by calling OLE or DDE; see **Calling Clause Options**. If you specify a command code for a standard MapInfo Professional Show/Hide command (such as M_WINDOW_STATISTICS), the *menuitem* string must start with an exclamation point and include a caret (^), to preserve the item's Show/Hide behavior.

menuname is the name of an existing menu to include as a hierarchical submenu.

Description

If the *newmenuname* parameter matches the name of an existing MapInfo Professional menu (such as **File**), the statement re-defines that menu. If the *newmenuname* parameter does not match the name of an existing menu, the **Create Menu** statement defines an entirely new menu. For a list of the standard MapInfo Professional menu names, see **Alter Menu statement**.

The **Create Menu** statement does not automatically display a newly-created menu; a new menu will only display as a result of a subsequent **Alter Menu Bar statement** or **Create Menu Bar statement**. However, if a **Create Menu** statement modifies an existing menu, and if that existing menu is already part of the menu bar, the change will be visible immediately.

MapInfo Professional can maintain no more than 96 menu definitions at one time, including the menus defined automatically by MapInfo Professional (**File**, etc.). This limit is independent of the number of menus displayed on the menu bar at one time.

The *menuitem* parameter identifies the name of the menu item. The item's name can contain special control characters to define menu item attributes (for example, whether a menu item is checkable). See tables below for details.

The following characters require special handling: slash (/), back slash(\), and less than (<). If you want to display any of these special characters in the menu or the status bar help, you must include an extra back slash in the *menuitem* string or the *help* string. For example, the following statement creates a menu item that reads, “Client/Server.”

```
Create Menu "Data" As
   "Client\/Server" Calling cs_proc
```

If a *menuitem* parameter begins with the character @, the custom menu breaks into two columns. The item whose name starts with @ is the first item in the second column.

Assigning Handlers to Custom Menu Items

Most menu items include the **Calling** *handler* clause; where *handler* is either the name of a MapBasic procedure or a numeric code identifying an MapInfo Professional operation (such as M_FILE_SAVE to specify the **File > Save** command). If the user chooses a menu item which has a handler, MapBasic automatically calls the handler (whether the handler is a sub procedure or a command code). Your program must Include the file MENU.DEF if you plan to refer to menu codes such as M_FILE_SAVE.

The optional **ID** clause lets you assign a unique integer ID to each custom menu item. Menu item IDs are useful if you want to allow multiple menu items to call the same handler procedure. Within the handler procedure, you can determine which menu item the user chose by calling **CommandInfo**(CMD_INFO_MENUITEM). Menu item IDs can also be used by other statements, such as the **Alter Menu Item statement**. If a menu item has neither a *handler* nor a *menuname* associated with it, that menu item is inert. Inert menu items are used for cosmetic purposes, such as displaying horizontal lines which break up a menu.

Creating Hierarchical Menus

To include a hierarchical menu on the new menu, use the **As** sub-clause instead of the **Calling** sub-clause. The **As** sub-clause must specify the name of the existing menu which should be attached to the new menu. The following example creates a custom menu containing one conventional menu item and one hierarchical menu.

```
Create Menu "Special" As
   "Configure" Calling config_sub_proc,
   "Objects" As "Objects"
```

When you add a hierarchical menu to the menu, the name of the hierarchical menu appears on the parent menu instead of the *menuitem* string.

Properties of a Menu Item

Menu items can be enabled or disabled; disabled items appear grayed out. Some menu items are checkable, meaning that the menu can display a check mark next to the item. At any given time, a checkable menu item is either checked or unchecked.

To set the properties of a menu item, include control codes (from the table below) at the start of the menuitem parameter.

Control code	Effect
(	The menu item is initially disabled. Example: (Close
(-	The menu item is a horizontal separator line; such a menu item cannot have a handler. Example: (-
($	This special code represents the File menu's most-recently-used (MRU) list. It may only appear once in the menu system, and it may not be used on a shortcut menu. To eliminate the MRU list from the File menu, either delete this code from MAPINFOW.MNU or re-create the File menu by issuing a **Create Menu** statement.
(>	This special code represents the Window menu's list of open windows. It may only appear once in the menu system.
!	Menu item is checkable, but it is initially unchecked. Example: !Confirm Deletion

Control code	Effect
! … ^ …	If a caret (^) appears within the text string of a checkable menu item, the item toggles between alternate text (for example, Show… vs. Hide…) instead of toggling between checked and unchecked. The text before the caret appears when the item is "checked." Example: !Hide Status Bar^Show Status Bar
!+	Menu item is checkable, and it is initially checked. Example: !+Confirm Deletions

Defining Keyboard Shortcuts

Menu items can have two different types of keyboard shortcuts, which let the user choose menu items through the keyboard rather than by clicking with the mouse.

One type of keyboard shortcut lets the user drop down a menu or choose a menu item by pressing keys. For example, on MapInfo Professional, the user can press Alt-W to show the Window menu, then press M (or Alt-M) to choose **New Map Window**. To create this type of keyboard shortcut, include the ampersand character (&) in the newmenuname or menuitem string (for example, specify "&Map" as the *menuitem* parameter in the **Create Menu** statement). Place the ampersand immediately before the character to be used as the shortcut.

The other type of keyboard shortcut allows the user to activate an option without going through the menu at all. If a menu item has a shortcut key sequence of Alt-F5, the user can activate the menu item by pressing Alt-F5. To create this type of shortcut, use the following key sequences.

ⓘ The codes in the following tables must appear at the end of a menu item name.

Windows Accelerator Code	Effect
/W {*letter* | %*number*}	Defines a Windows shortcut key which can be activated by pressing the appropriate key. Examples: Zap /WZ or Zap /W%120
/W# {*letter* | %*number*}	Defines a Windows shortcut key which also requires the shift key. Examples: Zap /W#Z or Zap /W#%120
/W@ {*letter* | %*number*}	Defines a Windows shortcut key which also requires the Alt key. Examples: Zap /W@Z or Zap /W@%120
/W^ {*letter* | %*number*}	Defines a Windows shortcut key which also requires the Ctrl key. Examples: Zap /W^Z or Zap /W^%120

To specify a function key as a Windows accelerator, the accelerator code must include a percent sign (%) followed by a number. The number 112 corresponds to F1, 113 corresponds to F2, etc.

The **Create Menu Bar As Default** statement removes and un-defines all custom menus created through the **Create Menu** statement. Alternately, if you need to un-define one, but not all, of the custom menus that your application has added, you can issue a statement of the form **Create Menu** *menuname* **As Default**.

After altering a standard MapInfo Professional menu (for example, “File”), you can restore the menu to its original state by issuing a **Create Menu** *menuname* **As Default** statement.

Calling Clause Options

The **Calling** clause specifies what should happen when the user chooses the custom menu command. The following table describes the available syntax.

Calling clause example	Description
`Calling M_FILE_NEW`	If **Calling** is followed by a numeric code from MENU.DEF, MapInfo Professional handles the event by running a standard MapInfo Professional menu command (the **File > New** command, in this example).
`Calling` *`my_procedure`*	If you specify a procedure name, MapInfo Professional handles the event by calling the procedure.
`Calling OLE "`*`methodname`*`"`	MapInfo Professional handles the event by making a method call to the OLE Automation object set by the SetCallback method.
`Calling DDE "server","topic"`	Windows only. MapInfo Professional handles the event by connecting through DDE to “*server*|*topic*” and sending an Execute message to the DDE server.

In the last two cases, the string sent to OLE or DDE starts with the three letters “MI:” (so that the server can detect that the message came from MapInfo Professional). The remainder of the string contains a comma-separated list of the values returned from relevant **CommandInfo( ) function** calls. For complete details on the string syntax, see the *MapBasic User Guide*.

Examples

The following example uses the **Create Menu** statement to create a custom menu, then adds the custom menu to MapInfo Professional's menu bar. This example removes the Window menu (ID 6) and the Help menu (ID 7), and then adds the custom menu, the Window menu, and the Help menu back to the menu bar. This technique guarantees that the last two menus will always be Window, and Help.

```
Declare Sub Main
Declare Sub addsub
```

```
Declare Sub editsub
Declare Sub delsub
Sub Main
   Create Menu "DataEntry" As
      "Add" Calling addsub,
      "Edit" Calling editsub,
      "Delete" Calling delsub

   Alter Menu Bar Remove ID 6, ID 7
   Alter Menu Bar Add "DataEntry", ID 6, ID 7
End Sub
```

The following example creates an abbreviated version of the File menu. The "(" control character specifies that the Close, Save, and Print options will be disabled initially. The Open and Save options have Windows accelerator key sequences (Ctrl+O and Ctrl+S, respectively). Note that both the Open and Save options use the **Chr$(9) function** to insert a Tab character into the menu item name, so that the remaining text is shifted to the right.

```
Include "MENU.DEF"

Create Menu "File" As
   "New" Calling M_FILE_NEW,
   "Open" +Chr$(9)+"Ctrl+O/W^O" Calling M_FILE_OPEN,
   "(-",
   "(Close" Calling M_FILE_CLOSE,
   "(Save" +Chr$(9)+"Ctrl+S /W^S" Calling M_FILE_SAVE,
   "(-",
   "(Print" Calling M_FILE_PRINT,
   "(-",
   "Exit" Calling M_FILE_EXIT
```

If you want to prevent the user from having access to MapInfo Professional's shortcut menus, use a **Create Menu** statement to re-create the appropriate menu, and define the menu as just a separator control code: "(-". The following example uses this technique to disable the Map window's shortcut menu.

```
Create Menu "MapperShortcut" As "(-"
```

See Also:

Alter Menu Item statement, **Create Menu Bar statement**

Create Menu Bar statement

Purpose

Rebuilds the entire menu bar, using the available menus. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax 1

```
Create Menu Bar As
   { menu_name | ID menu_number }
   [ , { menu_name | ID menu_number } ... ]
```

Syntax 2

```
Create Menu Bar As Default
```

menu_name is the name of a standard MapInfo Professional menu, or the name of a custom menu created through a **Create Menu statement**.

menu_number is the number associated with a standard MapInfo Professional menu (for example, 1 for the File menu).

Description

A **Create Menu Bar** statement tells MapInfo Professional which menus should appear on the menu bar, and in what order. If the statement omits one or more of the standard menu names, the resultant menu may be shorter than the standard MapInfo Professional menu. Conversely, if the statement includes the names of one or more custom menus (which were created through the **Create Menu statement**), the **Create Menu Bar** statement can create a menu bar that is longer than the standard MapInfo Professional menu.

Any menu can be identified by its name (for example, “File”), regardless of whether it is a standard menu or a custom menu. Each of MapInfo Professional's standard menus can also be referred to by its menu ID; for example, the File menu has an ID of 1.

See **Alter Menu Item statement** for a listing of the names and ID numbers of MapInfo Professional's menus.

After the menu bar has been customized, the following statement:

```
Create Menu Bar As Default
```

restores the standard MapInfo Professional menu bar. Note that the **Create Menu Bar As Default** statement removes any custom menu items that may have been added by other MapBasic applications that may be running at the same time. For the sake of not accidentally disabling other MapBasic applications, you should exercise caution when using the **Create Menu Bar As Default** statement.

Examples

The following example shortens the menu bar so that it includes only the File, Edit, Query, and window-specific (for example, Map, Browse, etc.) menus.

```
Create Menu Bar As
   "File", "Edit", "Query", "WinSpecific"
```

Ordinarily, the MapInfo Professional menu bar only displays a Map menu when a Map window is the active window. Similarly, MapInfo Professional only displays a Browse menu when a Browse window is the active window. The following example redefines the menu bar so that it always includes both the Map and Browse menus, even when no windows are on the screen. However, all

items on the Map menu will be disabled (grayed out) whenever the current window is not a Map window, and all items on the Browse menu will be disabled whenever the current window is not a Browse window.

```
Create Menu Bar As
   "File", "Edit", "Query", "Map", "Browse"
```

The following example creates a custom menu, called DataEntry, and then redefines the menu bar so that it includes only the File, Edit, and DataEntry menus.

```
Declare Sub AddSub
Declare Sub EditSub
Declare Sub DelSub

Create Menu "DataEntry" As
   "Add" calling AddSub,
   "Edit" calling EditSub,
   "Delete" calling DelSub

Create Menu Bar As
   "File", "Edit", "DataEntry"
```

See Also:

Alter Menu Bar statement, **Create Menu statement**, **Menu Bar statement**

Create MultiPoint statement

Purpose

Combines a number of points into a single object. All points have the same symbol. The Multipoint object displays in the Browser as a single record. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Create Multipoint
   [ Into { Window window_id | Variable var_name } ]
   [ num_points ]
   ( x1, y1 ) ( x2, y2 ) [ ... ]
   [ Symbol... ]
```

window_id is a window identifier.

var_name is the name of an existing object variable.

num_points is the number of points inside Multipoint object.

x y specifies the location of the point.

The **Symbol** clause specifies a symbol style.

ⓘ One symbol is used for all points contained in a Multipoint object.

Currently MapInfo Professional uses the following four different syntaxes to define a symbol used for points:

Syntax 2 (MapInfo Professional's 3.0 Symbol Syntax)

Symbol (*shape*, *color*, *size*)

shape is an integer, 31 or larger, specifying which character to use from MapInfo Professional's standard symbol set. MapInfo 3.0 symbols refers to the symbol set that was originally published with MapInfo for Windows 3.0 and has been maintained in subsequent versions of MapInfo Professional. To create an invisible symbol, use 31. The standard set of symbols includes symbols 31 through 67, but the user can customize the symbol set by using the Symbol application.

color is an integer RGB color value; see **RGB( ) function**.

size is an integer point size, from 1 to 48.

Syntax 3 (TrueType Font Syntax)

Symbol (*shape*, *color*, *size*, *fontname*, *fontstyle*, *rotation*)

shape is an integer, 31 or larger, specifying which character to use from a TrueType font. To create an invisible symbol, use 31.

color is an integer RGB color value; see RGB( ) function.

size is an integer point size, from 1 to 48.

fontname is a string representing a TrueType font name (for example, “Wingdings”).

fontstyle is an integer code controlling attributes such as bold.

rotation is a floating-point number representing a rotation angle, in degrees.

Syntax 4 (Custom Bitmap File Syntax)

Symbol (*filename*, *color*, *size*, *customstyle*)

filename is a string up to 31 characters long, representing the name of a bitmap file. The file must be in the CUSTSYMB directory (unless a **Reload Symbols statement** has been used to specify a different directory).

color is an integer RGB color value; see **RGB( ) function**.

size is an integer point size, from 1 to 48.

customstyle is an integer code controlling color and background attributes. See table below.

Syntax 5

Symbol *symbol_expr*

symbol_expr is a Symbol expression, which can either be the name of a Symbol variable, or a function call that returns a Symbol value, for example, the **MakeSymbol( ) function**.

Example

```
Create Multipoint 7 (0,0) (1,1) (2,2) (3,4) (-1,1) (3,-2) (4,3)
```

Create Object statement

Purpose

Creates one or more regions by performing a Buffer, Merge, Intersect, Union, Voronoi, or Isogram operation. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Create Object As { Buffer | Isogram | Union | Intersect | Merge |
ConvexHull | Voronoi }
   From fromtable
   [ Into { Table intotable | Variable varname } ]
   [ Data column = expression [ , column = expression... ] ]
   [ Group By { column | RowID } ]
```

fromtable is the name of an open table, containing one or more graphic objects.

intotable is the name of an open table where the new object(s) will be stored.

varname is the name of an Object variable where a new object will be stored.

column is the name of a column in the table.

expression is an expression used to populate *column*.

Description

The **Create Object** statement creates one or more new region objects, by performing a geographic operation (Buffer, Merge, Intersect, Union, ConvexHull, Voronoi, or Isogram) on one or more existing objects.

The **Into** clause specifies where results are stored. To store the results in a table, specify **Into Table**. To store the results in an Object variable, specify **Into Variable**. If you omit the **Into** clause, results are stored in the source table.

If you specify a **Group By** clause to perform data aggregation, you must store the results in a table rather than a variable.

The keyword which follows the **As** keyword dictates what type of objects are created. **Buffer** and **Isogram** are discussed in sections: **Create Object As Buffer** and **Create Object As Isogram**.

Union

Specify **Union** to perform a combine operation, which eliminates any areas of overlap. If you perform the union operation on two overlapping regions (each of which contains one polygon), the end result may be a region object that contains one polygon.

The union and merge operations are similar, but they behave very differently in cases where objects are completely contained within other objects. In this case, the merge operation removes the area of the smaller object from the larger object, leaving a hole where the smaller object was. The union operation does not remove the area of the smaller object.

Create Objects As Union is similar to the **Objects Combine statement**. The **Objects Combine statement** deletes the input and inserts a new combined object. **Create Objects As Union** only inserts the new combined object, it does not delete the input objects. Combining using a Target and potentially different tables is only available with the **Objects Combine statement**. The Combine Objects using Column functionality is only available using **Create Objects As Union** using the **Group By** clause.

If a **Create Object As Union** statement does not include a **Group By** clause, MapInfo Professional creates one combined object for all objects in the table. If the statement includes a **Group By** clause, it must name a column in the table to allow MapInfo Professional to group the source objects according to the contents of the column and produce a combined object for each group of objects.

If you specify a **Group By** clause, MapInfo Professional groups all records sharing the same value, and performs an operation (for example, Merge) on the group.

If you specify a **Data** clause, MapInfo Professional performs data aggregation. For example, if you perform merge or union operations, you may want to use the **Data** clause to assign data values based on the Sum( ) or Avg( ) aggregate functions.

Intersect

Specify **Intersect** to create an object representing the intersection of other objects (for example, if two regions overlap, the intersection is the area covered by both objects).

Merge

Specify **Merge** to create an object representing the combined area of the source objects. The Merge operation produces a results object that contains all of the polygons that belonged to the original objects. If the original objects overlap, the merge operation does not eliminate the overlap. Thus, if you merge two overlapping regions (each of which contains one polygon), the end result may be a region object that contains two overlapping polygons. In general, **Union** should be used instead.

Convex Hull

The **ConvexHull** operator creates a polygon representing a convex hull around a set of points. The convex hull polygon can be thought of as an operator that places a rubber band around all of the points. It consists of the minimal set of points such that all other points lie on or inside the polygon. The polygon is convex—no interior angle can be greater than 180 degrees.

The points used to construct the convex hull are any nodes from Regions, Polylines, or Points in the *fromtable*. If a **Create Object As ConvexHull** statement does not include a **Group By** clause, MapInfo Professional creates one convex hull polygon. If the statement includes a **Group By** clause

that names a column in the table, MapInfo Professional groups the source objects according to the contents of the column, then creates one convex hull polygon for each group of objects. If the statement includes a **Group By RowID** clause, MapInfo Professional creates one convex hull polygon for each object in the source table.

Voronoi

Specify **Voronoi** to create regions that represent the Voronoi solutions of the input points. The data values from the original input points can be assigned to the resultant polygon for that point by specifying data clauses.

Example

The following example merges region objects from the Parcels table, and stores the resultant regions in the table Zones. Since the **Create Object** statement includes a **Group By** clause, MapBasic groups the Parcel regions, then performs one merge operation for each group. Thus, the Zones table ends up with one region object for each group of objects in the Parcels table. Each group consists of all parcels having the same value in the zone_id column.

Following the **Create Object** statement, the parcelcount column in the Zones table indicates how many parcels were merged to produce that zone. The zonevalue column in the Zones table indicates the sum of the values from the parcels that comprised that zone.

```
Open Table "PARCELS"
Open Table "ZONES"
Create Object As Merge
   From PARCELS Into Table ZONES Data
   parcelcount=Count(*),zonevalue=Sum(parcelvalue)
   Group By zone_id
```

The next example shows a multi-object convex hull using the **Create Object As** statement.

```
Create Object As ConvexHull from state_caps into Table dump_table
```

Create Object As Buffer

Syntax

```
Create Object As Buffer
   From fromtable
   [ Into { Table intotable | Variable varname } ]
   [ Width bufferwidth [ Units unitname ]]]
   [ Type { Spherical | Cartesian } ] ]
   [ Resolution smoothness ]
   [ Data column = expression [ , column = expression... ] ]
   [ Group By { column | RowID } ]
```

bufferwidth is a number indicating the displacement used in a Buffer operation; if this number is negative, and if the source object is a closed object, the resulting buffer is smaller than the source object. If the width is negative, and the object is a linear object (line, polyline, arc) or a point, then the absolute value of width is used to produce a positive buffer.

unitname

smoothness is an integer from 2 to 100, indicating the number of segments per circle in a Buffer operation.

Description

If the **Create Object** statement performs a Buffer operation, the statement can include **Width** and **Resolution** clauses. The **Width** clause specifies the width of the buffer. The optional **Units** sub-clause lets you specify a distance unit name (such as “km” for kilometers) to apply to the **Width** clause. If the **Width** clause does not include the **Units** sub-clause, the buffer width is interpreted in MapBasic's current distance unit. By default, MapBasic uses miles as the distance unit; to change this unit, use the **Set Distance Units statement**.

Type is the method used to calculate the buffer width around the object. It can either be **Spherical** or **Cartesian**. Note that if the coordinate system of the *intotable* is NonEarth, then the calculations are performed using Cartesian methods regardless of the option chosen, and if the coordinate system of the *intotable* is Latitude/Longitude, then calculations are performed using Spherical methods regardless of the option chosen.

The optional **Type** sub-clause lets you specify the type of distance calculation used to create the buffer. If the **Spherical** type is used, then the calculation is done by mapping the data into a Latitude/Longitude On Earth projection and using widths measured using Spherical distance calculations. If the **Cartesian** type is used, then the calculation is done by considering the data to be projected to a flat surface and widths are measured using Cartesian distance calculations. If the **Width** clause does not include the **Type** sub-clause, then the default distance calculation type **Spherical** is used. If the data is in a Latitude/Longitude projection, then Spherical calculations are used regardless of the **Type** setting. If the data is in a NonEarth projection, the Cartesian calculations are used regardless of the **Type** setting.

The **Resolution** keyword lets you specify the number of segments comprising each circle of the buffer region. By default, a buffer object has a *smoothness* value of twelve (12), meaning that there are twelve segments in a simple ring-shaped buffer region. By specifying a larger *smoothness* value, you can produce smoother buffer regions. Note, however, that the larger the smoothness value, the longer the **Create Object** statement takes, and the more disk space the resultant object occupies.

If a **Create Object As Buffer** statement does not include a **Group By** clause, MapInfo Professional creates one buffer region. If the statement includes a **Group By** clause which names a column in the table, MapInfo Professional groups the source objects according to the contents of the column, then creates one buffer region for each group of objects. If the statement includes a **Group By RowID** clause, MapInfo Professional creates one buffer region for each object in the source table.

Example

The next example creates a region object, representing a quarter-mile buffer around whatever objects are currently selected. The buffer object is stored in the Object variable, corridor. A subsequent **Update statement** or **Insert statement** could then copy the object to a table.

```
Dim corridor As Object
Create Object As Buffer
   From Selection
   Into Variable corridor
   Width 0.25 Units "mi"
   Resolution 60
```

Create Object As Isogram

Syntax

```
Create Object As Isogram
   From fromtable
   [ Into { Table intotable } ]
   [ Data column = expression [ , column = expression... ] ]
   Connection connection_handle
   [ Distance dist1 [[ Brush ... ] [ Pen ... ]]
      [, dist2 [ Brush ... ] [ Pen ... ]]
      [, distN [ Brush ... ] [ Pen ... ] [,...]
      Units dist_unit ]
   [ Time time1 [[ Brush ... ] [ Pen ... ]]
      [, time2 [ Brush ... ] [ Pen ... ]]
      [, timeN [ Brush ... ] [ Pen ... ]] [,...]
      Units time_unit ]
```

connection_handle is a number expression returned from the **Open Connection statement** referencing the connection to be used.

dist1, *dist2*, *distN* are numeric expressions representing distances for the Isograms expressed in *dist_unit*s.

Brush is a valid **Brush clause** to specify fill style.

Pen is a valid **Pen clause** to specify a line style.

dist_unit is a valid unit of distance (for example, “km” for kilometers). See **Set Distance Units statement** for a complete list of possible values.

time1, *time2*, *timeN* are numeric values representing times for Isograms expressed in *time_unit*s.

time_unit is a string representing valid unit of time. Valid choices are: “hr”, “min”, or “sec”.

Description

If the **Create Object** statement performs an Isogram operation, you must pass a *connection_handle* that corresponds to an open connection created with an **Open Connection statement**. You must specify a **Distance** clause or a **Time** clause to create the size of the Isogram desired. The **Distance** clause can contain one or more distance expressions with an optional brush and/or pen for each one. If you do not specify a **Brush clause** or **Pen clause** the current brush and pen is used. No matter how many Distance instances you specify a single **Units** string must be provided to indicate the units in which the distances are expressed.

By specifying a **Time** clause, you can create regions based on time, with each one having an optional **Brush clause** and/or **Pen clause**. If you do not specify a **Brush clause** or **Pen clause** the current brush and pen is used. No matter how many Time instances you specify a single **Units** string must be provided to indicate the units in which the times are expressed. The maximum amount of values allowed is 50. Each value creates a separate band that can be either specific times or specific distances. Larger values take substantially longer to create. Many items factor into the equation, but in general, using the **Set Connection Isogram statement** with **MajorRoadsOnly** specified, results in a much quicker response compared to using the entire road network. MapBasic

only allows distances of 35 miles with **MajorRoadsOnly Off** and 280 miles with **MajorRoadsOnly On**. similarly, the maximum time is 0.5 hours with **MajorRoadsOnly Off** and 4 hours with **MajorRoadsOnly On**.

See Also:

Buffer( ) function, **ConvexHull( ) function**, **Objects Combine statement**, **Objects Erase statement**, **Objects Intersect statement**, **Open Connection statement**

Create Pline statement

Purpose

Creates a polyline object. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Create Pline
   [ Into { Window window_id | Variable var_name } ]
   [ Multiple num_sections ]
   num_points ( x1, y1 ) ( x2, y2 ) [ ... ]
   [ Pen... ]
   [ Smooth ]
```

window_id is a window identifier.

var_name is the name of an existing object variable.

num_points specifies how many nodes the polyline will contain.

num_sections specifies how many sections the multi-section polyline will contain.

each *x*, *y* pair defines a node of the polyline.

The **Pen** clause specifies a line style.

Description

The **Create Pline** statement creates a polyline object. If you need to create a polyline object, but do not know until run-time how many nodes the object should contain, create the object in two steps: First, use **Create Pline** to create an object with no nodes, and then use the **Alter Object statement** to add detail to the polyline object.

If the statement includes the optional **Into Variable** clause, the object will be stored in the specified object variable. If the **Into** clause specifies a **Window** identifier, the object will be stored in the appropriate place in the window (for example, in the editable layer of a Map window). If you omit the **Into** clause, MapInfo Professional attempts to store the object in the topmost window; if objects cannot be stored in the topmost window; no object is created.

The *x* and *y* parameters use whatever coordinate system MapBasic is currently using (Longitude/Latitude by default). Objects created on a Layout window, however, are specified in paper units. By default, MapBasic uses inches as the paper unit. To use a different paper unit, use the **Set Layout statement**. If you need to create objects on a Layout window, you must first issue a **Set CoordSys Layout** statement.

The optional **Pen clause** specifies a line style. If no Pen clause is specified, the **Create Pline** statement will use the current line style (the style which appears in the MapInfo Professional **Options > Line Style** dialog box). **Smooth** will smooth the line so that it appears to be one continuous line with curves instead of angles.

A single-section polyline can contain up to 32,763 nodes. For a multiple-section polyline, the limit is smaller: for each additional section, reduce the number of nodes by three.

See Also:

Alter Object statement, **Insert statement**, **Pen clause**, **Set CoordSys statement**, **Update statement**

CreatePoint( ) function

Purpose

Returns an Object value representing a point. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

CreatePoint(*x*, *y*)

x is a float value, representing an x-position (for example, Longitude).

y is a float value, representing a y-position (for example, Latitude).

Return Value

Object

Description

The **CreatePoint( )** function returns an Object value representing a point.

The *x* and *y* parameters should use whatever coordinate system MapBasic is currently using. By default, MapBasic uses a Longitude/Latitude coordinate system, although the **Set CoordSys statement** can re-configure MapBasic to use a different coordinate system. Note that MapBasic's coordinate system is independent of the coordinate system of any Map window.

The point object will use whatever Symbol style is currently selected. To create a point object with a specific Symbol style, you could issue the **Set Style statement** before calling CreatePoint( ). Alternately, instead of calling **CreatePoint( )**, you could issue a **Create Point statement**, which has an optional **Symbol** clause.

The point object created through the **CreatePoint( )** function could be assigned to an Object variable, stored in an existing row of a table (through the **Update statement**), or inserted into a new row of a table (through an **Insert statement**).

If you need to create objects on a Layout window, you must first issue a **Set CoordSys statement**.

Examples

The following example uses the **Insert statement** to insert a new row into the table Sites. The **CreatePoint( )** function is used within the body of the Insert statement to specify the graphic object that will be attached to the new row.

```
Open Table "sites"
Insert Into sites (obj)
   Values ( CreatePoint(-72.5, 42.4) )
```

The following example assumes that the table Sites has Xcoord and Ycoord columns, which indicate the longitude and latitude positions of the data. The **Update statement** uses the **CreatePoint( )** function to build a point object for each row in the table. Following the Update operation, each row in the Sites table will have a point object attached. Each point object will be located at the position indicated by the Xcoord, Ycoord columns.

```
Open Table "sites"
Update sites
   Set obj = CreatePoint(xcoord, ycoord)
```

The above example assumes that the Xcoord, Ycoord columns contain actual longitude and latitude degree values.

See Also:

Create Point statement, **Insert statement**, **Update statement**

Create Point statement

Purpose

Creates a point object. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Create Point
   [ Into { Window window_id | Variable var_name } ]
   ( x, y )
   [ Symbol... ]
```

window_id is a window identifier.

var_name is the name of an existing object variable.

x, *y* specifies the location of the point.

The **Symbol** clause specifies a symbol style.

Description

The **Create Point** statement creates a point object.

If the statement includes the optional Into **Variable** clause, the object will be stored in the specified object variable. If the **Into** clause specifies a window identifier, the object will be stored in the appropriate place in the window (for example, in the editable layer of a Map window). If the **Into** clause is not provided, MapBasic will attempt to store the object in the topmost window; if objects may not be stored in the topmost window (for example, if the topmost window is a grapher) no object will be created.

The *x* and *y* parameters use whatever coordinate system MapBasic is currently using. By default, MapBasic uses a longitude, latitude coordinate system, although the **Set CoordSys statement** can re-configure MapBasic to use a different coordinate system. Note that MapBasic's coordinate system is independent of the coordinate system of any Map window. Objects created on a Layout window, however, are specified in paper units: each x-coordinate represents a distance from the left edge of the page, while each y-coordinate represents the distance from the top edge of the page. By default, MapBasic uses inches as the default paper unit. To use a different paper unit, use the **Set Paper Units statement**.

If you need to create objects on a Layout window, you must first issue a **Set CoordSys Layout statement**.

The optional **Symbol clause** specifies a symbol style; see **Symbol clause** for more details. If no **Symbol clause** is specified, the **Create Point** statement uses the current symbol style (the style which appears in the **Options > Symbol Style** dialog box).

See Also:

CreatePoint( ) function, **Insert statement**, **Symbol clause**, **Update statement**

Create PrismMap statement

Purpose

Creates a Prism map. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Create PrismMap
   [ From Window window_ID | MapString mapper_creation_string ]
      { layer_id | layer_name }
   With expr
   [ Camera [Pitch angle | Roll angle | Yaw angle | Elevation angle] |
      [ Position ( x, y, z ) | FocalPoint ( x, y, z ) ] |
      [ Orientation(vu_1, vu_2, vu_3, vpn_1, vpn_2, vpn_3,
```

```
   clip_near, clip_far) ] ]
[ Light Color lightcolor ] ]
[ Scale grid_scale ]
[ Background backgroundcolor ]
```

window_id is a window identifier a for a Map window which contains a region layer. An error message is displayed if a layer with regions is not found.

mapper_creation_string specifies a command string that creates the mapper textured on the Prism map.

layer_id is the layer identifier of a layer in the map (one or larger).

layer_name is the name of a layer in the map.

expr is an expression that is evaluated for each row in the table.

Camera specifies the camera position and orientation.

angle is an angle measurement in degrees. The horizontal angle in the dialog box ranges from 0-360 degrees and rotates the maps around the center point of the grid. The vertical angle in the dialog box ranges from 0-90 and measures the rotation in elevation from the start point directly over the map.

Pitch adjusts the camera's current rotation about the x-axis centered at the camera's origin.

Roll adjusts the camera's current rotation about the z-axis centered at the camera's origin.

Yaw adjusts the camera's current rotation about the y-axis centered at the camera's origin.

Elevation adjusts the current camera's rotation about the x-axis centered at the camera's focal point.

Position indicates the camera and/or light position.

FocalPoint indicates the camera and/or light focal point.

Orientation specifies the camera's ViewUp (*vu_1*, *vu_2*, *vu_3*), ViewPlane Normal (*vpn_1*, *vpn_2*, *vpn_3*) and Clipping Range (*clip_near* and *clip_far*), used specifically for persistence of view).

grid_scale is the amount to scale the grid in the z direction. A value >1 will exaggerate the topology in the z direction, a value <1 will scale down the topological features in the z direction.

backgroundcolor is a color to be used to set the background and is specified using the **RGB( ) function**.

Description

The **Create PrismMap** statement creates a Prism Map window. The Prism Map is a way to associate multiple variables for a single object in one visual. For example, the color associated with a region may be the result of thematic shading while the height the object is extruded through may represent a different value. The **Create PrismMap** statement corresponds to MapInfo Professional's **Map > Create Prism Map** menu item.

Between sessions, MapInfo Professional preserves Prism Maps settings by storing a **Create PrismMap** statement in the workspace file. Thus, to see an example of the **Create PrismMap** statement, you could create a map, choose the **Map > Create Thematic Map** command, save the

workspace (for example, PRISM.WOR), and examine the workspace in a MapBasic text edit window. You could then copy the **Create PrismMap** statement in your MapBasic program. Similarly, you can see examples of the **Create PrismMap** statement by opening the MapBasic Window before you choose **Map > Create Thematic Map**.

Each **Create PrismMap** statement must specify an *expr* expression clause. MapInfo Professional evaluates this expression for each object in the layer; following the **Create PrismMap** statement, MapInfo Professional chooses each object's display style based on that record's *expr* value. The expression typically includes the names of one or more columns from the table being shaded.

The optional *window_id* clause identifies which map layer to use in the prism map; if no *window_id* is provided, MapBasic uses the topmost Map window. The **Create PrismMap** statement must specify which layer to use, even if the Map window has only one layer. The layer may be identified by number (*layer_id*), where the topmost map layer has a *layer_id* value of one, the next layer has a *layer_id* value of two, etc. Alternately, the **Create PrismMap** statement can identify the map layer by name (for example, “world”).

Example

```
Open Table "STATES.TAB" Interactive
Map From STATES
Create PrismMap From Window FrontWindow( ) STATES With Pop_1980 Background
RGB(192,192,192)
```

See Also:

Set PrismMap statement, **PrismMapInfo( ) function**

Create Ranges statement

Purpose

Calculates thematic ranges and stores the ranges in an array, which can then be used in a **Shade statement**. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Create Ranges
   From table
   With expr
   [ Use {"Equal Ranges" | "Equal Count" | "Natural Break" | "StdDev" } ]
   [ Quantile Using q_expr ]
   [ Number num_ranges ]
   [ Round rounding_factor ]
   Into Variable array_variable
```

table is the name of the table to be shaded thematically.

expr is an expression that is evaluated for each row in the table.

q_expr is the expression used to perform quantiling.

num_ranges specifies the number of ranges (default is 4).

rounding_factor is factor by which the range break numbers should be rounded (for example, 10 to round off values to the nearest ten).

array_variable is the float array variable in which the range information will be stored.

Description

The **Create Ranges** statement calculates a set of range values which can then be used in a **Shade statement** (which creates a thematic map layer). For an introduction to thematic maps, see the MapInfo Professional documentation.

The optional **Use** clause specifies how to break the data into ranges. If you specify “Equal Ranges” each range covers an equal portion of the spectrum of values (for example, 0-25, 25-50, 50-75, 75-100). If you specify “Equal Count” the ranges are constructed so that there are approximately the same number of rows in each range. If you specify “Natural Break” the ranges are dictated by natural breaks in the set of data values. If you specify “StdDev” the middle range breaks at the mean of your data values, and the ranges above and below the middle range are one standard deviation above or below the mean. MapInfo Professional uses the population standard deviation (N - 1).

The **Into Variable** clause specifies the name of the float array variable that will hold the range information. You do not need to pre-size the array; MapInfo Professional automatically enlarges the array, if necessary, to make room for the range information. The final size of the array is twice the number of ranges, because MapInfo Professional calculates a high value and a low value for each range.

After calling **Create Ranges**, call the **Shade statement** to create the thematic map, and use the Shade statement's optional **From Variable** clause to read the array of ranges. The Shade statement usually specifies the same table name and column expression as the **Create Ranges** statement.

Quantiled Ranges

If the optional **Quantile Using** clause is present, the **Use** clause is ignored and range limits are defined according to the *q_expr*.

Quantiled ranges are best illustrated by example. The following statement creates ranges of buying power index (BPI) values, and uses state population statistics to perform quantiling to set the range limits.

```
Create Ranges From states
   With BPI_1990 Quantile Using Pop_1990
   Number 5
   Into Variable f_ranges
```

Because of the `Number 5` clause, this example creates a set of five ranges.

Because of the `With BPI_1990` clause, states with the highest BPI values will be placed in the highest range (the deepest color), and states with the lowest BPI values will be placed in the lowest range (the palest color).

Because of the `Quantile Using Pop_1990` clause, the range limits for the intermediate ranges are calculated by quantiling, using a method that takes state population (`Pop_1990`) into account. Since the **Quantile Using** clause specifies the Pop_1990 column, MapInfo Professional calculates

the total 1990 population for the table (which, for the United States, is roughly 250 million). MapInfo Professional divides that total by the number of ranges (in this case, five ranges), producing a result of fifty million. MapInfo Professional then tries to define the ranges in such a way that the total population for each range approximates, but does not exceed, fifty million.

MapInfo Professional retrieves rows from the States table in order of BPI values, starting with the states having low BPI values. MapInfo Professional assigns rows to the first range until adding another row would cause the cumulative population to match or exceed fifty million. At that time, MapInfo Professional considers the first range “full” and then assigns rows to the second range. MapInfo Professional places rows in the second range until adding another row would cause the cumulative total to match or exceed 100 million; at that point, the second range is full, etc.

Example

```
Include "mapbasic.def"

Dim range_limits( ) As Float, brush_styles( ) As Brush
Dim col_name As Alias

Open Table "states" Interactive

Create Styles
   From Brush(2, CYAN, 0) 'style for LOW range
   To Brush (2, BLUE, 0) 'style for HIGH range
   Vary Color By "RGB"
   Number 5
   Into Variable brush_styles
' Store a column name in the Alias variable:
col_name = "Pop_1990"

Create Ranges From states
   With col_name
   Use "Natural Break"
   Number 5
   Into Variable range_limits

Map From states

Shade states
   With col_name
   Ranges
      From Variable range_limits
      Style Variable brush_styles

' Show the theme legend window:
Open Window Legend
```

See Also:

Create Styles statement, **Set Shade statement**, **Shade statement**

Create Rect statement

Purpose

Creates a rectangle or square object. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Create Rect
   [ Into { Window window_id | Variable var_name } ]
   ( x1, y1 ) ( x2, y2 )
   [ Pen ... ]
   [ Brush ... ]
```

window_id is a window identifier.

var_name is the name of an existing object variable.

x1, *y1* specifies the starting corner of the rectangle.

x2, *y2* specifies the opposite corner of the rectangle.

Pen is a valid **Pen clause** to specify a line style.

Brush is a valid **Brush clause** to specify fill style.

Description

If the **Create Rect** statement includes the optional **Into Variable** clause, the object will be stored in the specified object variable. If the **Into** clause specifies a **Window** identifier, the object will be stored in the appropriate place in the window (for example, in the editable layer of a Map window). If the **Into** clause is not provided, MapBasic will attempt to store the object in the topmost window; if objects may not be stored in the topmost window (for example, if the topmost window is a grapher) no object will be created.

The *x* and *y* parameters use whatever coordinate system MapBasic is currently using. By default, MapBasic uses a Longitude/Latitude coordinate system, although the **Set CoordSys statement** can re-configure MapBasic to use a different coordinate system. Note that MapBasic's coordinate system is independent of the coordinate system of any Map window. Objects created on a Layout window, however, are specified in paper units: each x-coordinate represents a distance from the left edge of the page, while each y-coordinate represents the distance from the top edge of the page. By default, MapBasic uses inches as the default paper unit. To use a different paper unit, call the **Set Paper Units statement**.

If you need to create objects on a Layout window, you must first issue a **Set CoordSys Layout statement**.

The optional **Pen clause** specifies a line style; see **Pen clause** for more details. If no **Pen clause** is specified, the **Create Rect** statement uses the current line style (the style which appears in the **Options > Line Style** dialog box). Similarly, the optional **Brush clause** specifies a fill style; see **Brush clause** for more details.

See Also:

Brush clause, **Create RoundRect statement**, **Insert statement**, **Pen clause**, **Update statement**

Create Redistricter statement

Purpose

Begins a redistricting session. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Create Redistricter source_table By district_column
   With
   [ Layer <layer_number> ]
   [ Count ]
   [ , Brush ] [ , Symbol ] [ , Pen ]
   [ , { Sum | Percent } ( expr ) ] [ , { Sum | Percent } ( expr ) ... ]
   [ Percentage From expr]
   [ Percentage from { column | row } ]
   [ Order { "MRU" | "Alpha" | "Unordered" } ]
```

source_table is the name of the table containing objects to be grouped into districts.

district_column is the name of a column; the initial set of districts is built from the original contents of this column, and as objects are assigned to different districts, MapInfo Professional stores the object's new district name in this column.

layer_number is the number of a layer in the current Map window (for example, 1 for the top layer); to determine the number of layers in a Map window, call the **MapperInfo( ) function**.

Count keyword specifies that the Districts Browser will show a count of the objects belonging to each district.

Brush keyword specifies that the Districts Browser will show each district's fill style.

Symbol keyword specifies that the Districts Browser will show each district's symbol style.

Pen keyword specifies that the Districts Browser will show each district's line style.

expr is a numeric column expression.

Percentage From clause specifies in-row calculation.

Order clause specifies the order of rows in the Districts Browser (alphabetical, unsorted, or based on most-recently-used); default is MRU.

Description

The **Create Redistricter** statement begins a redistricting session. This statement corresponds to choosing MapInfo Professional's **Window > New Redistrict Window** command. For an introduction to redistricting, see the MapInfo Professional documentation.

To control the set of districts, use the **Set Redistricter statement**. To end the redistricting session, use the **Close Window statement** to close the Districts Browser window.

If you include the **Brush** keyword, the Districts Browser includes a sample of each district's fill style. Note that this is not a complete **Brush clause**; the keyword **Brush** appears by itself. Similarly, the **Symbol** and **Pen** keywords are individual keywords, not a complete **Symbol clause** or **Pen clause**. If the Districts Browser includes brush, symbol, and/or pen styles, the user can change a district's style by clicking on the style sample that appears in the Districts Browser.

The **Percentage From** clause allows you to specify the in-row mode of percentage calculation. If the **Percentage From** clause is not specified, the in-column method of calculation is used.

See Also:

Set Redistricter statement

Create Region statement

Purpose

Creates a region object. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Create Region
   [ Into { Window window_id | Variable var_name } ]
     num_polygons
   [ num_points1 ( x1, y1 ) ( x2, y2 ) [ ... ] ]
   [ num_points2 ( x1, y1 ) ( x2, y2 ) [ ... ] ... ]
   [ Pen ... ]
   [ Brush ... ]
   [ Center ( center_x, center_y ) ]
```

window_id is a window identifier.

var_name is the name of an existing object variable.

num_polygons specifies the number of polygons that will make up the region (zero or more).

num_points1 specifies the number of nodes in the region's first polygon.

num_points2 specifies the number of nodes in the region's second polygon, etc.

Each *x*, *y* pair specifies one node of a polygon.

Pen is a valid **Pen clause** to specify a line style.

Brush is a valid **Brush clause** to specify fill style.

center_x is the x-coordinate of the object centroid.

center_y is the y-coordinate of the object centroid.

Description

The **Create Region** statement creates a region object.

The *num_polygons* parameter specifies the number of polygons which comprise the region object. If you specify a *num_polygons* parameter with a value of zero, the object will be created as an empty region (a region with no polygons). You can then use the **Alter Object statement** to add details to the region.

Depending on your application, you may need to create a region object in two steps, first using **Create Region** to create an object with no polygons, and then using the Alter Object statement to add details to the region object. If your application needs to create region objects, but it will not be known until run-time how many nodes or how many polygons the regions will contain, you must use the Alter Object statement to add the variable numbers of nodes. See **Alter Object statement** for more information.

If the statement includes the optional Into **Variable** clause, the object will be stored in the specified object variable. If the **Into** clause specifies a window identifier, the object will be stored in the appropriate place in the window (for example, in the editable layer of a Map window). If the **Into** clause is not provided, MapBasic will attempt to store the object in the topmost window; if objects may not be stored in the topmost window (for example, if the topmost window is a grapher) no object will be created.

The *x* and *y* parameters use whatever coordinate system MapBasic is currently using. By default, MapBasic uses a Longitude/Latitude coordinate system, although the **Set CoordSys statement** can re-configure MapBasic to use a different coordinate system. Note that MapBasic's coordinate system is independent of the coordinate system of any Map window. Objects created on a Layout window, however, are specified in paper units: each x-coordinate represents a distance from the left edge of the page, while each y-coordinate represents the distance from the top edge of the page. By default, MapBasic uses inches as the default paper unit. To use a different paper unit, To use a different paper unit, call the **Set Paper Units statement**.

If you need to create objects on a Layout window, you must first issue a **Set CoordSys Layout statement**.

The optional **Pen clause** specifies a line style used to draw the outline of the object; see **Pen clause** for more details. If no Pen clause is specified, the **Create Region** statement uses the current line style (the style which appears in the **Options > Line Style** dialog box). Similarly, the optional Brush clause specifies a fill style; see **Brush clause** for more details.

A single-polygon region can contain up to 1,048,572 nodes. For a multiple-polygon region, the limit is smaller: for each additional polygon, reduce the number of nodes by three. There can be a maximum of 32,000 polygons per region (multipolygon region).

Example

```
Dim obj_region As Object
Dim x(100), y(100) As Float
Dim i, node_count As Integer

' If you store a set of coordinates in the
' x( ) and y( ) arrays, the following statements
' will create a region object that has a node
' at each x,y location:

' First, create an empty region object
Create Region Into Variable obj_region 0

' Now add nodes to populate the object:
For i = 1 to node_count
   Alter Object obj_region Node Add ( x(i), y(i) )
Next

' Now store the object in the Sites table:
Insert Into Sites (Object) Values (obj_region)
```

See Also:

Alter Object statement, **Brush clause**, **Insert statement**, **Pen clause**, **Update statement**

Create Report From Table statement

Purpose

Creates a report file for Crystal Reports from an open MapInfo Professional table. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Create Report From Table *tablename* [**Into** *reportfilespec*][**Interactive**]

tablename is an open table in MapInfo Professional.

reportfilespec is a full path and filename for the new report file.

The **Interactive** keyword signifies that the new report should immediately be loaded into the Crystal Report Designer module. Interactive mode is implied if the **Into** clause is missing. You cannot create a report from a grid or raster table; you will get an error.

See Also:

Open Report statement

Create RoundRect statement

Purpose

Creates a rounded rectangle object. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Create RoundRect
   [ Into { Window window_id | Variable var_name } ]
   ( x1, y1 ) ( x2, y2 ) rounding
   [ Pen ... ]
   [ Brush ... ]
```

window_id is a window identifier.

var_name is the name of an existing object variable.

x1, y1 specifies one corner of the rounded rectangle.

x2, y2 specifies the opposite corner of the rectangle.

rounding is a float value, in coordinate units (for example, inches on a Layout or degrees on a Map), specifying the diameter of the circle which fills the rounded rectangle's corner.

Pen is a valid **Pen clause** to specify a line style.

Brush is a valid **Brush clause** to specify fill style.

Description

The **Create RoundRect** statement creates a rounded rectangle object (a rectangle with rounded corners).

The *x* and *y* parameters use whatever coordinate system MapBasic is currently using. By default, MapBasic uses a Longitude/Latitude coordinate system, although the **Set CoordSys statement** can re-configure MapBasic to use a different coordinate system. Note that MapBasic's coordinate system is independent of the coordinate system of any Map window. Objects created on a Layout window, however, are specified in paper units: each x-coordinate represents a distance from the left edge of the page, while each y-coordinate represents the distance from the top edge of the page. By default, MapBasic uses inches as the default paper unit. To use a different paper unit, call the **Set Paper Units statement**.

If you need to create objects on a Layout window, you must first issue a **Set CoordSys Layout statement**.

The optional **Pen clause** specifies a line style used to draw the outline of the object; see **Pen clause** for more details. If no Pen clause is specified, the **Create RoundRect statement** uses the current line style (the style which appears in the **Options > Line Style** dialog box). Similarly, the optional Brush clause specifies a fill style; see **Brush clause** for more details.

See Also:

Brush clause, **Create Rect statement**, **Insert statement**, **Pen clause**, **Update statement**

Create Styles statement

Purpose

Builds a set of Pen, Brush or Symbol styles, and stores the styles in an array. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Create Styles
   From { Pen ... | Brush ... | Symbol ... }
   To { Pen ... | Brush ... | Symbol ... }
   Vary { Color By { "RGB" | "HSV" } | Background By { "RGB" | "HSV" } |
      Size By { "Log" | "Sqrt" | "Constant" }}
   [ Number num_styles ]
   [ Inflect At range_number With { Pen ... | Brush ... | Symbol ...} ]
Into Variable array_variable
```

num_styles is the number of drawing styles (for example, the number of fill styles) to create. The default number is four.

range_number is a SmallInt range number; the inflection attribute is placed after this range.

array_variable is an array variable that will store the range of pens, brushes, or symbols.

Pen is a valid **Pen clause** to specify a line style.

Brush is a valid **Brush clause** to specify fill style.

Symbol is a valid **Symbol clause** to specify a point style.

Description

The **Create Styles** statement defines a set of Pen, Brush, or Symbol styles, and stores the styles in an array variable. The array can then be used in a **Shade statement** (which creates a thematic map layer). For an introduction to thematic mapping, see the MapInfo Professional documentation.

The **From** clause specifies a Pen, Brush, or Symbol style. If the array of styles is later used in a thematic map, the From style is the style assigned to the “low” range. The **To** clause specifies a style that corresponds to the “high” range of a thematic map.

The **Create Styles** statement builds a set of styles which are interpolated between the From style and the To style. For example, the From style could be a **Brush clause** representing a deep, saturated shade of blue, and the To style could be a Brush clause representing a pale, faint shade of blue. In this case, MapInfo Professional builds a set of Brush styles that vary from pale blue to saturated blue.

The optional **Number** clause specifies the total number of drawing styles needed; this number includes the two styles specified in the **To** and **From** clauses. Usually, this corresponds to the number of ranges specified in a subsequent **Shade statement**.

The **Vary** clause specifies how to spread an attribute among the styles. To spread the foreground color, use the **Color** sub-clause. To spread the background color, use the **Background** sub-clause. In either case, color can be spread by interpolating the RGB or HSV components of the From and To colors. If you are creating an array of Symbol styles, you can use the **Size** sub-clause to vary the symbols' point sizes. Similarly, if you are creating an array of Pen styles, you can use the **Size** sub-clause to vary line width.

The optional **Inflect At** clause specifies an inflection attribute that goes between the From and To styles. If you specify an **Inflect At** clause, MapInfo Professional creates two sets of styles: one set of styles interpolated between the From style and the Inflect style, and another set of styles interpolated between the Inflect style and the To style. For example, using an inflection style, you could create a thematic map of profits and losses, where map regions that have shown a profit appear in various shades of green, while regions that have shown a loss appear in various shades of red. Inflection only works when varying the color attribute.

The **Into Variable** clause specifies the name of the array variable that will hold the styles. You do not need to pre-size the array; MapInfo Professional automatically enlarges the array, if necessary, to make room for the set of styles. The *array_variable* (Pen, Brush, or Symbol) must match the style type specified in the **From** and **To** clauses.

Example

The following example demonstrates the syntax of the **Create Styles** statement.

```
Dim brush_styles( ) As Brush

Create Styles
   From Brush(2, CYAN, 0) 'style for LOW range
   To Brush (2, BLUE, 0) 'style for HIGH range
   Vary Color By "RGB"
   Number 5
   Into Variable brush_styles
```

This **Create Styles** statement defines a set of five Brush styles, and stores the styles in the b_ranges array. A subsequent **Shade statement** could create a thematic map which reads the Brush styles from the b_ranges array. For an example, see **Create Ranges statement**.

See Also:

Create Ranges statement, **Set Shade statement**, **Shade statement**

Create Table statement

Purpose

Creates a new table. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Create Table *table* [(**column** *columntype* [, ...]) | **Using** *from_table*]
   [**File** *filespec*]

```
[ {
   Type NATIVE |
   Type DBF [ CharSet char_set ] |
   Type { Access | ODBC } Database database_filespec [ Version version ]
      Table tablename [ Password pwd ] [ CharSet char_set ] |
   Type TILESERVER
      TileType { LevelRowColumn | QuadKey }
      URL url
      [ AttributionText "attributiontext" ] [ Font font_clause ]
      [ StartTileNum { 0 | 1 } ]
      [ Minlevel min_level ]
      MaxLevel max_level
      Height height [Width width]
      [ ReadTimeout read_time_out ]
      [ RequestTimeout request_time_out]
      CoordSys coordsys
} ]
[ Version version_pro ]
```

table is the name of the table as you want it to appear in MapInfo Professional.

column is the name of a column to create. Column names can be up to 31 characters long, and can contain letters, numbers, and the underscore (_) character. Column names cannot begin with numbers.

columntype is the data type associated with the column. Each columntype is defined as follows:

```
Char( width ) | Float | Integer | SmallInt |
   Decimal( width, decplaces ) | Date | Logical
```

from_table is the name of a currently open table in which the column you want to place in a new table is stored. The *from_table* must be a base table, and must contain column data. Query tables and raster tables cannot be used and will produce an error. The column structure of the new table will be identical to this table.

filespec specifies where to create the .TAB, .MAP, and .ID files (and in the case of Access, .AID files). If you omit the **File** clause, files are created in the current directory.

version_pro is 100 (to create a table that can be read by versions of MapInfo Professional), or 300 for (MapInfo Professional 3.0 format). Does not apply when creating an Access table; the version of the Access table is handled by DAO.

Type DBF:

char_set is the name of a character set; see **CharSet clause**.

Type { Access | ODBC }:

database_filespec is a string that identifies a valid Access database. If the specified database does not exist, MapInfo Professional creates a new Access (.MDB or .ACCDB) file.

version is an expression that specifies the version of the Microsoft Jet database format to be used by the new database. Acceptable values are 4.0 (for Access 2000) or 3.0 (for Access '95/'97). If omitted, the default version is 12.0. If the database in which the table is being created already exists, the specified database version is ignored.

tablename is a string that indicates the name of the table as it will appear in Access.

pwd is the database-level password for the database, to be specified when database security is turned on.

char_set is the name of a character set; see **CharSet clause**.

Type TILESERVER:

url is the fully qualified URL, either http://<server> or https://<server>, to request a tile from a tile server. If the URL does not have the following replaceable tags, then the Create Table fails:

- If *tile_type* is **QuadKey**, then the URL must contain {QUADKEY}.
- If *tile_type* is **LevelRowColumn**, then the URL must contain {LEVEL}, {ROW}, and {COL} tags that will be replaced at runtime. Servers support the {ROW} and {COL} tags differently; sometimes these tags may need to be reversed for row and column (or X, Y).

attributiontext is the attribution text that will display as text in the map window when displaying tiles from a tileserver. This text must be in quotes (`"..."`).

font_clause is optional and specifies the font style to use on the attribution text. This is a Font expression, for example, **`MakeFont`**`( fontname, style, size, fgcolor, bgcolor ).`

```
Font ("Verdana", 1, 24, 0, 255)
Font MakeFont("Verdana", 1, 24, 0, 255)
```

For more details, see **Font clause on page 298** or **MakeFont( ) function on page 392**.

min_level is the minimum level for a tile server. This must be either zero (0) or a positive value and less than the **max_level**. The default is zero (0).

max_level is the max level the tile server supports. This must be a positive value.

height is the height in pixels of a single tile from the tile server. This must be a positive value.

width is the width in pixels of a single tile from the tile server. If specified, this must be a positive value. If not specified, the height is used as the width.

read_time_out is the number in seconds until the read of tiles times out (the default is 300). This must be a positive value.

request_time_out is a value in seconds until the request of the tiles times out (the default is 100). This must be a positive value.

coordsys is the default coordinate system for the tile server. MapInfo Professional cannot reproject the tile server image, so it reprojects the map to use this coordinate system. You are unable to change the coordinate system for the map when it includes a tile server layer. The bounds of the coordinate system also specify the bounds of the first tile (the one tile in the minimum level). This is how MapInfo Professional knows how to calculate the extent of the tiles in the other levels.

Description

The **Create Table** statement creates a new empty table with up to 250 columns. Specify **ODBC** to create new tables on a DBMS server.

The **Using** clause allows you to create a new table as part of the “Combine Objects Using Column” functionality. The *from_table* must be a base table, and must contain column data. Query tables and raster tables cannot be used and will produce an error. The column structure of the new table being created will be identical to this table.

The optional **File** clause specifies where to create the new table. If no **File** clause is used, the table is created in the current directory or folder.

The optional **Type** clause specifies the table's data format. The default type is **NATIVE**, but can alternately be **DBF**. The NATIVE format takes up less disk space than the DBF format, but the DBF format produces base files that can be read in any dBASE-compatible database manager. Also, create new tables on DBMS Servers from the **ODBC Type** clause in the **Create Table** statement.

The **CharSet** clause specifies a character set. The *char_set* parameter should be a string constant, such as “WindowsLatin1”. If no **CharSet** clause is specified, MapBasic uses the default character set for the hardware platform that is in use at runtime. For more details, see **CharSet clause**.

The **SmallInt** column type reserves two bytes for each value; thus, the column can contain values from -32,767 to +32,767. The **Integer** column type reserves four bytes for each value; thus, the column can contain values from -2,147,483,647 to +2,147,483,647.

The **TileType** clause specifies the type of the tile server this table will use, **QuadKey** or **LevelRowColumn**. This represents the way that the tile server retrieves the tiles. You must set this based on what the server supports and uses:

QuadKey – A server that uses a quad tree algorithm splits the world up into squares that are 256 pixels by 256 pixels. Each tile is referred to by a unique string of characters between 0 – 3 (**QuadKey**), which describes the position and zoom level at which to place the tile.

LevelRowColumn – A server that splits the world up into squares where each tile identifier is a list containing the zoom level, row, and column number of the tile. The format of the tile identifier may vary from server to server, so the {ROW} and {COL} tags may seem reversed for some servers.

The **StartTileNum** clause is optional. It is the number of the starting tile, either zero (0) or one (1). Zero (0) is the default start tile number.

The **Version** clause controls the table's format. If you specify `Version 100`, MapInfo Professional creates a table in a format that can be read by versions of MapInfo Professional. If you specify `Version 300`, MapInfo Professional creates a table in the format used by MapInfo Professional 3.0. Note that region and polyline objects having more than 8,000 nodes and multiple-segment polyline objects require version 300. If you omit the **Version** clause, the table is created in the version 300 format.

Messages when creating a Tile Server Table

If an error occurs while fetching tiles from the server, which can happen when drawing a tile server layer in a map window, then check:

- The tile server URL is incorrect.
- The tile server is currently available.
- The amount of time the server takes to respond to the request does not exceed the specified timeout value.

- Improper authentication due to the tile server being on a secure server or is accessed through a proxy server.

If an error occurs loading a tile server table, then check if:

- A required property in the configuration file is missing.
- The configuration file is missing.

If an error occurs creating the tile server configuration file, which is an XML file, then check that the configuration file can be created in the path supplied.

Example

The following example shows how to create a table called Towns, containing 3 fields: a character field called townname, an integer field called population, and a decimal field called median_income. The file will be created in the subdirectory C:\MAPINFO\DATA. Since an optional **Type** clause is used, the table will be built around a dBASE file.

```
Create Table Towns
( townname Char(30),
   population SmallInt,
   median_income Decimal(9,2) )
   File "C:\MAPINFO\TEMP\TOWNS"
   Type DBF
```

Examples for TILESERVER

The following examples show how to create tile server tables for various tile servers. In the examples, the table name and file name are determined by the users. The attribution text should be the attribution legally required by the provider of the server.

The following example shows how to create a tile server table which uses a **MapInfo Developer** tile server:

```
Create Table MIDev_TileServer
   File "MIDev_TileServer"
   Type TILESERVER
   TileType "LevelRowColumn"
   URL
   "http://INSERT_SERVER_NAME_HERE/MapTilingService/MapName/{LEVEL}/{ROW}
   :{COL}/tile.gif"
   AttributionText "required attribution text"
   Font("Verdena",255,16,0,255)
   StartTileNum 1
   MaxLevel 20
   Height 256
   CoordSys Earth Projection 10, 157, 7, 0 Bounds(-20037508.34,-
   20037508.34) (20037508.34,20037508.34)
```

The following example shows how to create a tile server table which uses a **MapXtreme.NET** tile server:

```
Create Table MXT_TileServer
   File "MXT_TileServer"
   Type TILESERVER
```

```
   TileType "LevelRowColumn"
   URL
   "http://INSERT_SERVER_NAME_HERE/TileServer/MapName/{LEVEL}/{ROW};{COL}
   /tile.png"
   AttributionText "required attribution text"
   Font("Calibri",255,16,0,255)
   MaxLevel 20
   Height 256
   RequestTimeout 90
   ReadTimeout 60
   CoordSys Earth Projection 10, 157, 7, 0 Bounds(-20037508.34,-
   20037508.34) (20037508.34,20037508.34)
```

The following example shows how to create a tile server table which uses an **OpenStreetMap** tile server:

```
Create Table OSM_TileServer
   File "OSM_TileServer"
   Type TILESERVER
   TileType "LevelRowColumn"
   URL
   "http://INSERT_OPEN_STREET_MAP_SERVER_NAME_HERE/{LEVEL}/{ROW}/{COL}.pn
   g"
   AttributionText "required attribution text" Font("Arial",255,16,0,255)
   MinLevel 0
   MaxLevel 15
   Height 256
   CoordSys Earth Projection 10, 157, 7, 0 Bounds(-20037508.34,-
   20037508.34) (20037508.34,20037508.34)
```

See Also:

Alter Table statement, **Create Index statement**, **Create Map statement**, **Drop Table statement**, **Export statement**, **Import statement**, **Open Table statement**

CreateText( ) function

Purpose

Returns a text object created for a specific map window. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

CreateText(*window_id*, *x*, *y*, *text*, *angle*, *anchor*, *offset*)

window_id is an integer window identifier that represents a Map window.

x, *y* are float values, representing the x/y location where the text is anchored.

text is a string value, representing the text that will comprise the text object.

angle is a float value, representing the angle of rotation; for horizontal text, specify zero.

anchor is an integer value from 0 to 8, controlling how the text is placed relative to the anchor location. Specify one of the following codes; codes are defined in MAPBASIC.DEF.

```
LAYER_INFO_LBL_POS_CC (0)
LAYER_INFO_LBL_POS_TL (1)
LAYER_INFO_LBL_POS_TC (2)
LAYER_INFO_LBL_POS_TR (3)
LAYER_INFO_LBL_POS_CL (4)
LAYER_INFO_LBL_POS_CR (5)
LAYER_INFO_LBL_POS_BL (6)
LAYER_INFO_LBL_POS_BC (7)
LAYER_INFO_LBL_POS_BR (8)
```

The two-letter suffix indicates the label orientation: T=Top, B=Bottom, C=Center, R=Right, L=Left. For example, to place the text below and to the right of the anchor location, specify the define code LAYER_INFO_LBL_POS_BR, or specify the value 8.

offset is an integer from zero to 200, representing the distance (in points) the text is offset from the anchor location; offset is ignored if anchor is zero (centered).

Return Value

Object

Description

The **CreateText( )** function returns an Object value representing a text object.

The text object uses the current Font style. To create a text object with a specific Font style, issue the **Set Style statement** before calling **CreateText( )**.

At the moment the text is created, the text height is controlled by the current Font. However, after the text object is created, its height depends on the Map window's zoom; zooming in will make the text appear larger.

The object returned could be assigned to an Object variable, stored in an existing row of a table (through the **Update statement**), or inserted into a new row of a table (through an **Insert statement**).

Example

The following example creates a text object and inserts it into the map's Cosmetic layer (given that the variable i_map_id is an integer containing a Map window's ID).

```
Insert Into Cosmetic1 (Obj)
   Values ( CreateText(i_map_id, -80, 42.4, "Sales Map", 0,0,0) )
```

See Also:

AutoLabel statement, **Create Text statement**, **Font clause**, **Insert statement**, **Update statement**

Create Text statement

Purpose

Creates a text object, such as a title, for a Map or Layout window. You can use the pen clause to persist the new label line styles in layouts. Changing layouts in this manner sets the version of the workspace to 9.5. Any MIF file that contains text objects with a Label line and a pen clause will be version 950 as a result. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Create Text
   [ Into { Window window_id | Variable var_name } ]
    text_string
   ( x1, y1 ) ( x2, y2 )
   [ Font... ]
   [ Label Line { Simple | Arrow } ( label_x, label_y ) Pen (pen_expr) ]
   [ Spacing { 1.0 | 1.5 | 2.0 } ]
   [ Justify { Left | Center | Right } ]
   [ Angle text_angle ]
```

window_id is an integer window ID number, identifying a Map or Layout window.

var_name is the name of an existing object variable.

text_string specifies the string, up to 255 characters long, that will constitute the text object; to create a multiple-line text object, embed the function call Chr$(10) in the string.

x1, *y1* are floating-point coordinates, specifying one corner of the rectangular area which the text will fill.

x2, *y2* specify the opposite corner of the rectangular area which the text will fill.

The **Font clause** specifies a text style. The point-size element of the Font is ignored if the text object is created in a Map window; see below.

label_x, *label_y* specifies the position where the text object's label line is anchored.

Pen specifies the pen clause settings of callouts created in the Layout window.

text_angle is a float value indicating the angle of rotation for the text object (in degrees).

Example

When the user creates a label line in a Layout window, the Create Text Label Line Pen clause is invoked and the workspace version is incremented to 950:

```
!Workspace
!Version 950
!Charset WindowsLatin1
Open Table "Data\Introductory_Data\World\WORLD" As WORLD Interactive
Map From WORLD
  Position (0.0520833,0.0520833) Units "in"
```

```
  Width 6.625 Units "in" Height 4.34375 Units "in"
Set Window FrontWindow() ScrollBars Off Autoscroll On
Set Map
  CoordSys Earth Projection 1, 104
  Center (35.204159,-25.3575215)
  Zoom 18063.92971 Units "mi"
  Preserve Zoom Display Zoom
  Distance Units "mi" Area Units "sq mi" XY Units "degree"
Set Map
  Layer 1
    Display Graphic
    Global Pen (1,2,0)  Brush (2,16777215,16777215)  Symbol (35,0,12)
Line (1,2,0)  Font ("Arial",0,9,0)
    Label Line None Position Center Font ("Arial",0,9,0) Pen (1,2,0)
```

Description

The *x* and *y* parameters use whatever coordinate system MapBasic is currently using. By default, MapBasic uses a Longitude/Latitude coordinate system, although the **Set CoordSys statement** can re-configure MapBasic to use a different coordinate system. If you need to create objects on a Layout window, you must first issue a **Set CoordSys Layout statement**.

The *x1*, *y1*, *x2*, and *y2* arguments define a rectangular area. When you create text in a Map window, the text fills the rectangular area, which controls the text height; the point size specified in the **Font clause** is ignored. In a Layout window, text is drawn at the point size specified in the Font clause, with the upper-left corner of the text placed at the (*x1*, *y1*) location; the (*x2*, *y2*) arguments are ignored.

See Also:

AutoLabel statement, **CreateText( ) function**, **Font clause**, **Insert statement**, **Update statement**

CurDate( ) function

Purpose

Returns the current date in YYYYMMDD format. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

`CurDate``( )`

Return Value

Date

Description

The **Curdate( )** function returns a Date value representing the current date. The format will always be YYYYMMDD. To change the value to a string in the local system format use the **FormatDate$( ) function** or **Str$( ) function**.

Example

```
Dim d_today As Date
d_today = CurDate( )
```

See Also:

Day( ) function, **Format$( ) function**, **Month( ) function**, **StringToDate( ) function**, **Timer( ) function**, **Weekday( ) function**, **Year( ) function**

CurDateTime() function

Purpose

Returns the current date and time. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

```
CurDateTime
```

Return Value

DateTime

Example

Copy this example into the MapBasic window for a demonstration of this function.

```
dim X as datetime
X = CurDateTime()
Print X
```

CurrentBorderPen( ) function

Purpose

Returns the current border pen style currently in use. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

CurrentBorderPen( )

Return Value

Pen

Description

The **CurrentBorderPen( )** function returns the current border pen style. MapInfo Professional assigns the current style to the border of any region objects drawn by the user. If a MapBasic program creates an object through a statement such as **Create Region statement**, but the statement does not include a **Pen clause**, the object uses the current BorderPen style.

The return value can be assigned to a Pen variable, or may be used as a parameter within a statement that takes a Pen setting as a parameter (such as **Set Map statement**).

To extract specific attributes of the Pen style (such as the color), call the **StyleAttr( ) function**. For more information about Pen settings, see **Pen clause**.

Example

```
Dim p_user_pen As Pen p_user_pen = CurrentBorderPen( )
```

See Also:

CurrentPen( ) function, **Pen clause**, **Set Style statement**, **StyleAttr( ) function**

CurrentBrush( ) function

Purpose

Returns the Brush (fill) style currently in use. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

CurrentBrush( )

Return Value

Brush

Description

The **CurrentBrush( )** function returns the current Brush style. This corresponds to the fill style displayed in the **Options > Region Style** dialog box. MapInfo Professional assigns the current Brush value to any filled objects (ellipses, rectangles, rounded rectangles, or regions) drawn by the user. If a MapBasic program creates a filled object through a statement such as the **Create Region statement**, but the statement does not include a **Brush clause**, the object will be assigned the current Brush value.

The return value of the **CurrentBrush( )** function can be assigned to a Brush variable, or may be used as a parameter within a statement that takes a Brush setting as a parameter (such as **Set Map statement** or **Shade statement**).

To extract specific Brush attributes (such as the color), call the **StyleAttr( ) function**.

For more information about Brush settings, see **Brush clause**.

Example

```
Dim b_current_fill As Brush
b_current_fill = CurrentBrush( )
```

See Also:

Brush clause, **MakeBrush( ) function**, **Set Style statement**, **StyleAttr( ) function**

CurrentFont( ) function

Purpose

Returns the Font style currently in use for Map and Layout windows. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

CurrentFont( )

Return Value

Font

Description

The **CurrentFont( )** function returns the current Font style. This corresponds to the text style displayed in the **Options > Text Style** dialog box when a Map or Layout window is the active window. MapInfo Professional will assign the current Font value to any text object drawn by the user. If a MapBasic program creates a text object through the **Create Text statement**, but the statement does not include a **Font clause**, the text object will be assigned the current Font value.

The return value of the **CurrentFont( )** function can be assigned to a Font variable, or may be used as a parameter within a statement that takes a Font setting as a parameter (such as **Set Legend statement**).

To extract specific attributes of the Font style (such as the color), call the **StyleAttr( ) function**.

For more information about Font settings, see **Font clause**.

Example

```
Dim f_user_text As Font
f_user_text = CurrentFont( )
```

See Also:

Font clause, **MakeFont( ) function**, **Set Style statement**, **StyleAttr( ) function**

CurrentLinePen( ) function

Purpose

Returns the Pen (line) style currently in use. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

```
CurrentLinePen( )
```

Return Value

Pen

Description

The **CurrentLinePen( )** function returns the current Pen style. MapInfo Professional assigns the current style to any line or polyline objects drawn by the user. If a MapBasic program creates an object through a statement such as **Create Line statement**, but the statement does not include a Pen clause, the object uses the current Pen style. The return value can be assigned to a Pen variable, or may be used as a parameter within a statement that takes a Pen setting as a parameter (such as **Set Map statement**).

To extract specific attributes of the Pen style (such as the color), call the **StyleAttr( ) function**. For more information about Pen settings, see **Pen clause**.

Example

```
Dim p_user_pen As Pen p_user_pen = CurrentPen( )
```

See Also:

CurrentBorderPen( ) function, **Pen clause**, **Set Style statement**, **StyleAttr( ) function**

CurrentPen( ) function

Purpose

Returns the Pen (line) style currently in use and sets the border pen to the same style as the line pen. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

```
CurrentPen( )
```

Return Value

Pen

Description

The **CurrentPen( )** function returns the current Pen style. MapInfo Professional assigns the current style to any line or polyline objects drawn by the user. If a MapBasic program creates an object through a statement such as the **Create Line statement**, but the statement does not include a Pen clause, the object uses the current Pen style. If you want to use the current line pen without re-setting the border pen, use the **CurrentLinePen( ) function**.

The return value can be assigned to a Pen variable, or may be used as a parameter within a statement that takes a Pen setting as a parameter (such as the **Set Map statement**).

To extract specific attributes of the Pen style (such as the color), call the **StyleAttr( ) function**. For more information about Pen settings, see **Pen clause**.

Example

```
Dim p_user_pen As Pen
p_user_pen = CurrentPen( )
```

See Also:

MakePen( ) function, **Pen clause**, **Set Style statement**, **StyleAttr( ) function**

CurrentSymbol( ) function

Purpose

Returns the Symbol style currently in use. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

CurrentSymbol( )

Return Value

Symbol

Description

The **CurrentSymbol( )** function returns the current symbol style. This is the style displayed in the **Options > Symbol Style** dialog box. MapInfo Professional assigns the current Symbol style to any point objects drawn by the user. If a MapBasic program creates a point object through a **Create Point statement**, but the statement does not include a Symbol clause, the object will be assigned the current Symbol value.

The return value of the **CurrentSymbol( )** function can be assigned to a Symbol variable, or may be used as a parameter within a statement that takes a Symbol clause as a parameter (such as **Set Map statement** or **Shade statement**).

To extract specific attributes of the Symbol style (such as the color), call the **StyleAttr( ) function**. For more information about Symbol settings, see **Symbol clause**.

Example

```
Dim sym_user_symbol As Symbol
sym_user_symbol = CurrentSymbol( )
```

See Also:

MakeSymbol( ) function, **Set Style statement**, **StyleAttr( ) function**, **Symbol clause**

CurTime() function

Purpose

Returns the current time. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

```
CurTime
```

Return Value

Time

Example

Copy this example into the MapBasic window for a demonstration of this function.

```
dim Y as time
Y = CurTime()
Print Y
```

DateWindow( ) function

Purpose

Returns the current date window setting as an integer in the range 0 to 99, or (-1) if date windowing is off. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

DateWindow(*context*)

context is a SmallInt that can either be DATE_WIN_CURPROG (2) or DATE_WIN_SESSION (1).

Description

This depends on which context is passed. If context is DATE_WIN_SESSION (1), then the current session setting in effect is returned. If context is DATE_WIN_CURPROG (2), then the current MapBasic program's local setting is returned, if a program is not running the session setting is returned.

Example

In the following example the variable Date1 = 19890120, Date2 = 20101203 and MyYear = 1990.

```
DIM Date1, Date2 as Date
DIM MyYear As Integer
Set Format Date "US"
Set Date Window 75
   Date1 = StringToDate("1/20/89")
   Date2 = StringToDate("12/3/10")
   MyYear = Year("12/30/90")
```

See Also:

Set Date Window statement

Day( ) function

Purpose

Returns the day component from a Date expression. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Day(*date_expr*)

date_expr is a Date expression.

Return Value

SmallInt from 1 to 31

Description

The **Day( )** function returns an integer value from one to thirty-one, representing the day-of-the-month component of the specified date. For example, if the specified date is 12/17/93, the **Day( )** function returns a value of 17.

Example

```
Dim day_var As SmallInt, date_var As Date
date_var = StringToDate("05/23/1985")
day_var = Day(date_var)
```

See Also:

CurDate( ) function, **Month( ) function**, **Timer( ) function**, **Year( ) function**

DDEExecute statement

Purpose

Issues a command across an open DDE channel.

Syntax

DDEExecute *channel, command*

channel is an integer channel number returned by DDEInitiate( ).

command is a string representing a command for the DDE server to execute.

Description

The **DDEExecute** statement sends a command string to the server application in a DDE conversation.

The channel parameter must correspond to the number of a channel opened through a **DDEInitiate( ) function** call.

The command parameter string must represent a command which the DDE server (the passive application) is able to carry out. Different applications have different requirements regarding what constitutes a valid command; to learn about the command format for a particular application, see the documentation for that application.

Error Conditions

ERR_CMD_NOT_SUPPORTED (642) error generated if not running on Windows.

ERR_NO_RESPONSE_FROM_APP (697) error if server application does not respond.

Example

Through MapBasic, you can open a DDE channel with Microsoft Excel as the server application. If the conversation specifies the “System” topic, you can use the **DDEExecute** statement to send Excel a command string. Provided that the command string is equivalent to an Excel macro function, and provided that the command string is enclosed in square brackets, Excel can execute the command. The example below instructs Excel to open the worksheet “TRIAL.XLS”.

```
Dim i_chan As Integer
i_chan = DDEInitiate("Excel", "System")
DDEExecute i_chan, "[OPEN(""C:\DATA\TRIAL.XLS"")]"
```

See Also:

DDEInitiate( ) function, **DDEPoke statement**, **DDERequest$( ) function**

DDEInitiate( ) function

Purpose

Initiates a new DDE conversation, and returns the associated channel number.

Syntax

DDEInitiate(*appl_name*, *topic_name*)

appl_name is a string representing an application name (for example, “MapInfo”).

topic_name is a string representing a topic name (for example, “System”).

Return Value

Integer

Description

The **DDEInitiate( )** function initiates a DDE (Dynamic Data Exchange) conversation, and returns the number that identifies that conversation's channel.

A DDE conversation allows two Microsoft Windows applications to exchange information. Once a DDE conversation has been initiated, a MapBasic program can issue **DDERequest$( ) function** calls (to read information from the other application) and **DDEPoke statement**s (to write information to the other application). Once a DDE conversation has served its purpose and is no longer needed, the MapBasic program should terminate the conversation through the **DDETerminate statement** or **DDETerminateAll statement**.

DDE conversations are a feature specific to Microsoft Windows; therefore, MapBasic generates an error if a program issues DDE-related function calls when running on a non-Windows platform. To determine the current hardware platform at run-time, call the **SystemInfo( ) function**.

The *appl_name* parameter identifies a Windows application. For example, to initiate a conversation with Microsoft Excel, you should specify the *appl_name* parameter "Excel." The application named by the *appl_name* parameter must already be running before you can initiate a DDE conversation; note that the MapBasic **Run Program statement** allows you to run another Windows application. Not all Windows applications support DDE conversations. To determine if an application supports DDE conversations, see the documentation for that application.

The *topic_name* parameter is a string that identifies the topic for the conversation. Each application has its own set of valid topic names; for a list of topics supported by a particular application, refer to the documentation for that application. With many applications, the name of a file that is in use is a valid topic name. Thus, if Excel is currently using the worksheet file "ORDERS.XLS", you could issue the following MapBasic statements:

```
Dim i_chan As Integer
i_chan = DDEInitiate("Excel", "C:\ORDERS.XLS")
```

to initiate a DDE conversation with that Excel worksheet.

Many applications support a special topic called "System". If you initiate a conversation using the "System" topic, you can then use the **DDERequest$( ) function** to obtain a list of the strings which the application accepts as valid topic names (for example, a list of the files that are currently in use). Knowing what topics are available, you can then initiate another DDE conversation with a specific document. See the example below.

The following table lists some sample application and topic names which you could use with the **DDEInitiate( )** function.

DDEInitiate( ) call	Nature of conversation
`DDEInitiate("Excel", "System")`	**DDERequest$( ) function** calls can return Excel system information, such as a list of the names of the worksheets in use; **DDEExecute statement**s can send commands for Excel to execute.
`DDEInitiate("Excel", wks)`	If *wks* is the name of an Excel document in use, subsequent **DDEPoke statement**s can store values in the worksheet, and **DDERequest$( ) function** calls can read information from the worksheet.
`DDEInitiate("MapInfo", "System")`	**DDERequest$( ) function** calls can provide system information, such as a list of the MapBasic applications currently in use by MapInfo Professional.
`DDEInitiate("MapInfo" mbx)`	If *mbx* is the name of a MapBasic application in use, **DDEPoke statement**s can assign values to global variables in the specified application, and **DDERequest$( ) function** calls can read the current values of global variables.

When a MapBasic program issues a **DDEInitiate( )** function call, the MapBasic program is known as the “client” in the DDE conversation. The other Windows application is known as the “server.” Within one particular conversation, the client is always the active party; the server merely responds to actions taken by the client. A MapBasic program can carry on multiple conversations at the same time, limited only by memory and system resources. A MapBasic application could act as the client in one conversation (by issuing statements such as **DDEInitiate( )**, etc.) while acting as the server in another conversation (by defining a **RemoteMsgHandler procedure**).

Error Conditions

ERR_CMD_NOT_SUPPORTED (642) error generated if not running on Windows.

ERR_INVALID_CHANNEL (696) error generated if the specified channel number is invalid.

Example

The following example attempts to initiate a DDE conversation with Microsoft Excel, version 4 or later. The goal is to store a simple text message (“Hello from MapInfo!”) in the first cell of a worksheet that Excel is currently using, but only if that cell is currently empty. If the first cell is not empty, we will not overwrite its current contents.

```
Dim chan_num, tab_marker As Integer
Dim topiclist, topicname, cell As String

chan_num = DDEInitiate("EXCEL", "System")
If chan_num = 0 Then
   Note "Excel is not responding to DDE conversation."
   End Program
End If

' Get a list of Excel's valid topics
topiclist = DDERequest$(chan_num, "topics")

' If Excel 4 is running, topiclist might look like:
'     ": Sheet1 System"
' (if spreadsheet is still "unnamed"),or like:
'     ": C:Orders.XLS Sheet1 System"
'
' If Excel 5 is running, topiclist might look like:
'     "[Book1]Sheet1 [Book2]Sheet2 ..."
'
' Next, extract just the first topic (for example,"Sheet1")
' by extracting the text between the 1st & 2nd tabs;
' or, in the case of Excel 5, by extracting the text
' that appears before the first tab.

If Left$(topiclist, 1) = ":" Then
   ' ...then it's Excel 4.
   tab_marker = InStr(3, topiclist, Chr$(9) )
   If tab_marker = 0 Then
      Note "No Excel documents in use! Stopping."
      End Program
```

```
   End If
   topicname = Mid$(topiclist, 3, tab_marker - 3)
   Else
      ' ... assume it's Excel 5.
      tab_marker = Instr(1, topiclist, Chr$(9) )
      topicname = Left$( topiclist, tab_marker - 1)
   End If

' open a channel to the specific document
' (e.g., "Sheet1")
DDETerminate chan_num
chan_num = DDEInitiate("Excel", topicname)
If chan_num = 0 Then
   Note "Problem communicating with " + topicname End Program
End If

' Let's examine the 1st cell in Excel.
' If cell is blank, put a message in the cell.
' If cell isn't blank, don't alter it -
' just display cell contents in a MapBasic NOTE.
' Note that a "Blank cell" gets returned as a
' carriage-return line-feed sequence:
'    Chr$(13) + Chr$(10).
cell = DDERequest$( chan_num, "R1C1" )
If cell <> Chr$(13) + Chr$(10) Then
   Note
      "Message not sent; cell already contains:" + cell
   Else
      DDEPoke chan_num, "R1C1", "Hello from MapInfo!"
      Note "Message sent to Excel,"+topicname+ ",R1C1."
   End If
DDETerminateAll
```

This example does not anticipate every possible obstacle. For example, Excel might currently be editing a chart (for example, “Chart1”) instead of a worksheet, in which case we will not be able to reference cell “R1C1”.

See Also:

DDEExecute statement, **DDEPoke statement**, **DDERequest$( ) function**, **DDETerminate statement**, **DDETerminateAll statement**

DDEPoke statement

Purpose

Sends a data value to an item in a DDE server application.

Syntax

DDEPoke *channel, itemname, data*

channel is an integer channel number returned by the **DDEInitiate( ) function**.

itemname is a string value representing the name of an item.

data is a character string to be sent to the item named in the itemname parameter.

Description

The **DDEPoke** statement stores the data text string in the specified DDE item.

The channel parameter must correspond to the number of a channel which was opened through the **DDEInitiate( ) function**.

The *itemname* parameter should identify an item which is appropriate for the specified channel. Different DDE applications support different item names; to learn what item names are supported by a particular Windows application, refer to the documentation for that application.

In a DDE conversation with Excel, a string of the form R1C1 (for Row 1, Column 1) is a valid item name. In a DDE conversation with another MapBasic application, the name of a global variable in the application is a valid item name.

Error Conditions

ERR_CMD_NOT_SUPPORTED (642) error generated if not running on Windows.

ERR_INVALID_CHANNEL (696) error generated if the specified channel number is invalid.

Example

If Excel is already running, the following example stores a simple message (“Hello from MapInfo!”) in the first cell of an Excel worksheet.

```
Dim i_chan_num As Integer
i_chan_num = DDEInitiate("EXCEL", "Sheet1")
DDEPoke i_chan_num, "R1C1", "Hello from MapInfo!"
```

The following example assumes that there is another MapBasic application currently in use—”Dispatch.mbx”—and assumes that the Dispatch application has a global variable called Address. The example below uses **DDEPoke** to modify the Address global variable.

```
i_chan_num = DDEInitiate("MapInfo","C:\DISPATCH.MBX")
DDEPoke i_chan_num, "Address", "23 Main St."
```

See Also:

DDEExecute statement, **DDEInitiate( ) function**, **DDERequest$( ) function**

DDERequest$( ) function

Purpose

Returns a data value obtained from a DDE conversation.

Syntax

DDERequest$(*channel*, *itemname*)

channel is an integer channel number returned by the **DDEInitiate( ) function**.

itemname is a string representing the name of an item in the server application.

Return Value

String

Description

The **DDERequest$( )** function returns a string of information obtained through a DDE conversation. If the request is unsuccessful, the **DDERequest$( )** function returns a null string.

The channel parameter must correspond to the number of a channel which was opened through the **DDEInitiate( ) function**.

The *itemname* parameter should identify an item which is appropriate for the specified channel. Different DDE applications support different item names; to learn what item names are supported by a particular Windows application, refer to the documentation for that application.

The following table lists some topic and item combinations that can be used when conducting a DDE conversation with Microsoft Excel as the server:

Topic name	Item names to use with DDERequest$( )
"System"	"Systems" returns a list of item names accepted under the "System" topic; "Topics" returns a list of DDE topic names accepted by Excel, including the names of all open worksheets; "Formats" returns a list of clipboard formats accepted by Excel (for example, "TEXT BITMAP …")
wks (name of a worksheet in use)	A string of the form R1C1 (for Row 1, Column 1) returns the contents of that cell

ⓘ Through the **DDERequest$( )** function, one MapBasic application can observe the current values of global variables in another MapBasic application. The following table lists the topic and item combinations that can be used when conducting a DDE conversation with MapInfo Professional as the server.

Topic name	item names to use with DDERequest$( )
"System"	"Systems" returns a list of item names accepted under the "System" topic; "Topics" returns a list of DDE topic names accepted by MapInfo Professional, which includes the names of all MapBasic applications currently in use; "Formats" returns a list of clipboard formats accepted by MapInfo Professional ("TEXT"); "Version" returns the MapInfo version number, multiplied by 100.
mbx (name of .MBX in use)	"{items}" returns a list of the names of global variables in use by the specified MapBasic application; specifying the name of a global variable lets **DDERequest$( )** return the value of the variable

Error Conditions

ERR_CMD_NOT_SUPPORTED (642) error generated if not running on Windows.

ERR_INVALID_CHANNEL (696) error if the specified channel number is invalid.

ERR_CANT_INITIATE_LINK (698) error generated if MapBasic cannot link to the topic.

Example

The following example uses the **DDERequest$( )** function to obtain the current contents of the first cell in an Excel worksheet. Note that this example will only work if Excel is already running.

```
Dim i_chan_num As Integer
Dim s_cell As String
i_chan_num = DDEInitiate("EXCEL", "Sheet1")
s_cell = DDERequest$(i_chan_num, "R1C1")
```

The following example assumes that there is another MapBasic application currently in use—”Dispatch”—and assumes that the Dispatch application has a global variable called Address. The example below uses **DDERequest$( )** to obtain the current value of the Address global variable.

```
Dim i_chan_num As Integer, s_addr_copy As String
i_chan_num = DDEInitiate("MapInfo","C:\DISPATCH.MBX")
s_addr_copy = DDERequest$(i_chan_num, "Address")
```

See Also:

DDEInitiate( ) function

DDETerminate statement

Purpose

Closes a DDE conversation.

Syntax

DDETerminate *channel*

channel is an integer channel number returned by the **DDEInitiate( ) function**.

Description

The **DDETerminate** statement closes the DDE channel specified by the channel parameter.

The channel parameter must correspond to the channel number returned by the **DDEInitiate( ) function** call (which initiated the conversation). Once a DDE conversation has served its purpose and is no longer needed, the MapBasic program should terminate the conversation through the **DDETerminate** statement or the **DDETerminateAll statement**.

Multiple MapBasic applications can be in use simultaneously, and each application can open its own DDE channels. However, a given MapBasic application may only close the DDE channels which it opened. A MapBasic application may not close DDE channels which were opened by another MapBasic application.

Error Conditions

ERR_CMD_NOT_SUPPORTED (642) error generated if not running on Windows.

ERR_INVALID_CHANNEL (696) error generated if the specified channel number is invalid.

Example

```
DDETerminate i_chan_num
```

See Also:

DDEInitiate( ) function, **DDETerminateAll statement**

DDETerminateAll statement

Purpose

Closes all DDE conversations which were opened by the same MapBasic program.

Syntax

DDETerminateAll

Description

The **DDETerminateAll** statement closes all open DDE channels which were opened by the same MapBasic application. Note that multiple MapBasic applications can be in use simultaneously, and each application can open its own DDE channels. However, a given MapBasic application may only close the DDE channels which it opened. A MapBasic application may not close DDE channels which were opened by another MapBasic application.

Once a DDE conversation has served its purpose and is no longer needed, the MapBasic program should terminate the conversation through the **DDETerminate statement** or the **DDETerminateAll** statement.

Error Conditions

ERR_CMD_NOT_SUPPORTED (642) error generated if not running on Windows.

See Also:

DDEInitiate( ) function, **DDETerminate statement**

Declare Function statement

Purpose

Defines the name and parameter list of a function.

Restrictions

This statement may not be issued from the MapBasic window.

Accessing external functions (using syntax 2) is platform-dependent. DLL files may only be accessed by applications running on Windows.

Syntax 1

```
Declare Function fname
   ([ [ ByVal ] parameter As var_type ]
    [ , [ ByVal ] parameter As var_type... ] ) As return_type
```

fname is the name of the function.

parameter is the name of a parameter to the function.

var_type is a variable type, such as integer; arrays and custom Types are allowed.

return_type is a standard scalar variable type; arrays and custom Types are not allowed.

Syntax 2 (external routines in Windows DLLs)

```
Declare Function fname Lib "file_name" [ Alias "function_alias" ]
   ( [ [ ByVal ] parameter As var_type ]
    [, [ ByVal ] parameter As var_type... ] ) As return_type
```

fname is the name by which a function will be called.

file_name is the name of a Windows DLL file.

function_alias is the original name of the external function.

parameter is the name of a parameter to the function.

var_type is a data type: with Windows DLLs, this can be a standard variable type or a custom Type.

return_type is a standard scalar variable type.

Description

The **Declare Function** statement pre-declares a user-defined MapBasic function or an external function.

A MapBasic program can use a **Function…End Function statement** to create a custom function. Every function defined in this fashion must be preceded by a **Declare Function** statement. For more information on creating custom functions, see **Function…End Function statement**.

Parameters passed to a function are passed by reference unless you include the optional **ByVal** keyword. For information on the differences between by-reference and by-value parameters, see the *MapBasic User Guide*.

Calling External Functions

Using Syntax 2 (above), you can use a **Declare Function** statement to define an external function. An external function is a function that was written in another language (for example, C or Pascal), and is stored in a separate file. Once you have declared an external function, your program can call the external function as if it were a conventional MapBasic function.

If the **Declare Function** statement declares an external function, the *file_name* parameter must specify the name of the file containing the external function. The external file must be present at run-time.

Every external function has an explicitly assigned name. Ordinarily, the **Declare Function** statement's *fname* parameter matches the explicit routine name from the external file. Alternately, the **Declare Function** statement can include an **Alias** clause, which lets you call the external function by whatever name you choose. The **Alias** clause lets you override an external function's explicit name, in situations where the explicit name conflicts with the name of a standard MapBasic function.

If the **Declare Function** statement includes an **Alias** clause, the *function_alias* parameter must match the external function's original name, and the *fname* parameter indicates the name by which MapBasic will call the routine.

Restrictions on Windows DLL parameters

You can pass a custom variable type as a parameter to a DLL. However, the DLL must be compiled with “structure packing” set to the tightest packing. See the *MapBasic User Guide* for more information.

Example

The following example defines a custom function, CubeRoot, which returns the cube root of a number (the number raised to the one-third power).

```
Declare Sub Main
Declare Function CubeRoot(ByVal x As Float) As Float
Sub Main
   Note Str$( CubeRoot(23) )
End Sub
Function CubeRoot(ByVal x As Float) As Float
   CubeRoot = x ^ (1 / 3)
End Function
```

See Also:

Declare Sub statement, **Function…End Function statement**

Declare Method statement

Purpose

Defines the name and argument list of a method/function in a .Net assembly, so that a MapBasic application can call the function.

Restrictions

This statement may not be issued from the MapBasic window.

Syntax

```
Declare Method fname  Class "class_name"  Lib "assembly_name"
     [ Alias function_alias ]
   ( [ [ ByVal ] parameter As var_type ]
```

```
[, [ ByVal ] parameter As var_type... ] ) [ As return_type ]
```

fname is the name by which a function will be called; if the optional Alias clause is omitted, fname must be the same as the actual .Net method/function name. This option can not be longer than 31 characters.

class_name is the name of the .Net class that provides the function to be called, including the class's namespace (such as System.Windows.Forms.MessageBox)

assembly_name is the name of a .Net assembly file, such as filename.dll. If the assembly is to be loaded from the GAC, *assembly_name* must be a fully qualified assembly name.

function_alias is the original name of the .Net method/function (the name as defined in the .Net assembly). Note: Include the Alias clause only when you want to call the method by a name other than its original name.

parameter is the name of a parameter to the function.

var_type is a MapBasic data type, such as Integer

return_type is a standard MapBasic scalar variable type, such as Integer. If the As clause is omitted, the MapBasic program can call the method as a Sub (using the **Call statement**).

Description

The Declare Method statement allows a MapBasic program to call a method (function or procedure) from a .Net assembly. The .Net assembly can be created using various languages, such as C# or VB.Net. For details on calling .Net from MapBasic, see the MapBasic *User Guide*.

MapBasic programs can only call .Net methods or functions that are declared as static. (VB.NET refers to such functions as "shared functions," while C# refers to them as "static methods.")

At run time, if the *assembly_name* specifies a fully-qualified assembly name, and if the assembly is registered in the Global Assembly Cache (GAC), MapInfo Professional will load the assembly from the GAC. Otherwise, the assembly will be loaded from the same directory as the .MBX file (in which case, assembly_name should be a filename such as "filename.dll"). Thus, you can have your assembly registered in the GAC, but you are not required to do so.

Examples

Here is a simple example of a C# class that provides a static method:

```
namespace MyProduct
{
    class MyWrapper
    {
        public static int ShowMessage(string s)
        {
            System.Windows.Forms.MessageBox.Show(s);
            return 0;
        }
    }
}
```

In VB.Net, the class definition might look like this.

```
Namespace MyProduct

  Public Class MyWrapper

    Public Shared Function ShowMessage(ByVal s As String) As Integer
       System.Windows.Forms.MessageBox.Show(s)
       Return 0
    End Function

  End Class

End Namespace
```

A MapBasic program could call the method with this syntax:

```
Declare Method ShowMessage
    Class "MyProduct.MyWrapper"
    Lib "MyAssembly.DLL" (ByVal str As String) As Integer

. . .

Dim retval As Integer
retval = ShowMessage("Here I am")
```

The following example demonstrates how to declare methods in assemblies that are registered in the GAC. Note that when an assembly is loaded from the GAC, the Lib clause must specify a fully-qualified assembly name. Various utilities exist that can help you to identify an assembly's fully-qualified name, including the gacutil utility provided by Microsoft as part of Visual Studio.

```
' Declare a method from the System.Windows.Forms.dll assembly:
Declare Method Show
  Class "System.Windows.Forms.MessageBox"
  Lib "System.Windows.Forms, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089"
  (ByVal str As String, ByVal caption As String)

' Declare a method from the mscorlib.dll assembly:
Declare Method Move
  Class "System.IO.File"
     Lib "mscorlib, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089"
     (ByVal sourceFileName As String, ByVal destFileName As String)


' Display a .Net MessageBox dialog box with both a message and a caption:
Call Show("Table update is complete.", "Tool name")

' Call the .Net Move method to move a file
Call Move("C:\work\pending\entries.txt", "C:\work\finished\entries.txt")
```

Declare Sub statement

Purpose

Identifies the name and parameter list of a sub procedure.

Restrictions

This statement may not be issued from the MapBasic window.

Accessing external functions (using Syntax 2) is platform-dependent. DLL files may only be accessed by applications running on Windows.

Syntax 1

```
Declare Sub sub_proc
   [ ( [ ByVal ] parameter As var_type [ , ... ] ) ]
```

sub_proc is the name of a sub procedure.

parameter is the name of a sub procedure parameter.

var_type is a standard data type or a custom Type.

Syntax 2 (external routines in Windows DLLs)

```
Declare Sub sub_proc Lib "file_name" [ Alias "sub_alias" ]
   [ ( [ ByVal ] parameter As var_type [ , ... ] ) ]
```

sub_proc is the name by which an external routine will be called.

file_name is a string; the DLL name.

sub_alias is an external routine's original name.

parameter is the name of a sub procedure parameter.

var_type is a data type: with Windows DLLs, this can be a standard variable type or a custom Type.

Description

The **Declare Sub** statement establishes a sub procedure's name and parameter list. Typically, each **Declare Sub** statement corresponds to an actual sub procedure which appears later in the same program.

A MapBasic program can use a **Sub…End Sub statement** to create a procedure. Every procedure defined in this manner must be preceded by a **Declare Sub** statement. For more information on creating procedures, see **Sub…End Sub statement**.

Parameters passed to a procedure are passed by reference unless you include the optional **ByVal** keyword.

Calling External Routines

Using Syntax 2 (above), you can use a **Declare Sub** statement to define an external routine. An external routine is a routine that was written in another language (for example, C or Pascal), and is stored in a separate file. Once you have declared an external routine, your program can call the external routine as if it were a conventional MapBasic procedure.

If the **Declare Sub** statement declares an external routine, the *file_name* parameter must specify the name of the file containing the routine. The file must be present at run-time.

Every external routine has an explicitly assigned name. Ordinarily, the **Declare Sub** statement's *sub_proc* parameter matches the explicit routine name from the external file. The **Declare Sub** statement can include an **Alias** clause, which lets you call the external routine by whatever name you choose. The **Alias** clause lets you override an external routine's explicit name, in situations where the explicit name conflicts with the name of a standard MapBasic function.

If the **Declare Sub** statement includes an **Alias** clause, the *sub_alias* parameter must match the external routine's original name, and the *sub_proc* parameter indicates the name by which MapBasic will call the routine. You can pass a custom variable type as a parameter to a DLL. However, the DLL must be compiled with “structure packing” set to the tightest packing. For information on custom variable types, see **Type statement**.

Example

```
Declare Sub Main
Declare Sub Cube(ByVal original As Float, cubed As Float)

Sub Main
   Dim x, result As Float
   Call Cube(2, result)
   ' result now contains the value: 8 (2 x 2 x 2)
   x = 1
   Call Cube(x + 2, result)
   ' result now contains the value: 27 (3 x 3 x 3)
End Sub
Sub Cube (ByVal original As Float, cubed As Float)
   '
   ' Cube the "original" parameter value, and store
   ' the result in the "cubed" parameter.
   '
   cubed = original ^ 3
End Sub
```

See Also:

Call statement, **Sub…End Sub statement**

Define statement

Purpose

Defines a custom keyword with a constant value.

Restrictions

You cannot issue a **Define** statement through the MapBasic window.

Syntax

Define *identifier definition*

identifier is an identifier up to 31 characters long, beginning with a letter or underscore (_).

definition is the text MapBasic should substitute for each occurrence of identifier.

Description

The **Define** statement defines a new identifier. For the remainder of the program, whenever MapBasic encounters the same identifier the original definition will be substituted for the identifier. For examples of **Define** statements, see the standard MapBasic definitions file, MAPBASIC.DEF.

An identifier defined through a **Define** statement is not case-sensitive. If you use a **Define** statement to define the keyword FOO, your program can refer to the identifier as Foo or foo. You cannot use the **Define** statement to re-define a MapBasic keyword, such as **Set** or **Create**. For a list of reserved keywords, see **Dim statement**.

Examples

Your application may need to reference the mathematical value known as Pi, which has a value of approximately 3.141593. Accordingly, you might want to use the following definition:

```
Define PI 3.141593
```

Following such a definition, you could simply type PI wherever you needed to reference the value 3.141593.

The definition portion of a **Define** statement can include quotes. For example, the following statement creates a keyword with a definition including quotes:

```
Define FILE_NAME "World.tab"
```

The following **Define** is part of the standard definitions file, MAPBASIC.DEF. This **Define** provides an easy way of clearing the Message window:

```
Define CLS Print Chr$(12)
```

DeformatNumber$( ) function

Purpose

Removes formatting from a string that represents a number. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

DeformatNumber$ (*numeric_string*)

numeric_string is a string that represents a numeric value, such as “12,345,678”.

Return Value

String

Description

Returns a string that represents a number. The return value does not include thousands separators, regardless of whether the *numeric_string* argument included comma separators. The return value uses a period as the decimal separator, regardless of whether the user's computer is set up to use another character as the decimal separator.

Examples

The following example calls the **Val( ) function** to determine the numeric value of a string. Before calling the **Val( ) function**, this example calls the **DeformatNumber$( ) function** to remove comma separators from the string. (The string that you pass to the **Val( ) function** cannot contain comma separators.)

```
Dim s_number As String
Dim f_value As Float

s_number = "1,222,333.4"
s_number = DeformatNumber$(s_number)

' the variable s_number now contains the
' string: "1222333.4"

f_value = Val(s_number)

Print f_value
```

See Also:

FormatNumber$( ) function, **Val( ) function**

Delete statement

Purpose

Deletes one or more graphic objects, or one or more entire rows, from a table. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Delete [ Object ] From table [ Where Rowid = id_number ]
```

table is the name of an open table.

id_number is the number of a single row (an integer value of one or more).

Description

The **Delete** statement deletes graphical objects or entire records from an open table.

By default, the **Delete** statement deletes all records from a table. However, if the statement includes the optional **Object** keyword, MapBasic only deletes the graphical objects that are attached to the table, rather than deleting the records themselves.

By default, the **Delete** statement affects all records in the table. However, if the statement includes the optional **Where Rowid =**…clause, then only the specified row is affected by the **Delete** statement.

There is an important difference between a **Delete Object From** statement and a **Drop Map statement**. A **Delete Object From** statement only affects objects or records in a table, it does not affect the table structure itself. A Drop Map statement actually modifies the table structure, so that graphical objects may not be attached to the table.

Examples

The following **Delete** statement deletes all of the records from a table. At the conclusion of this operation, the table still exists, but it is completely empty - as if the user had just created it by choosing **File > New**.

```
Open Table "clients"
Delete From clients
 Table clients
```

The following **Delete** statement deletes only the object from the tenth row of the table:

```
Open Table "clients"
Delete Object From clients Where Rowid = 10
 Table clients
```

See Also:

Drop Map statement, **Insert statement**

Dialog statement

Purpose

Displays a custom dialog box.

Restrictions

You cannot issue a **Dialog** statement through the MapBasic window.

Syntax

```
Dialog
   [ Title title ]
   [ Width w ] [ Height h ] [ Position x, y ]
   [ Calling handler ]
      Control control_clause
   [ Control control_clause... ]
```

title is a string expression that appears in the title bar of the dialog box.

h specifies the height of the dialog box, in dialog box units (8 dialog box height units represent the height of one character).

w specifies the width of the dialog, in dialog units (4 dialog height units represent the width of one character).

x, y specifies the dialog box's initial position, in pixels, representing distance from the upper-left corner of MapInfo Professional's work area; if the **Position** clause is omitted, the dialog box appears centered.

handler is the name of a procedure to call before the user is allowed to use the dialog box; this procedure is typically used to issue **Alter Control statement**s.

Each *control_clause* can specify one of the following types of controls:

- Button
- OKButton
- CancelButton
- EditText
- StaticText
- PopupMenu
- CheckBox
- MultiListBox
- GroupBox
- RadioGroup
- PenPickerm
- BrushPicker
- FontPicker
- SymbolPicker
- ListBox

See the separate discussions of those control types for more details (for example, for details on CheckBox controls, see **Control CheckBox clause**; for details on Picker controls, see **Control PenPicker/BrushPicker/SymbolPicker/FontPicker clause**; etc.).

Each control_clause can specify one of the following control types:

- Button / OKButton / CancelButton
- CheckBox
- GroupBox
- RadioGroup
- EditText
- StaticText
- PenPicker / BrushPicker / SymbolPicker / FontPicker
- ListBox / MultiListBox
- PopupMenu

Description

The **Dialog** statement creates a dialog box, displays it on the screen, and lets the user interact with it. The dialog box is modal; in other words, the user must dismiss the dialog box (for example, by clicking **OK** or **Cancel**) before doing anything else in MapInfo Professional. For an introduction to custom dialog boxes, see the *MapBasic User Guide*.

Anything that can appear on a dialog box is known as a control. Each dialog box must contain at least one control (for example, an OKButton control). Individual control clauses are discussed in separate entries (for example, see **Control CheckBox clause** for a discussion of check-box controls). As a general rule, every dialog box should include an OKButton control and/or a CancelButton control, so that the user has a way of dismissing the dialog box.

The **Dialog** statement lets you create a custom dialog box. If you want to display a standard dialog box (for example, a **File > Open** dialog box), use one of the following statements or functions: **Ask( ) function**, **Note statement**, **ProgressBar statement**, **FileOpenDlg( ) function**, **FileSaveAsDlg( ) function**, or **GetSeamlessSheet( ) function**).

For an introduction to the concepts behind MapBasic dialog boxes, see the *MapBasic User Guide*.

Sizes and Positions of Dialog Boxes and Dialog Box Controls

Within the **Dialog** statement, sizes and positions are stated in terms of dialog box units. A width of four dialog box units equals the width of one character, and a height of eight dialog box units equals the height of one character. Thus, if a dialog box control has a height of 40 and a width of 40, that control is roughly ten characters wide and 5 characters tall. Control positions are relative to the upper left corner of the dialog box. To place a control at the upper-left corner of a dialog box, use x- and y-coordinates of zero and zero.

The **Position**, **Height**, and **Width** clauses are all optional. If you omit these clauses, MapBasic places the controls at default positions in the dialog box, with subsequent control clauses appearing further down in the dialog box.

Terminating a Dialog Box

After a MapBasic program issues a **Dialog** statement, the user will continue interacting with the dialog box until one of four things happens:

- The user clicks the OKButton control (if the dialog box has one);
- The user clicks the CancelButton control (if the dialog box has one);
- The user clicks a control with a handler that issues a **Dialog Remove statement**; or
- The user otherwise dismisses the dialog box (for example, by pressing Esc on a dialog box that has a CancelButton).

To force a dialog box to remain on the screen after the user has clicked **OK** or **Cancel**, assign a handler procedure to the OKButton or CancelButton control and have that handler issue a **Dialog Preserve statement**.

Reading the User's Input

After a **Dialog** statement, call the **CommandInfo( ) function** to determine whether the user clicked **OK** or **Cancel** to dismiss the dialog box. If the user clicked **OK**, the following function call returns TRUE:

```
CommandInfo(CMD_INFO_DLG_OK)
```

There are two ways to read values entered by the user: Include Into clauses in the **Dialog** statement, or call the **ReadControlValue( ) function** from a handler procedure.

If a control specifies the **Into** clause, and if the user clicks the OKButton, MapInfo Professional stores the control's final value in a program variable.

MapInfo Professional only updates the variable if the user clicks **OK**. Also, MapInfo Professional only updates the variable after the dialog box terminates.

To read a control's value from within a handler procedure, call the **ReadControlValue( ) function**.

Specifying Hotkeys for Controls

When a MapBasic application runs on MapInfo, dialog boxes can assign hotkeys to the various controls. A hotkey is a convenience allowing the user to choose a dialog box control by pressing key sequences rather than clicking with the mouse.

To specify a hotkey for a control, include the ampersand character (&) in the title for that control. Within the **Title** clause, the ampersand should appear immediately before the character which is to be used as a hotkey character. Thus, the following Button clause defines a button which the user can choose by pressing Alt-R:

```
Control Button
   Title "&Reset"
```

Although an ampersand appears within the **Title** clause, the final dialog box does not show the ampersand. If you need to display an ampersand character in a control (for example, if you want a button to read "Find & Replace"), include two successive ampersand characters in the **Title** clause:

```
Title "Find && Replace"
```

If you position a StaticText control just before or above an EditText control, and you define the StaticText control with a hotkey designation, the user is able to jump to the EditText control by pressing the hotkey sequence.

Specifying the Tab Order

The user can press the Tab key to move the keyboard focus through the dialog box. The focus moves from control to control according to the dialog box's tab order.

Tab order is defined by the order of the **Control** clauses in the **Dialog** statement. When the focus is on the third control, pressing Tab moves the focus to the fourth control, etc. If you want to change the tab order, change the order of the **Control** clauses.

Examples

The following example creates a simple dialog box with an EditText control. In this example, none of the **Control** clauses use the optional **Position** clause; therefore, MapBasic places each control in a default position.

```
Dialog
   Title "Search"
   Control StaticText
      Title "Enter string to find:"
   Control EditText
      Value gs_searchfor 'this is a Global String variable
      Into gs_searchfor
   Control OKButton
   Control CancelButton
If CommandInfo(CMD_INFO_DLG_OK) Then
   ' ...then the user clicked OK, and the variable
   ' gs_searchfor contains the text the user entered.
End If
```

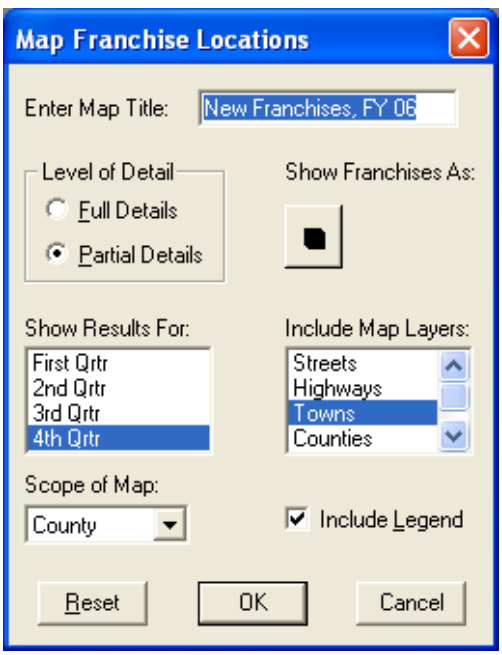
The following program demonstrates the syntax of all of MapBasic's control types.

```
Include "mapbasic.def"
Declare Sub reset_sub `resets dialog to default settings
Declare Sub ok_sub ' notes values when user clicks OK.
Declare Sub Main
Sub Main
   Dim s_title As String 'the title of the map
   Dim l_showlegend As Logical 'TRUE means include legend
   Dim i_details As SmallInt '1 = full details; 2 = partial
   Dim i_quarter As SmallInt '1=1st qrtr, etc.
   Dim i_scope As SmallInt '1=Town;2=County; etc.
   Dim sym_variable As Symbol

   Dialog
   Title "Map Franchise Locations"

   Control StaticText
      Title "Enter Map Title:"
      Position 5, 10
```

```
Control EditText
   Value "New Franchises, FY 95"
   Into s_title
   ID 1
   Position 65, 8 Width 90
Control GroupBox
   Title "Level of Detail"
   Position 5, 30 Width 70 Height 40

Control RadioGroup
   Title "&Full Details;&Partial Details"
   Value 2
   Into i_details
   ID 2
   Position 12, 42 Width 60

Control StaticText
   Title "Show Franchises As:" Position 95, 30
Control SymbolPicker
   Position 95, 45
   Into sym_variable
   ID 3

Control StaticText
   Title "Show Results For:"
   Position 5, 80
Control ListBox
   Title "First Qrtr;2nd Qrtr;3rd Qrtr;4th Qrtr"
   Value 4
   Into i_quarter
   ID 4
   Position 5, 90 Width 65 Height 35
Control StaticText
   Title "Include Map Layers:"
   Position 95, 80
Control MultiListBox
   Title "Streets;Highways;Towns;Counties;States"
   Value 3
   ID 5
   Position 95, 90 Width 65 Height 35
Control StaticText
   Title "Scope of Map:"
   Position 5, 130
Control PopupMenu
   Title "Town;County;Territory;Entire State"
   Value 2
   Into i_scope
   ID 6
   Position 5, 140
Control CheckBox
   Title "Include &Legend"
```

```
      Into l_showlegend
      ID 7
      Position 95, 140
   Control Button
      Title "&Reset"
      Calling reset_sub
      Position 10, 165
   Control OKButton
      Position 65, 165
      Calling ok_sub

   Control CancelButton
      Position 120, 165
   If CommandInfo(CMD_INFO_DLG_OK) Then
      ' ... then the user clicked OK.
   Else
      ' ... then the user clicked Cancel.
   End If
End Sub
Sub reset_sub
   ' here, you could use Alter Control statements
   ' to reset the controls to their original state.
End Sub
Sub ok_sub
   ' Here, place code to handle user clicking OK
End Sub
```

The preceding program produces the following dialog box.



See Also:

Alter Control statement, **Ask( ) function**, **Dialog Preserve statement**, **Dialog Remove statement**, **FileOpenDlg( ) function**, **FileSaveAsDlg( ) function**, **Note statement**, **ReadControlValue( ) function**

Dialog Preserve statement

Purpose

Reactivates a custom dialog box after the user clicked **OK** or **Cancel**.

Syntax

```
Dialog Preserve
```

Restrictions

This statement may only be issued from within a sub procedure that acts as a handler for an OKButton or CancelButton dialog box control. You cannot issue this statement from the MapBasic window.

Description

The **Dialog Preserve** statement allows the user to resume using a custom dialog box (which was created through a **Dialog statement**) even after the user clicked the OKButton or CancelButton control.

The **Dialog Preserve** statement lets you “confirm” the user's OK or Cancel action. For example, if the user clicks **Cancel**, you may wish to display a dialog box asking a question such as “Do you want to lose your changes?” If the user chooses “No” on the confirmation dialog box, the application should reactivate the original dialog box. You can provide this functionality by issuing a **Dialog Preserve** statement from within the CancelButton control's handler procedure.

Example

The following procedure could be used as a handler for a CancelButton control.

```
Sub confirm_cancel
   If Ask("Do you really want to lose your changes?",
      "Yes", "No") = FALSE Then
      Dialog Preserve
   End If
End Sub
```

See Also:

Alter Control statement, **Dialog statement**, **Dialog Remove statement**, **ReadControlValue( ) function**

Dialog Remove statement

Purpose

Removes a custom dialog from the screen.

Syntax

```
Dialog Remove
```

Restrictions

This statement may only be issued from within a sub procedure that acts as a handler for a dialog box control. You cannot issue this statement from the MapBasic window.

Description

The **Dialog Remove** statement removes the dialog box created by the most recent **Dialog statement**. A dialog box disappears automatically after the user clicks on an OKButton control or a CancelButton control. Use the **Dialog Remove** statement (within a dialog box control's handler routine) to remove the dialog box before the user clicks **OK** or **Cancel**. This is useful, for example, if you have a dialog box with a ListBox control, and you want the dialog box to come down if the user double-clicks an item in the list.

Dialog Remove signals to remove the dialog box after the handler sub procedure returns. It does not remove the dialog box instantaneously.

Example

The following procedure is part of the sample program NVIEWS.MB. It handles the ListBox control in the Named Views dialog box. When the user single-clicks a list item, this handler procedure enables various buttons on the dialog box. When the user double-clicks a list item, this handler uses a **Dialog Remove** statement to dismiss the dialog box.

MapInfo Professional calls this handler procedure for click events and for double-click events.

```
Sub listbox_handler
   Dim i As SmallInt
   Alter Control 2 Enable
   Alter Control 3 Enable
   If CommandInfo(CMD_INFO_DLG_DBL) = TRUE Then
      '
      ' ... then the user double-clicked.
      '
      i = ReadControlValue(1)
      Dialog Remove
      Call go_to_view(i)
```

```
   End If
End Sub
```

See Also:

Alter Control statement, **Dialog statement**, **Dialog Preserve statement**, **ReadControlValue( ) function**

Dim statement

Purpose

Defines one or more variables. You can issue this statement from the MapBasic Window in MapInfo Professional.

Restrictions

When you issue **Dim** statements through the MapBasic window, you can only define one variable per **Dim** statement, although a **Dim** statement within a compiled program may define multiple variables. You cannot define array variables using the MapBasic window.

Syntax

```
Dim var_name [ , var_name ... ] As var_type
   [ , var_name [ , var_name ... ] As var_type ... ]
```

var_name is the name of a variable to define.

var_type is a standard or custom variable Type.

Description

A **Dim** statement declares one or more variables. The following table summarizes the types of variables which you can declare through a **Dim** statement.

Location of Dim Statements and Scope of Variables

Variable Type	Description
SmallInt	Whole numbers from -32768 to 32767 (inclusive); stored in 2 bytes.
Integer	Whole numbers from -2,147,483,648 to +2,147,483,647 (inclusive); stored in 4 bytes.
Float	Floating point value; stored in eight-byte IEEE format.
String	Variable-length character string, up to 32768 bytes long.
String * length	Fixed-length character string (where *length* dictates the length of the string, in bytes, up to 32768 bytes); fixed-length strings are padded with trailing blanks.

Location of Dim Statements and Scope of Variables *(continued)*

Variable Type	Description
Logical	TRUE or FALSE, stored in 1 byte: zero=FALSE, non-zero=TRUE.
Date	Date, stored in four bytes: two bytes for the year, one byte for the month, one byte for the day.
DateTime	DateTime is stored in nine bytes: 4 bytes for date, 5 bytes for time. 5 bytes for time include: 2 for millisec, 1 for sec, 1 for min, 1 for hour.
Object	Graphical object (Point, Region, Line, Polyline, Arc, Rectangle, Rounded Rectangle, Ellipse, Text, or Frame).
Alias	Column name.
Pen	Pen (line) style setting.
Brush	Brush (fill) style setting.
Font	Font (text) style setting.
Symbol	Symbol (point-marker) style setting.

The **Dim** statement which defines a variable must precede any other statements which use that variable. **Dim** statements usually appear at the top of a procedure or function.

If a **Dim** statement appears within a **Sub…End Sub statement** or within a **Function…End Function statement**, the statement defines variables that are local in scope. Local variables may only be accessed from within the procedure or function that contained the **Dim** statement.

If a **Dim** statement appears outside of any procedure or function definition, the statement defines variables that are module-level in scope. Module-level variables can be accessed by any procedure or function within a program module (for example, within the .MB program file).

To declare global variables (variables that can be accessed by any procedure or function in any of the modules that make up a project), use the **Global statement**.

Declaring Multiple Variables and Variable Types

A single **Dim** statement can declare two or more variables that are separated by commas. You also can define variables of different types within one **Dim** statement by grouping like variables together, and separating the different groups with a comma after the variable type:

```
Dim jointer, i_min, i_max As Integer, s_name As String
```

Array Variables

MapBasic supports one-dimensional array variables. To define an array variable, add a pair of parentheses immediately after the variable name. To specify an initial array size, include a constant integer expression between the parentheses.

The following example declares an array of ten float variables, then assigns a value to the first element in the array:

```
Dim f_stats(10) As Float
f_stats(1) = 17.23
```

The number that appears between the parentheses is known as the subscript. The first element of the array is the element with a subscript of one (as shown in the example above).

To re-size an array, use the **ReDim statement**. To determine the current size of an array, use the **UBound( ) function**. If the **Dim** statement does not specify an initial array size, the array will initially contain no members; in such a case, you will not be able to store any data in the array until re-sizing the array with a **ReDim statement**. A MapBasic array can have up to 32,767 items.

String Variables

A string variable can contain a text string up to 32 kilobytes in length. However, there is a limit to how long a string constant you can specify in a simple assignment statement. The following example performs a simple string variable assignment, where a constant string expression is assigned to a string variable:

```
Dim status As String
status = "This is a string constant ... "
```

In this type of assignment, the constant string expression to the right of the equal sign has a maximum length of 256 characters.

MapBasic, like other BASIC languages, pads fixed-length string variables with blanks. In other words, if you define a 10-byte string variable, then assign a five-character string to that variable, the variable will actually be padded with five spaces so that it fills the space allotted. (This feature makes it easier to format text output in such a way that columns line up).

Variable-length string variables, however, are not padded in this fashion. This difference can affect comparisons of strings; you must exercise caution when comparing fixed-length and variable-length string variables. In the following program, the **If…Then statement** would determine that the two strings are not equal:

```
Dim s_var_len As String
Dim s_fixed_len As String * 10
s_var_len = "testing"
s_fixed_len = "testing"
If s_var_len = s_fixed_len Then
   Note "strings are equal" ' this won't happen
Else
   Note "strings are NOT equal" ' this WILL happen
End If
```

Restrictions on Variable Names

Variable names are case-insensitive. Thus, if a **Dim** statement defines a variable called abc, the program may refer to that variable as abc, ABC, or Abc.

Each variable name can be up to 31 characters long, and can include letters, numbers, and the underscore character (_). Variable names can also include the punctuation marks $, %, &, !, #, and @, but only as the final character in the name. A variable name may not begin with a number.

Many MapBasic language keywords, such as **Open**, **Close**, **Set**, and **Do**, are reserved words which may not be used as variable names. If you attempt to define a variable called **Set**, MapBasic will generate an error when you compile the program. The table below summarizes the MapBasic keywords which may not be used as variable names.

Add	**Alter**	**Browse**	**Call**
Close		**Create**	**DDE**
DDEExecute	**DDEPoke**	**DDETerminate**	**DDETerminateAll**
Declare	**Delete**	**Dialog**	**Dim**
Do	**Drop**	**Else**	**ElseIf**
End	**Error**	**Event**	**Exit**
Export	**Fetch**	**Find**	**For**
Function	**Get**	**Global**	**Goto**
Graph	**If**	**Import**	**Insert**
Layout	**Map**	**Menu**	**Note**
Objects	**OnError**	**Open**	**Pack**
Print	**PrintWin**	**ProgressBar**	**Put**
ReDim	**Register**	**Reload**	**Remove**
Rename	**Resume**	**Rollback**	**Run**
Save	**Seek**	**Select**	**Set**
Shade	**StatusBar**	**Stop**	**Sub**
Type	**Update**	**While**	

In some BASIC languages, you can dictate a variable's type by ending the variable with one of the punctuation marks listed above. For example, some BASIC languages assume that any variable named with a dollar sign (for example, LastName$) is a string variable. In MapBasic, however, you must declare every variable's type explicitly, through the **Dim** statement.

Initial Values of Variables

MapBasic initializes numeric variables to a value of zero when they are defined. Variable-length string variables are initialized to an empty string, and fixed-length string variables are initialized to all spaces.

Object and style variables are not automatically initialized. You must initialize Object and style variables before making references to those variables.

Example

```
' Below is a custom Type definition, which creates
' a new data type known as Person
Type Person
   Name As String
   Age As Integer
   Phone As String
End Type

' The next Dim statement creates a Person variable
Dim customer As Person

' This Dim creates an array of Person variables:
Dim users(10) As Person

' this Dim statement defines an integer variable
' "counter", and an integer array "counters" :
Dim counter, counters(10) As Integer

' the next statement assigns the "Name" element
' of the first member of the "users" array
users(1).Name = "Chris"
```

See Also:

Global statement, **ReDim statement**, **Type statement**, **UBound( ) function**

Distance( ) function

Purpose

Returns the distance between two locations. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Distance (*x1*, *y1*, *x2*, *y2*, *unit_name*)

x1 and *x2* are x-coordinates (for example, longitude).

y1 and *y2* are y-coordinates (for example, latitude).

unit_name is a string representing the name of a distance unit (for example, “km”).

Return Value

Float

Description

The **Distance( )** function calculates the distance between two locations.

The function returns the distance measurement in the units specified by the *unit_name* parameter; for example, to obtain a distance in miles, specify “mi” as the *unit_name* parameter. See **Set Distance Units statement** for the list of available unit names.

The x- and y-coordinate parameters must use MapBasic's current coordinate system. By default, MapInfo Professional expects coordinates to use a Longitude/Latitude coordinate system. You can reset MapBasic's coordinate system through the **Set CoordSys statement**.

If the current coordinate system is an earth coordinate system, **Distance( )** returns the great-circle distance between the two points. A great-circle distance is the shortest distance between two points on a sphere. (A great circle is a circle that goes around the earth, with the circle's center at the center of the earth; a great-circle distance between two points is the distance along the great circle which connects the two points.)

For the most part, MapInfo Professional performs a Cartesian or Spherical operation. Generally, a spherical operation is performed unless the coordinate system is NonEarth, in which case, a Cartesian operation is performed.

Example

```
Dim dist, start_x, start_y, end_x, end_y As Float
Open Table "cities"
Fetch First From cities
start_x = CentroidX(cities.obj)
start_y = CentroidY(cities.obj)
Fetch Next From cities
end_x = CentroidX(cities.obj)
end_y = CentroidY(cities.obj)
dist = Distance(start_x,start_y,end_x,end_y,"mi")
```

See Also:

Area( ) function, **ObjectLen( ) function**, **Set CoordSys statement**, **Set Distance Units statement**

Do Case…End Case statement

Purpose

Decides which group of statements to execute, based on the current value of an expression.

Restrictions

You cannot issue a **Do Case** statement through the MapBasic window.

Syntax

```
Do Case do_expr
    Case case_expr [ , case_expr ]
       statement_list
   [ Case ... ]
   [ Case Else
       statement_list ]
End Case
```

do_expr is an expression.

case_expr is an expression representing a possible value for *do_expr*.

statement_list is a group of statements to carry out under the appropriate circumstances.

Description

The **Do Case** statement is similar to the **If…Then statement**, in that **Do Case** tests for the existence of certain conditions, and decides which statements to execute (if any) based on the results of the test. MapBasic's **Do Case** statement is analogous to the BASIC language's Select Case statement. (In MapBasic, the name of the statement was changed to avoid conflicting with the **Select statement**).

In executing a **Do Case** statement, MapBasic examines the first **Case** *case_expr* clause. If one of the expressions in the **Case** *case_expr* clause is equal to the value of the *do_expr* expression, that case is considered a match. Accordingly, MapBasic executes the statements in that Case's *statement_list*, and then jumps down to the first statement following the **End Case** statement.

If none of the expressions in the first **Case** *case_expr* clause equal the *do_expr* expression, MapBasic tries to find a match in the following **Case** *case_expr* clause. MapBasic will test each **Case** *case_expr* clauses in succession, until one of the cases is a match or until all of the cases are exhausted.

MapBasic will execute at most one *statement_list* from a **Do Case** statement. Upon finding a matching Case, MapBasic will execute that Case's *statement_list*, and then jump immediately down to the first statement following **End Case**.

If none of the *case_expr* expressions are equal to the *do_expr* expression, none of the cases will match, and thus no *statement_list* will be executed. However, if a **Do Case** statement includes a **Case Else** clause, and if none of the **Case** *case_expr* clauses match, then MapBasic will carry out the statement list from the **Case Else** clause.

Note that a **Do Case** statement of this form:

```
Do Case expr1
   Case expr2
       statement_list1
   Case expr3, expr4
       statement_list2
   Case Else
       statement_list3
End Case
```

would have the same effect as an **If…Then statement** of this form:

```
If expr1 = expr2 Then
    statement_list1
ElseIf expr1 = expr3 Or expr1 = expr4 Then
    statement_list2
Else
    statement_list3
End If
```

Example

The following example builds a text string such as “First Quarter”, “Second Quarter”, etc., depending on the current date.

```
Dim cur_month As Integer, msg As String
cur_month = Month( CurDate( ) )
Do Case cur_month
   Case 1, 2, 3
      msg = "First Quarter"
   Case 4, 5, 6
      msg = "Second Quarter"
   Case 7, 8, 9
      msg = "Third Quarter"
   Case Else
      msg = "Fourth Quarter"
End Case
```

See Also:

If…Then statement

Do…Loop statement

Purpose

Defines a loop which will execute until a specified condition becomes TRUE (or FALSE).

Restrictions

You cannot issue a **Do Loop** statement through the MapBasic window.

Syntax 1

```
Do
    statement_list
Loop [ { Until | While } condition ]
```

Syntax 2

```
Do [ { Until | While } condition ]
    statement_list
Loop
```

statement_list is a group of statements to be executed zero or more times.

condition is a conditional expression which controls when the loop terminates.

Description

The **Do…Loop** statement provides loop control. Generally speaking, the **Do…Loop** repeatedly executes the statements in a *statement_list* as long as a **While** condition remains TRUE (or, conversely, the loop repeatedly executes the *statement_list* until the **Until** condition becomes TRUE).

If the **Do…Loop** does not contain the optional **Until** / **While** clause, the loop will repeat indefinitely. In such a case, a flow control statement, such as **Goto statement** or **Exit Do statement**, will be needed to halt or exit the loop. The **Exit Do statement** halts any **Do…Loop** immediately (regardless of whether the loop has an **Until** / **While** clause), and resumes program execution with the first statement following the **Loop** clause.

As indicated above, the optional **Until** / **While** clause may either follow the **Do** keyword or the **Loop** keyword. The position of the **Until** / **While** clause dictates whether MapBasic tests the condition before or after executing the *statement_list*. This is of particular importance during the first iteration of the loop. A loop using the following syntax:

```
Do
    statement_list
Loop While condition
```

will execute the *statement_list* and then test the condition. If the condition is TRUE, MapBasic will continue to execute the *statement_list* until the condition becomes FALSE. Thus, a **Do…Loop** using the above syntax will execute the *statement_list* at least once.

By contrast, a **Do…Loop** of the following form will only execute the *statement_list* if the condition is TRUE.

```
Do While condition
    statement_list
Loop
```

Example

The following example uses a **Do…Loop** statement to read the first ten records of a table.

```
Dim sum As Float, counter As Integer
Open Table "world"
Fetch First From world
counter = 1
Do
   sum = sum + world.population
   Fetch Next From world
   counter = counter + 1
Loop While counter <= 10
```

See Also:

Exit Do statement, **For…Next statement**

Drop Index statement

Purpose

Deletes an index from a table. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Drop Index *table*(*column*)

table is the name of an open table.

column is the name of a column in that table.

Description

The **Drop Index** statement deletes an existing index from an open table. Dropping an index reduces the amount of disk space occupied by a table. (To re-create that index at a later time, issue a **Create Index statement**.)

ⓘ MapInfo Professional cannot drop an index if the table has unsaved edits. Use the **Commit Table statement** to save edits.

The **Drop Index** statement takes effect immediately; no save operation is required. You cannot undo the effect of a **Drop Index** statement by selecting **File > Revert** or **Edit > Undo**. Similarly, the MapBasic **Rollback statement** will not undo the effect of a **Drop Index** statement.

Example

The following example deletes the index from the Name field of the World table.

```
Open Table "world"
Drop Index world(name)
```

See Also:

Create Index statement

Drop Map statement

Purpose

Deletes all graphical objects from a table. Cannot be used on linked tables. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Drop Map *table*

table is the name of an open table.

Description

A **Drop Map** statement deletes all graphical objects (points, lines, regions, circles, etc.) from an open table, and modifies the table structure so that graphical objects may not be attached to the table.

The **Drop Map** statement takes effect immediately; no save operation is required. You cannot undo the effect of a **Drop Map** statement by selecting **File > Revert** or **Edit > Undo**. Similarly, the MapBasic **Rollback statement** will not undo the effect of a **Drop Map** statement. Accordingly, you should be extremely cautious when using the **Drop Map** statement.

After performing a **Drop Map** operation, you will no longer be able to display the corresponding table in a Map window; the **Drop Map** statement modifies the table's structure so that objects may no longer be associated with the table. (A subsequent **Create Map statement** will restore the table's ability to contain graphical objects; however, a Create Map statement will not restore the graphical objects which were discarded during a **Drop Map** operation.) The **Drop Map** statement does not affect the number of records in a table. You still can browse a table after performing **Drop Map**.

If you wish to delete all of the graphical objects from a table, but you intend to attach new graphical objects to the same table, use **Delete Object** instead of **Drop Map**.

The **Drop Map** statement does not work on linked tables.

Example

```
Open Table "clients"
Drop Map clients
```

See Also:

Create Map statement, **Create Table statement**, **Delete statement**

Drop Table statement

Purpose

Deletes a table in its entirety. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

`Drop Table` *`table`*

table is the name of an open table.

Description

The **Drop Table** statement completely erases the specified table from the computer's disk. The table must already be open.

Note that if a table is based on a pre-existing database or spreadsheet file, the **Drop Table** statement will delete the original file as well as the component files which make it a table. In other words, a **Drop Table** operation may have the effect of deleting a file which is used outside of MapInfo Professional.

The **Drop Table** statement takes effect immediately; no save operation is required. You cannot undo the effect of a **Drop Table** statement by selecting **File > Revert** or **Edit > Undo**. Similarly, the MapBasic **Rollback statement** will not undo the effect of a **Drop Table** statement. You should be extremely cautious when using the **Drop Table** statement.

Many MapInfo table operations (for example, Select) store results in temporary tables (for example, Query1). Temporary tables are deleted automatically when you exit MapInfo Professional; you do not need to use the **Drop Table** statement to delete temporary tables.

The **Drop Table** statement cannot be used to delete a table that is actually a “view.” For example, a StreetInfo table (such as SF_STRTS) is actually a view, combining two other tables (SF_STRT1 and SF_STRT2). So, you could not delete the SF_STRTS table by using the **Drop Table** statement.

Example

```
Open Table "clients"
Drop Table clients
```

See Also:

Create Table statement, **Delete statement**, **Kill statement**

End MapInfo statement

Purpose

This statement halts MapInfo Professional.

Syntax

```
End MapInfo [ Interactive ]
```

Description

The **End MapInfo** statement halts MapInfo Professional.

An application can define a special procedure called **EndHandler**, which is executed automatically when MapInfo Professional terminates. Accordingly, when an application issues an **End MapInfo** statement, MapInfo Professional automatically executes any sleeping **EndHandler** procedures before shutting down. See **EndHandler procedure** for more information.

If an application issues an **End MapInfo** statement, and one or more tables have unsaved edits, MapInfo Professional prompts the user to save or discard the table edits.

If you include the **Interactive** keyword, and if there are unsaved themes or labels, MapInfo Professional prompts the user to save or discard the unsaved work. However, if the user's system is set up so that it automatically saves MAPINFOW.WOR on exit, this prompt does not appear. If you omit the **Interactive** keyword, this prompt does not appear.

To halt a MapBasic application without exiting MapInfo Professional, use the **End Program statement**.

See Also:

End Program statement, **EndHandler procedure**

End Program statement

Purpose

Halts a MapBasic application.

Restrictions

The **End Program** statement may not be issued from the MapBasic window.

Syntax

```
End Program
```

Description

The **End Program** statement halts execution of a MapBasic program. A MapBasic application can add items to MapInfo Professional menus, and even add entirely new menus to the menu bar. Typically, a menu item added in this fashion calls a sub procedure from a MapBasic program. Once a MapBasic application has connected a procedure to the menu in this fashion, the application is said to be “sleeping.”

If any procedure in a MapBasic application issues an **End Program** statement, that entire application is halted—even if “sleeping” procedures have been attached to custom menu items. When an application halts, MapInfo Professional automatically removes any menu items created by that application.

If an application defines a procedure named **EndHandler**, MapBasic automatically calls that procedure when the application halts, for whatever reason the application halts.

See Also:

End MapInfo statement, **EndHandler procedure**

EndHandler procedure

Purpose

A reserved procedure name, called automatically when an application terminates.

Syntax

```
Declare Sub EndHandler

Sub EndHandler
    statement_list
End Sub
```

statement_list is a list of statements to execute when the application terminates.

Description

EndHandler is a special-purpose MapBasic procedure name.

If the user runs an application containing a sub procedure named **EndHandler**, the **EndHandler** procedure is called automatically when the application ends. This happens whether the user exited MapInfo Professional or another procedure in the application issued an **End Program statement**.

Multiple MapBasic applications can be “sleeping” at the same time. When MapInfo Professional terminates, MapBasic automatically calls all sleeping **EndHandler** procedures, one after another.

See Also:

RemoteMsgHandler procedure, **SelChangedHandler procedure**, **ToolHandler procedure**, **WinChangedHandler procedure**, **WinClosedHandler procedure**

EOF( ) function

Purpose

Returns TRUE if MapBasic tried to read past the end of a file, FALSE otherwise.

Syntax

```
EOF( filenum )
```

filenum is the number of a file opened through the **Open File statement**.

Return Value

Logical

Description

The **EOF( )** function returns a logical value indicating whether the End-Of-File condition exists for the specified file. The integer filenum parameter represents the number of an open file.

If a **Get statement** tries to read past the end of the specified file, the **EOF( )** function returns a value of TRUE; otherwise, **EOF( )** returns a value of FALSE.

The **EOF( )** function works with open files; when you wish to check the current position of an open table, use the **EOT( ) function**.

For an example of calling **EOF( )**, see the sample program NVIEWS.MB (Named Views).

Error Conditions

ERR_FILEMGR_NOTOPEN (366) error generated if the specified file is not open.

See Also:

EOT( ) function, **Open File statement**

EOT( ) function

Purpose

Returns TRUE if MapBasic has reached the end of the specified table, FALSE otherwise. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

EOT(*table*)

table is the name of an open table.

Return Value

Logical

Description

The **EOT( )** function returns TRUE or FALSE to indicate whether MapInfo Professional has tried to read past the end of the specified table. The table parameter represents the name of an open table.

Error Conditions

ERR_TABLE_NOT_FOUND (405) error generated if the specified table is not available

Example

The following example uses the logical result of the **EOT( )** function to decide when to terminate a loop. The loop repeatedly fetches the next record in a table, until the point when the **EOT( )** function indicates that the program has reached the end of the table.

```
Dim f_total As Float
Open Table "customer"
Fetch First From customer
Do While Not EOT(customer)
   f_total = f_total + customer.order
   Fetch Next From customer
Loop
```

See Also:

EOF( ) function, **Fetch statement**, **Open File statement**, **Open Table statement**

EPSGToCoordSysString$( ) function

Purpose

Converts a string containing a Spatial Reference System into a **CoordSys clause** that can be used with any MapBasic function or statement. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

EPSGToCoordSysString$(*epsg_string*)

epsg_string is a String describing a Spatial Reference System (SRS) for any supported coordinate systems. SRS strings are also referred to as EPSG (European Petroleum Survey Group) strings (for example, epsg:2600). For a complete list of EPSG codes used with MapInfo Professional see the MAPINFOW.PRJ file in your MapInfo Professional installation. The EPSG codes are identified by a "\p" followed by a number.

Description

The **EPSGToCoordSysString$( )** is used to convert a SRS String into a CoordSys clause that can be used in any MapBasic function or statement that takes a CoordSys clause as an input.

Example

The following example sets the coordinate system of a map to Earth Projection 1, 104.

```
run command("Set Map " +  EPSGToCoordSysString$("EPSG:4326"))
```

See Also:

CoordSys clause

Erase( ) function

Purpose

Returns an object created by erasing part of another object. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Erase(*source_object*, *eraser_object*)

source_object is an object, part of which is to be erased; cannot be a point or text object.

eraser_object is a closed object, representing the area that will be erased.

Return Value

Returns an object representing what remains of *source_object* after erasing *eraser_object*.

Description

The **Erase( )** function erases part of an object, and returns an object expression representing what remains of the object.

The *source_object* parameter can be a linear object (line, polyline, or arc) or a closed object (region, rectangle, rounded rectangle, or ellipse), but cannot be a point object or text object. The *eraser_object* must be a closed object. The object returned retains the color and pattern styles of the *source_object*.

Example

```
' In this example, o1 and o2 are Object variables
' that already contain Object expressions.
If o1 Intersects o2 Then
   If o1 Entirely Within o2 Then
      Note "Cannot Erase; nothing would remain."
   Else
      o3 = Erase( o1, o2 )
   End If
Else
   Note "Cannot Erase; objects do not intersect."
End If
```

See Also:

Objects Erase statement, **Objects Intersect statement**

Err( ) function

Purpose

Returns a numeric code, representing the current error. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Err( )

Return Value

Integer

Description

The **Err( )** function returns the numeric code indicating which error occurred most recently.

By default, a MapBasic program which generates an error will display an error message and then halt. However, by issuing an **OnError statement**, a program can set up an error handling routine to respond to error conditions. Once an error handling routine is specified, MapBasic jumps to that routine automatically in the event of an error. The error handling routine can then call the **Err( )** function to determine which error occurred.

The **Err( )** function can only return error codes while within the error handler. Once the program issues a **Resume statement** to return from the error handling routine, the error condition is reset. This means that if you call the **Err( )** function outside of the error handling routine, it returns zero.

Some statement and function descriptions within this document contain an Error Conditions heading (just before the Example heading), listing error codes related to that statement or function. However, not all error codes are identified in the Error Conditions heading.

Some MapBasic error codes are only generated under narrowly-defined, specific circumstances; for example, the ERR_INVALID_CHANNEL (696) error is only generated by DDE-related functions or statements. If a statement might generate such an “unusual” error, the discussion for that statement will identify the error under the Error Conditions heading.

However, other MapBasic errors are “generic”, and might be generated under a variety of broadly-defined circumstances. For example, many functions, such as **Area( ) function** and **ObjectInfo( ) function**, take an Object expression as a parameter. Any such function will generate the ERR_FCN_OBJ_FETCH_FAILED (650) error if you pass an expression of the form *tablename*.obj as a parameter, when the current row from that table has no associated object. In other words, any function which takes an Object parameter might generate the ERR_FCN_OBJ_FETCH_FAILED (650) error. Since the ERR_FCN_OBJ_FETCH_FAILED (650) error can occur in so many different places, individual functions do not explicitly identify the error.

Similarly, there are two math errors—ERR_FP_MATH_LIB_DOMAIN (911) and ERR_FP_MATH_LIB_RANGE (912)—which can occur as a result of an invalid numeric parameter. These errors might be generated by calls to any of the following functions: **Acos( ) function**, **Asin( ) function**, **Atn( ) function**, **Cos( ) function**, **Exp( ) function**, **Log( ) function**, **Sin( ) function**, **Sqr( ) function**, or **Tan( ) function**.

The complete list of potential MapBasic error codes is included in the file ERRORS.DOC.

See Also:

Error statement, **Error$( ) function**, **OnError statement**

Error statement

Purpose

Simulates the occurrence of an error condition. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Error *error_num*

error_num is an integer error number.

Description

The **Error** statement simulates the occurrence of an error.

If an error-handling routine has been enabled through an **OnError statement**, the simulated error will cause MapBasic to perform the appropriate error-handling routine. If no error handling routine has been enabled, the error simulated by the **Error** statement will cause the MapBasic application to halt after displaying an appropriate error message.

See Also:

Err( ) function, **Error$( ) function**, **OnError statement**

Error$( ) function

Purpose

Returns a message describing the current error. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Error$( )

Return Value

String

Description

The **Error$( )** function returns a character string describing the current run-time error, if an error has occurred. If no error has occurred, the **Error$( )** function returns a null string.

The **Error$( )** function should only be called from within an error handling routine. See **Err( ) function** for more information.

See Also:

Err( ) function, **Error statement**, **OnError statement**

Exit Do statement

Purpose

Exits a **Do…Loop statement** prematurely.

Restrictions

You cannot issue an **Exit Do** statement through the MapBasic window.

Syntax

```
Exit Do
```

Description

An **Exit Do** statement terminates a **Do…Loop statement**. Upon encountering an **Exit Do** statement, MapBasic will jump to the first statement following the **Do…Loop statement**. Note that the **Exit Do** statement is only valid within a **Do…Loop statement**.

Do…Loop statements can be nested; that is, a **Do…Loop statement** can appear within the body of another, "outer" **Do…Loop statement**. An **Exit Do** statement only halts the iteration of the nearest **Do…Loop statement**. Thus, in an arrangement of this sort:

```
Do While condition1
:
   Do While condition2
      :
      If error_condition
         Exit Do
      End If
      :
   Loop
   :
Loop
```

the **Exit Do** statement will halt the inner loop (**Do While** *condition2*) without necessarily affecting the outer loop (**Do While** *condition1*).

See Also:

Do…Loop statement, **Exit For statement**, **Exit Sub statement**

Exit For statement

Purpose

Exits a **For…Next statement** prematurely.

Restrictions

You cannot issue an **Exit For** statement through the MapBasic window.

Syntax

```
Exit For
```

Description

An **Exit For** statement terminates a **For…Next statement**. Upon encountering an **Exit For** statement, MapBasic will jump to the first statement following the **For…Next statement**. Note that the **Exit For** statement is only valid within a **For…Next statement**.

For…Next statements can be nested; that is, a **For…Next statement** can appear within the body of another, "outer" **For…Next statement**. Note that an **Exit For** statement only halts the iteration of the nearest **For…Next statement**. Thus, in an arrangement of this sort:

```
For x = 1 to 5
   :
   For y = 2 to 10 step 2
      :
      If error_condition
         Exit For
      End If
      :
   Next
   :
Next
```

the **Exit For** statement will halt the inner loop (`For y = 2 to 10 step 2`) without necessarily affecting the outer loop (`For x = 1 to 5`).

See Also:

Exit Do statement, **For…Next statement**

Exit Function statement

Purpose

Exits a **Function…End Function statement**.

Restrictions

You cannot issue an **Exit Function** statement through the MapBasic window.

Syntax

```
Exit Function
```

Description

An **Exit Function** statement causes MapBasic to exit the current function. Accordingly, an **Exit Function** statement may only be issued from within a **Function…End Function statement**.

Function calls may be nested; in other words, one function can call another function, which, in turn, can call yet another function. Note that a single **Exit Function** statement exits only the current function.

See Also:

Function…End Function statement

Exit Sub statement

Purpose

Exits a **Sub…End Sub statement**.

Restrictions

You cannot issue an **Exit Sub** statement through the MapBasic window.

Syntax

```
Exit Sub
```

Description

An **Exit Sub** statement causes MapBasic to exit the current sub procedure. Accordingly, an **Exit Sub** statement may only be issued from within a sub procedure.

Sub…End Sub statement may be nested; in other words, one sub procedure can call another sub procedure, which, in turn, can call yet another sub procedure, etc. Note that a single **Exit Sub** statement exits only the current sub procedure.

See Also:

Call statement, **Sub…End Sub statement**

Exp( ) function

Purpose

Returns the number e raised to a specified exponent. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Exp(*num_expr*)

num_expr is a numeric expression.

Return Value

Float

Description

The **Exp( )** function raises the mathematical value e to the power represented by *num_expr*. e has a value of approximately 2.7182818.

MapBasic supports general exponentiation through the caret operator (^).

Example

```
Dim e As Float
e = Exp(1)
' the local variable e now contains
' approximately 2.7182818
```

See Also:

Cos( ) function, **Sin( ) function**, **Log( ) function**

Export statement

Purpose

Exports a table to another file format. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax 1 (for exporting MIF/MID files, DBF files, or ASCII text files)

```
Export table
   Into file_name
   [ Type
      { "MIF" |
      "DBF" [ Charset char_set ] |
```

```
      "ASCII" [ Charset char_set ] [ Delimiter "d " ] [ Titles ] |
      "CSV" [ Charset char_set ] [ Titles ] } ]
   [ Overwrite ]
```

Syntax 2 (for exporting DXF files)

```
Export table
   Into file_name
   [ Type "DXF" ]
   [ Overwrite ]
   [ Preserve
      [ AttributeData ] [ Preserve ] [ MultiPolygonRgns [ As Blocks ] ] ]
   [ { Binary | ASCII [ DecimalPlaces decimal_places ] } ]
   [ Version { 12 | 13 } ]
   [ Transform
      ( MI_x1, MI_y1 ) ( MI_x2, MI_y2 )
      ( DXF_x1, DXF_y1 ) ( DXF_x2, DXF_y2 ) ]
```

table is the name of an open table; do not use quotation marks around this name.

file_name is a string specifying the file name to contain the exported data; if the file name does not include a path, the export file is created in the current working directory.

char_set is a string that identifies a character set, such as “WindowsLatin1”; see **CharSet clause** for details.

d is a character used as a delimiter when exporting an ASCII file.

decimal_places is a small integer (from 0 to 16, default value is 6), which controls the number of decimal places used when exporting floating-point numbers in ASCII.

MI_x1, *MI_y1*, etc. are numbers that represent bounds coordinates in the MapInfo Professional table.

DXF_x1, *DXF_y1*, etc. are numbers that represent bounds coordinates in the DXF file.

Description

The **Export** statement copies the contents of a MapInfo table to a separate file, using a file format which other packages could then edit or import. For example, you could export the contents of a table to a DXF file, then use a CAD software package to import the DXF file. The **Export** statement does not alter the original table.

Specifying the File Format

The optional **Type** clause specifies the format of the file you want to create.

Type clause	File Format Specified
Type “MIF”	MapInfo Interchange File format. For information on the MIF file format, see the MapInfo Professional documentation.
Type “DXF”	DXF file (a format supported by CAD packages, such as AutoCAD).

Type clause	File Format Specified
Type “DBF”	dBASE file format. ⓘ Map objects are not exported when you specify DBF format.
Type “ASCII”	Text file format. ⓘ Map objects are not exported when you specify ASCII format.
Type “CSV”	Comma-delimited text file format. ⓘ Map objects are not exported when you specify CSV format.

If you omit the **Type** clause, MapInfo Professional assumes that the file extension indicates the desired file format. For example, if you specify the file name “PARCELS.DXF” MapInfo Professional creates a DXF file.

If you include the optional **Overwrite** keyword, MapInfo Professional creates the export file, regardless of whether a file by that name already exists. If you omit the **Overwrite** keyword, and the file already exists, MapInfo Professional does not overwrite the file.

Exporting ASCII Text Files

When you export a table to an ASCII or CSV text file, the text file will contain delimiters. A delimiter is a special character that separates the fields within each row of data. CSV text files automatically use a comma (`,`) as the delimiter. No other delimiter can be specified for CSV export.

The default delimiter for an ASCII text file is the TAB character (`Chr$(9)`). To specify a different delimiter, include the optional **Delimiter** clause. The following example uses a colon (`:`) as the delimiter:

```
Export sites Into "sitedata.txt" Type "ASCII"
   Delimiter ":" Titles
```

When you export to an ASCII or CSV text file, you may want to include the optional **Titles** keyword. If you include **Titles**, the first row of the text file will contain the table's column names. If you omit **Titles**, the column names will not be stored in the text file (which could be a problem if you intend to re-import the file later).

Exporting DXF Files

If you export a table into DXF file, using Syntax 2 as shown above, the **Export** statement can include the following DXF-specific clauses:

Include the **Preserve AttributeData** clause if you want to export the table's tabular data as attribute data in the DXF file.

Include the **Preserve MultiPolygonRgns As Blocks** clause if you want MapInfo Professional to export each multiple-polygon region as a DXF block entity. If you omit this clause, each polygon from a multiple-polygon region is stored separately.

Include the **Binary** keyword to export into a binary DXF file; or, include the **ASCII** keyword to export into an ASCII text DXF file. If you do not include either keyword, MapInfo Professional creates an ASCII DXF file. Binary DXF files are generally smaller, and can be processed much faster than ASCII. When you export as ASCII, you can specify the number of decimal places used to store floating-point numbers (0 to 16 decimal places; 6 is the default).

The **Version 12** or **Version 13** clause controls whether MapInfo Professional creates a DXF file compliant with AutoCAD 12 or 13. If you omit the clause, MapInfo Professional creates a version 12 DXF file.

Transform specifies a coordinate transformation. In the **Transform** clause, you specify the minimum and maximum x- and y-bounds coordinates of the MapInfo table, and then specify the minimum and maximum coordinates that you want to have in the DXF file.

Example

The following example takes an existing MapInfo table, Facility, and exports the table to a DXF file called “FACIL.DXF”.

```
Open Table "facility"

Export facility
   Into "FACIL.DXF"
   Type "DXF"
   Overwrite
   Preserve AttributeData
   Preserve MultiPolygonRgns As Blocks
   ASCII DecimalPlaces 3
   Transform (0, 0) (1, 1) (0, 0) (1, 1)
```

See Also:

Import statement

ExtractNodes( ) function

Purpose

Returns a polyline or region created from a subset of the nodes in an existing object. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

ExtractNodes(*object*, *polygon_index*, *begin_node*, *end_node*, *b_region*)

object is a polyline or region object.

polygon_index is an integer value, 1 or larger: for region objects. This indicates which polygon (for regions) or section (for polylines) to query.

begin_node is a SmallInt node number, 1 or larger; indicates the beginning of the range of nodes to return.

end_node is a SmallInt node number, 1 or larger; indicates the end of the range of nodes to return.

b_region is a logical value that controls whether a region or polyline object is returned; use TRUE for a region object or FALSE for a polyline object.

Return Value

Returns an object with the specified nodes. MapBasic applies all styles (color, etc.) of the original object; then, if necessary, MapBasic applies the current drawing styles.

Description

If the *begin_node* is equal to or greater than *end_node*, the nodes are returned in the following order:

- *begin_node* through the next-to-last node in the polygon;
- First node in polygon through *end_node*.

If *object* is a region object, and if *begin_node* and *end_node* are both equal to 1, MapBasic returns the entire set of nodes for that polygon. This provides a simple mechanism for extracting a single polygon from a multiple-polygon region. To determine the number of polygons in a region, call the **ObjectInfo( ) function**.

Error Conditions

ERR_FCN_ARG_RANGE (644) error generated if *b_region* is FALSE and the range of nodes contains fewer than two nodes, or if *b_region* is TRUE and the range of nodes contains fewer than three nodes.

See Also:

ObjectNodeX( ) function, **ObjectNodeY( ) function**

Farthest statement

Purpose

Find the object in a table that is farthest from a particular object. The result is a two-point Polyline object representing the farthest distance. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Farthest [ N | All ] From { Table fromtable | Variable fromvar }
   To totable Into intotable
   [ Type { Spherical | Cartesian }]
```

```
[ Ignore [ Contains ][ Min min_value ][ Max max_value ] Units unitname]
[ Data clause ]
```

N is an optional parameter for the number of “farthest” objects to find. The default is 1. If **All** is used, then a distance object is created for every combination.

fromtable is a table of objects from which you want to find farthest distances.

fromvar is a MapBasic variable representing an object that you want to find the farthest distances from.

totable is a table of objects that you want to find farthest distances to.

intotable is a table to place the results into.

min_value is the minimum distance to include in the results.

max_value is the maximum distance to include in the results.

unitname is string representing the name of a distance unit (for example, “km”) used for *min_value* and/or *max_value*.

Description

The **Farthest** statement finds all the objects in the *fromtable* that is furthest from a particular object. Every object in the *fromtable* is considered. For each object in the *fromtable*, the furthest object in the *totable* is found. If *N* is defined, then the *N* farthest objects in the *totable* are found. A two-point Polyline object representing the farthest points between the *fromtable* object and the chosen *totable* object is placed in the *intotable*. If **All** is specified, then an object is placed in the *intotable* representing the distance between the *fromtable* object and each *totable* object.

If there are multiple objects in the *totable* that are the same distance from a given *fromtable* object, then only one of them may be returned. If multiple objects are requested (for example, if *N* is greater than 1), then objects of the same distance will fill subsequent slots. If a tie exists at the second farthest object, and three objects are requested, then one of the second farthest objects will become the third farthest object.

The types of the objects in the *fromtable* and *totable* can be anything except Text objects. For example, if both tables contain Region objects, then the minimum distance between Region objects is found, and the two-point Polyline object produced represents the points on each object used to calculate that distance. If the Region objects intersect, then the minimum distance is zero, and the two-point Polyline returned will be degenerate, where both points are identical and represent a point of intersection.

The distances calculated do not take into account any road route distance. It is strictly a “as the bird flies” distance.

The **Ignore** clause can be used to limit the distances to be searched, and can effect how many totable *objects* are found for each *fromtable* object. One use of the **Min** distance could be to eliminate distances of zero. This may be useful in the case of two point tables to eliminate comparisons of the same point. For example, if there are two point tables representing Cities, and we want to find the closest cities, we may want to exclude cases of the same city.

The **Max** distance can be used to limit the objects to consider in the *totable*. This may be most useful in conjunction with *N* or **All**. For example, we may want to search for the five airports that are closest to a set of cities (where the *fromtable* is the set of cities and the *totable* is a set of airports), but we don't care about airports that are farther away than 100 miles. This may result in less than five airports being returned for a given city. This could also be used in conjunction with the **All** parameter, where we would find all airports within 100 miles of a city.

Supplying a **Max** parameter can improve the performance of the **Farthest** statement, since it effectively limits the number of *totable* objects that are searched.

The effective distances found are strictly greater than the *min_value* and less than or equal to the *max_value*:

```
min_value < distance <= max_value
```

This can allow ranges or distances to be returned in multiple passes using the **Farthest** statement. For example, the first pass may return all objects between 0 and 100 miles, and the second pass may return all objects between 100 and 200 miles, and the results should not contain duplicates (for example, a distance of 100 should only occur in the first pass and never in the second pass).

Type is the method used to calculate the distances between objects. It can either be Spherical or Cartesian. The type of distance calculation must be correct for the coordinate system of the *intotable* or an error will occur. If the Coordsys of the *intotable* is NonEarth and the distance method is Spherical, then an error will occur. If the Coordsys of the *intotable* is Latitude/Longitude, and the distance method is Cartesian, then an error will occur.

The **Ignore** clause limits the distances returned. Any distances found which are less than or equal to *min_value* or greater than *max_value* are ignored. *min_value* and *max_value* are in the distance unit signified by *unitname*. If *unitname* is not a valid distance unit, an error will occur. See **Set Distance Units statement** for the list of available unit names. The entire **Ignore** clause is optional, as are the **Min** and **Max** sub clauses within it.

Normally, if one object is contained within another object, the distance between the objects is zero. For example, if *fromtable* is WorldCaps and *totable* is World, then the distance between London and the United Kingdom would be zero. If the **Contains** keyword is used within the **Ignore** clause, then the distance will not be automatically be zero. Instead, the distance from London to the boundary of the United Kingdom will be returned. In effect, this will treat all closed objects, such as regions, as polylines for the purpose of this operation.

The **Data** clause can be used to mark which *fromtable* object and which *totable* object the result came from.

Data Clause

Data *IntoColumn1=column1, IntoColumn2=column2*

The *IntoColumn* on the left hand side of the equals sign must be a valid column in *intotable*. The *column* name on the right hand side of the equals sign must be a valid column name from either *totable* or *fromtable*. If the same column name exists in both *totable* and *fromtable*, then the column in *totable* will be used (e.g., *totable* is searched first for column names on the right hand side of the equals sign).

To avoid any conflicts such as this, the column names can be qualified using the table alias. For example:

```
Data name1=states.state_name, name2=county.state_name
```

To fill a column in the *intotable* with the distance, we can either use the **Table > Update Column** functionality from the menu or use the **Update statement**.

See Also:

Nearest statement, **CartesianObjectDistance( ) function**, **ObjectDistance( ) function**, **SphericalObjectDistance( ) function**, **CartesianConnectObjects( ) function**, **ConnectObjects( ) function**, **SphericalConnectObjects( ) function**

Fetch statement

Purpose

Sets a table's cursor position (for example, which row is the current row). You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Fetch { **First** | **Last** | **Next** | **Prev** | **Rec** *n* } **From** *table*

n is the number of the record to read.

table is the name of an open table.

Description

Use the **Fetch** statement to retrieve records from an open table. By issuing a **Fetch** statement, your program places the table cursor at a certain row position in the table; this dictates which of the records in the table is the “current” record.

The term “cursor” is used here to signify a row's position in a table. This has nothing to do with the on-screen mouse cursor.

After you issue a **Fetch** statement, you can retrieve data from the current row by using one of the following expression types:

Syntax	Example
table.column	`World.Country`
table.col#	`World.col1`
table.col(*number*)	`World.col( 1 )`

A **Fetch First** statement positions the cursor at the first un-deleted row in the table.

A **Fetch Last** statement positions the cursor at the last un-deleted row in the table.

A **Fetch Next** statement moves the cursor forward to the next un-deleted row.

A **Fetch Prev** statement moves the cursor backward to the previous un-deleted row.

A **Fetch Rec** *n* statement positions the cursor on a specific row, even if that row is deleted.

If the specified record is deleted, the statement generates run-time error 404.

Various MapInfo Professional and MapBasic operations (for example, Select, Update, and screen redraws) automatically reset the current row. Accordingly, **Fetch** statements should be issued just before any statements that make assumptions about which row is current.

Reading Past the End of the Table

After you issue a **Fetch** statement, you may need to call the **EOT( ) function** to determine whether you fetched an actual row.

If the **Fetch** statement placed the cursor on an actual row, the **EOT( ) function** returns FALSE (meaning, there is not an end-of-table condition).

If the **Fetch** statement attempted to place the cursor past the last row, the **EOT( ) function** returns TRUE (meaning, there is an end-of-table condition; therefore there is no “current row”).

The following example shows how to use a **Fetch Next** statement to loop through all rows in a table. As soon as a **Fetch Next** statement attempts to read past the final row, the **EOT( ) function** returns TRUE, causing the loop to halt.

```
Dim i As Integer

   i = 0
   Fetch First From world
   Do While Not EOT(world)
      i = i + 1
      Fetch Next From world
   Loop

   Print "Number of undeleted records: " + i
```

Examples

The following example shows how to fetch the 3rd record from the table States:

```
Open Table "states"
Fetch Rec 3 From states 'position at 3rd record
Note states.state_name 'display name of state
```

As illustrated in the example below, the **Fetch** statement can operate on a temporary table (for example, Selection).

```
Select * From states Where pop_1990 < pop_1980
Fetch First From Selection
Note Selection.col1 + " has negative net migration"
```

See Also:

EOT( ) function, **Open Table statement**

FileAttr( ) function

Purpose

Returns information about an open file. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

FileAttr(*filenum*, *attribute*)

filenum is the number of a file opened through an Open File statement.

attribute is a code indicating which file attribute to return; see table below.

Return Value

Integer

Description

The **FileAttr( )** function returns information about an open file. The *attribute* parameter must be one of the codes in this table:

attribute parameter	ID	Return Value
FILE_ATTR_MODE	1	Small integer, indicating the mode in which the file was opened. Return value will be one of the following: • MODE_INPUT (0) • MODE_OUTPUT (1) • MODE_APPEND (2) • MODE_RANDOM (3) • MODE_BINARY (4)
FILE_ATTR_FILESIZE	2	Integer, indicating the file size in bytes.

Error Conditions

ERR_FILEMGR_NOTOPEN (366) error is generated if the specified file is not open.

See Also:

EOF( ) function, **Get statement**, **Open File statement**, **Put statement**

FileExists( ) function

Purpose

Returns a logical value indicating whether or not a file exists. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

FileExists(*filespec*)

filespec is a string that specifies the file path and name.

Return Value

Logical: TRUE if the file already exists, otherwise FALSE.

Example

```
If FileExists("C:\MapInfo\TODO.TXT") Then

   Open File "C:\MapInfo\TODO.TXT" For INPUT As #1

End If
```

See Also:

TempFileName$( ) function

FileOpenDlg( ) function

Purpose

Displays a File Open dialog box, and returns the name of the file the user selected. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

FileOpenDlg(*path*, *filename*, *filetype*, *prompt*)

path is a string value, indicating the directory or folder to choose files from.

filename is a string value, indicating the default file name for the user to choose.

filetype is a string value, three or four characters long, indicating a file type (for example, “TAB” to specify tables).

prompt is a string title that appears on the bar at the top of the dialog box.

Return Value

String value, representing the name of the file the user chose (or an empty string if the user cancelled).

Description

The **FileOpenDlg( )** function displays a dialog box similar to the one that displays when the user chooses **File > Open**.

To choose a file from the list that appears in the dialog box, the user can either click a file in the list and click the **OK** button, or simply double-click a file in the list. In either case, the **FileOpenDlg( )** function returns a character string representing the full path and name of the file the user chose. Alternately, if the user clicks the **Cancel** button instead of picking a file, the dialog returns a null string ("").

The **FileOpenDlg( )** function does not actually open any files; it merely presents the user with a dialog box, and lets the user choose a file. If your application then needs to actually open the file chosen by the user, the application must issue a statement such as the **Open Table statement**. If you want your application to display an Open dialog box, and then you want MapInfo Professional to automatically open the selected file, you can issue a statement such as the **Run Menu Command statement** with M_FILE_OPEN or M_FILE_ADD_WORKSPACE.

The path parameter specifies the directory or folder from which the user will choose an existing file. Note that the path parameter only dictates the initial directory, it does not prevent the user from changing directories once the dialog box appears. If the path parameter is blank (a null string), the dialog box presents a list of files in the current working directory.

The *filename* parameter specifies the default file name for the user to choose.

The *filetype* parameter is a string, usually three or four characters long, which indicates the type of files that should appear in the dialog box. Some *filetype* settings have special meaning; for example, if the *filetype* parameter is “TAB”, the dialog box presents a list of MapInfo tables, and if the *filetype* parameter is “WOR”, the dialog box presents a list of MapInfo workspace files.

There are also a variety of other *filetype* values, summarized in the table below. If you specify one of the special type values from the table below, the dialog box includes a control that lets the user choose between seeing a list of table files or a list of all files (“*.*”).

***filetype* parameter**	**Type of files that appear**
“TAB”	MapInfo tables
“WOR”	MapInfo workspaces
“MIF”	MapInfo Interchange Format files, used for importing / exporting maps from / to ASCII text files.
“DBF”	dBASE or compatible data files
“WKS”, “WK1”	Lotus spreadsheet files

filetype parameter	Type of files that appear
“XLS”, "XLSX"	Excel spreadsheet files
“DXF”	AutoCAD data interchange format files
“MMI”, “MBI”	MapInfo for DOS interchange files
“MB”	MapBasic source program files
“MBX”	Compiled MapBasic applications
“TXT”	Text files
“BMP”	Windows bitmap files
“WMF”	Windows metafiles

Each of the three-character file types listed above corresponds to an actual file extension; in other words, specifying a *filetype* parameter of “WOR” tells MapBasic to display a list of files having the “.WOR” file extension, because that is the extension used by MapInfo Professional workspaces.

To help you write portable applications, MapBasic lets you use the same three-character *filetype* settings on all platforms. On Windows, a control in the lower left corner of the dialog box lets the user choose whether to see a list of files with the .TAB extension, or a list of all files in the current directory. If the **FileOpenDlg( )** function specifies a *filetype* parameter which is not listed in the table of file extensions above, the dialog box appears without that control.

Example

```
Dim s_filename As String
s_filename = FileOpenDlg("","","TAB","Open Table")
```

See Also:

FileSaveAsDlg( ) function, **Open File statement**, **Open Table statement**

FileSaveAsDlg( ) function

Purpose

Displays a Save As dialog box, and returns the name of the file the user entered. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

FileSaveAsDlg(*path*, *filename*, *filetype*, *prompt*)

path is a string value, indicating the default destination directory.

filename is a string value, indicating the default file name.

filetype is a string value, indicating the type of file that the dialog box lets the user choose.

prompt is a string title that appears at the top of the dialog box.

Return Value

String value, representing the name of the file the user entered (or an empty string if the user cancelled).

Description

The **FileSaveAsDlg( )** function displays a Save As dialog box, similar to the dialog box that displays when the user chooses **File > Save Copy As**.

The user can type in the name of the file they want to save. Alternately, the user can double-click from the list of grayed-out filenames that appears in the dialog box. Since each file name in the list represents an existing file, MapBasic asks the user to verify that they want to overwrite the existing file.

If the user specifies a filename and clicks **OK**, the **FileSaveAsDlg( )** function returns a character string representing the full path and name of the file the user chose. If the user clicks the **Cancel** button instead of picking a file, the function returns a null string ("").

The path parameter specifies the initial directory path. The user can change directories once the dialog box appears. If the path parameter is blank (a null string), the dialog box presents a list of files in the current directory.

The *filename* parameter specifies the default file name for the user to choose.

The *filetype* parameter is a three-character (or shorter) string which identifies the type of files that should appear in the dialog box. To display a dialog box that lists workspaces, specify the string “WOR” as the *filetype* parameter; to display a dialog box that lists table names, specify the string “TAB.” See **FileOpenDlg( ) function** for more information about three-character filetype codes.

The **FileSaveAsDlg( )** function does not actually save any files; it merely presents the user with a dialog box, and lets the user choose a file name to save. To save data under the file name chosen by the user, issue a statement such as the **Commit Table statement**.

See Also:

Commit Table statement, **FileOpenDlg( ) function**

Find statement

Purpose

Finds a location in a mappable table. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Find address [ , region ] [ Interactive ]
```

address is a string expression representing the name of a map object to find; to find the intersection of two streets, use the syntax: `streetname && streetname`.

region is the name of a region object which refines the search.

Description

The **Find** statement searches a mappable table for a named location (represented by the *address* parameter). MapBasic stores the search results in system variables, which a program can then access through the **CommandInfo( ) function**. If the **Find** statement includes the optional **Interactive** keyword, and if MapBasic is unable to locate the specified address, a dialog box displays a list of "near matches."

The **Find** statement can only search a mappable table (for example, a table which has graphic objects attached). The table must already be open. The **Find** statement operates on whichever column is currently chosen for searching. A MapBasic program can issue a **Find Using statement** to identify a specific table column to search. If the **Find** statement is not preceded by a Find Using statement, MapBasic searches whichever table was specified the last time the user chose MapInfo Professional's **Query > Find** command.

The **Find** statement can optionally refine a search by specifying a region name in addition to the address parameter. In other words, you could simply try to find a city name (for example, "Albany") by searching a table of cities; or you could refine the search by specifying both a city name and a region name (for example, "Albany", "CA"). The **Find** statement does not automatically add a symbol to the map to mark where the address was found. To create such a symbol, call the **CreatePoint( ) function** or the **Create Point statement**; see example below.

Determining Whether the Address Was Found

Following a **Find** statement, a MapBasic program can issue the function call **CommandInfo(CMD_INFO_FIND_RC)** to determine if the search was successful. If the search was successful, call **CommandInfo(CMD_INFO_X)** to determine the x-coordinate of the queried location, and call **CommandInfo(CMD_INFO_Y)** to determine the y-coordinate. To determine the row number that corresponds to the "found" address, call **CommandInfo(CMD_INFO_FIND_ROWID)**.

The **Find** statement may result in an exact match, an approximate match, or a failure to match. If the **Find** statement results in an exact match, the function call **CommandInfo(CMD_INFO_FIND_RC)** returns a value of one (1). If the **Find** statement results in an approximate match, the function call returns a value greater than one (1). If the **Find** statement fails to match the address, the function call returns a negative value.

The table below summarizes the Find-related information represented by the **CommandInfo(CMD_INFO_FIND_RC)** return value. The return value has up to three digits, and that each of the three digits indicates the relative success or failure of a different part of the search.

Digit Values	Meaning
*xx*1	Exact match.
*xx*2	A substitution from the abbreviations file used.
*xx*3 (–)	Exact match not found.
*xx*4 (–)	No object name specified; match not found.
*xx*5 (+)	The user chose a name from the Interactive dialog box.
*x*1*x*	Side of street undetermined.
*x*2*x* (+ / –)	Address number was within min/max range.
*x*3*x* (+ / –)	Address number was not within min/max range.
*x*4*x* (+ / –)	Address number was not specified.
*x*5*x* (–)	Streets do not intersect.
*x*6*x* (–)	The row matched does not have a map object.
*x*7*x* (+)	The user chose an address number from the Interactive dialog box.
1xx (+ / –)	Name found in only one region other than specified region.
2xx (–)	Name found in more than one region other than the specified region.
3xx (+ / –)	No refining region was specified, and one match was found.
4xx (–)	No region was specified, and multiple matches were found.
5xx (+)	Name found more than once in the specified region.
6xx (+)	The user chose a region name from the Interactive dialog box.

The Mod operator is useful when examining individual digits from the Find result. For example, to determine the last digit of a number, use the expression `number Mod 10`. To determine the last two digits of a number, use the expression `number Mod 100`; etc.

The distinction between exact and approximate matches is best illustrated by example. If a table of cities contains one entry for “Albany”, and the Find Using statement attempts to locate a city name without a refining region name, and the **Find** statement specifies an address parameter value of “Albany”, the search results in an exact match. Following such a **Find** statement, the function call **CommandInfo(CMD_INFO_FIND_RC)** would return a value of 1 (one), indicating that an exact match was found.

Now suppose that the Find operation has been set up to refine the search with an optional region name; in other words, the **Find** statement expects a city name followed by a state name (for example, “Albany”, “NY”). If a MapBasic program then issues a **Find** statement with “Albany” as the

address and a null string as the state name, that is technically not an exact match, because MapBasic expects the city name to be followed by a state name. Nevertheless, if there is only one “Albany” record in the table, MapBasic will be able to locate that record. Following such a Find operation, the function call **CommandInfo(CMD_INFO_FIND_RC)** would return a value of 301. The 1 digit signifies that the city name matched exactly, while the 3 digit indicates that MapBasic was only partly successful in locating a correct refining region.

If a table of streets contains “Main St”, and a **Find** statement attempts to locate “Main Street”, MapBasic considers the result to be an approximate match (assuming that abbreviation file processing has been enabled; see **Find Using statement**). Strictly speaking, the string “Main Street” does not match the string “Main St”. However MapBasic is able to match the two strings after substituting possible abbreviations from the MapInfo Professional abbreviations file (MAPINFOW.ABB). Following the **Find** statement, the **CommandInfo(CMD_INFO_FIND_RC)** function call returns a value of 2.

If the Find operation presents the user with a dialog box, and the user enters text in the dialog box in order to complete the find, then the return code will have a 1 (one) in the millions place.

Example

```
Include "mapbasic.def"
Dim x, y As Float, win_id As Integer
Open Table "states" Interactive
Map From States
win_id = FrontWindow( )
Find Using states(state)
Find "NY"
If CommandInfo(CMD_INFO_FIND_RC) >= 1 Then
   x = CommandInfo(CMD_INFO_X)
   y = CommandInfo(CMD_INFO_Y)
   Set Map
      Window win_id
      Center (x, y)
   ' Now create a symbol at the location we found.
   ' Create the object in the Cosmetic layer.
   Insert Into
      WindowInfo( win_id, WIN_INFO_TABLE) (Object)
      Values ( CreatePoint(x, y) )
Else
   Note "Location not found."
End If
```

See Also:

Find Using statement, **CommandInfo( ) function**

Find Using statement

Purpose

Dictates which table(s) and column(s) should be searched in subsequent Find operations. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Find Using table ( column )
   [ Refine Using table ( column ) ]
   [ Options
      [ Abbrs { On | Off } ]
      [ ClosestAddr { On | Off } ]
      [ OtherBdy { On | Off } ]
      [ Symbol symbol_style ] ]
      [ Inset inset_value { Percent | Distance Units dist_unit} ]
      [ Offset value ] [ Distance Units dist_unit ] ]
```

table is the name of an open table.

column is the name of a column in the table.

symbol_style is a Symbol variable or a function call that returns a Symbol value; this controls what type of symbol is drawn on the map if the user chooses **Query > Find**.

inset_value is a positive integer value representing how far from the ends of the line to adjust the placement of an address location.

value specifies the Offset value (the distance back from the street).

dist_unit is a string that represents the name of a distance unit (for example, “mi” for miles, “m” for meters.

Description

The **Find Using** statement specifies which table(s) and column(s) MapBasic will search when performing a **Find statement**. Note that the column specified must be indexed.

The optional **Refine** clause specifies a second table, which will act as an additional search criterion; the table must contain region objects. The specified column does not need to be indexed. If you omit the **Refine** clause, subsequent Find statements expect a simple location name (for example, “Portland”). If you include a **Refine** clause, subsequent Find statements expect a location name and a region name (for example, “Portland”, “OR”).

The optional **Abbrs** clause dictates whether MapBasic will try substituting abbreviations from the abbreviations file in order to find a match. By default, this option is enabled (**On**); to disable the option, specify the clause **Abbrs Off**.

The optional **ClosestAddr** clause dictates whether MapBasic will use the closest available address number in cases where the address number does not match. By default, this option is disabled (**Off**); to enable the option, specify the clause **ClosestAddr On**.

The optional **OtherBdy** clause dictates whether MapBasic will match to a record found in a refining region other than the refining region specified. By default, this option is disabled (**Off**); to enable the option, specify the clause **OtherBdy On**.

MapInfo Professional saves the **Inset** and **Offset** settings specified the last time the user chose **Query > Find Options**, **Table > Geocode Options** or executed a **Find Using** statement. Thus, the last specified inset/offset options becomes the default settings for the next time.

If **Percent** is specified, it represents the percentage of the length of the line where the address is to be placed. For **Percent**, valid values for *inset_value* are from 0 to 50. If **Distance Units** are specified, *inset_value* represents the distance from the ends of the line where the address is to be placed. For distance, valid values for *inset_value* are from 0 to 32,767. The inset takes the addresses that would normally fall at the end of the street and moves them away from the end going in the direction towards the center.

The **Offset** value sets the addresses back from the street instead of right on the street. *value* is a positive integer value representing how far to offset the placement of an address location back from the street. Valid values are from 0 to 32,767.

Example

```
Find Using city_1k(city)
   Refine Using states(state)

Find "Albany", "NY"
```

See Also:

Create Index statement, **Find statement**

Fix( ) function

Purpose

Returns an integer value, obtained by removing the fractional part of a decimal value. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Fix(*num_expr*)

num_expr is a numeric expression.

Return Value

Integer

Description

The **Fix( )** function removes the fractional portion of a number, and returns the resultant integer value. The **Fix( )** function is similar to, but not identical to, the **Int( ) function**. The two functions differ in the way that they treat negative fractional values. When passed a negative fractional number, **Fix( )** returns the nearest integer value greater than or equal to the original value; thus, the function call:

```
Fix(-2.3)
```

returns a value of -2. But when the Int( ) function is passed a negative fractional number, it returns the nearest integer value that is less than or equal to the original value. Thus, the function call:

```
Int(-2.3)
```

returns a value of -3.

Example

```
Dim i_whole As Integer
i_whole = Fix(5.999)
' i_whole now has the value 5.

i_whole = Fix(-7.2)
' i_whole now has the value -7.
```

See Also:

Int( ) function, **Round( ) function**

Font clause

Purpose

Specifies a text style. You can use this clause in the MapBasic Window in MapInfo Professional.

Syntax

Font *font_expr*

font_expr is a Font expression, for example, **MakeFont**(*fontname*, *style*, *size*, *fgcolor*, *bgcolor*).

Description

The **Font** clause specifies a text style. **Font** is a clause, not a complete MapBasic statement. Various object-related statements, such as the **Create Text statement**, allow you to specify a Font setting; this lets you choose the typeface and point size of the new text object. If you omit the **Font** expression from a Create Text statement, the new object uses MapInfo Professional's current Font. The keyword **Font** may be followed by an expression that evaluates to a Font value.

This expression can be a Font variable:

```
Font font_var
```

or a call to a function (for example, **CurrentFont( ) function** or **MakeFont( ) function**) which returns a Font value:

```
Font MakeFont("Helvetica", 1, 12, BLACK, WHITE)
```

With some MapBasic statements (for example, the **Set Legend statement**), the keyword **Font** can be followed immediately by the five parameters that define a Font style (font name, style, point size, foreground color, and background color) within parentheses:

```
Font("Helvetica", 1, 12, BLACK, WHITE)
```

The following table summarizes the components that define a font:

Component	Description
font name	A string that identifies a font. The set of available fonts depends on the user's system and the hardware platform in use.
style	Integer value. Controls text attributes such as bold, italic, and underline. See table below for details.
size	Integer value representing a point size. A point size of twelve is one-sixth of an inch tall.
foreground color	Integer RGB color value, representing the color of the text. See **Rnd( ) function**.
background color	Integer RGB color value. If the halo style is used, this is the halo color; otherwise, this is the background fill color. To specify a transparent background style in a **Font** clause, omit the background color. For example: `Font( "Helvetica", 1, 12, BLACK).` To specify a transparent fill when calling the **MakeFont( ) function**, specify -1 as the background color.

The following table shows how the style parameter corresponds to font styles.

Style Value	Description of text style
0	Plain
1	Bold
2	Italic
4	Underline
8	Strikethrough
32	Shadow

Style Value	Description of text style
256	Halo
512	All Caps
1024	Expanded

To specify two or more style attributes, add the values from the left column. For example, to specify both the Bold and All Caps attributes, use a style value of 513.

Example

```
Include "MAPBASIC.DEF"
Dim o_title As Object
Create Text
   Into Variable o_title
   "Your message could go HERE"
   (73.5, 42.6) (73.67, 42.9)
   Font MakeFont("Helvetica",1,12,BLACK,WHITE)
```

See Also:

Alter Object statement, **Chr$( ) function**, **Create Text statement**, **RGB( ) function**

For…Next statement

Purpose

Defines a loop which will execute for a specific number of iterations.

Restrictions

You cannot issue a **For…Next** statement through the MapBasic window.

Syntax

```
For var_name = start_expr To end_expr [ Step inc_expr ]
    statement_list
Next
```

var_name is the name of a numeric variable.

start_expr is a numeric expression.

end_expr is a numeric expression.

inc_expr is a numeric expression.

statement_list is the group of statements to execute with each iteration of the For loop.

Description

The **For…Next** statement provides loop control. This statement requires a numeric variable (identified by the *var_name* parameter). A **For…Next** statement either executes a group of statements (the *statement_list*) a number of times, or else skips over the *statement_list* completely. The *start_expr*, *end_expr*, and *inc_expr* values dictate how many times, if any, the *statement_list* will be carried out.

Upon encountering a **For…Next** statement, MapBasic assigns the *start_expr* value to the *var_name* variable. If the variable is less than or equal to the *end_expr* value, MapBasic executes the group of statements in the *statement_list*, and then adds the *inc_expr* increment value to the variable. If no **Step** clause was specified, MapBasic uses a default increment value of one. MapBasic then compares the current value of the variable to the *end_expr* expression; if the variable is currently less than or equal to the *end_expr* value, MapBasic once again executes the statements in the *statement_list*. If, however, the *var_name* variable is greater than the *end_expr*, MapBasic stops the For loop, and resumes execution with the statement which follows the **Next** statement.

Conversely, the **For…Next** statement can also count downwards, by using a negative **Step** value. In this case, each iteration of the For loop decreases the value of the *var_name* variable, and MapBasic will only decide to continue executing the loop as long as *var_name* remains greater than or equal to the *end_expr*.

Each **For** statement must be terminated by a **Next** statement. Any statements which appear between the **For** and **Next** statements comprise the *statement_list*; this is the list of statements which will be carried out upon each iteration of the loop.

The **Exit For statement** allows you to exit a For loop regardless of the status of the *var_name* variable. The **Exit For statement** tells MapBasic to jump out of the loop, and resume execution with the first statement which follows the **Next** statement.

MapBasic permits you to modify the value of the *var_name* variable within the body of the For loop; this can affect the number of times that the loop is executed. However, as a matter of programming style, you should try to avoid altering the contents of the *var_name* variable within the loop.

Example

```
Dim i As Integer

' the next loop will execute a Note statement 5 times
For i = 1 to 5
   Note "Hello world!"
Next

' the next loop will execute the Note statement 3 times
For i = 1 to 5 Step 2
   Note "Hello world!"
Next

' the next loop will execute the Note statement 3 times
For i = 5 to 1 Step -2
   Note "Hello world!"
Next
```

```
' MapBasic will skip the following For statement
' completely, because the initial start value is
' already larger than the initial end value
For i = 100 to 50 Step 5
   Note "This note will never be executed"
Next
```

See Also:

Do…Loop statement, **Exit For statement**

ForegroundTaskSwitchHandler procedure

Purpose

A reserved procedure name, called automatically when MapInfo Professional receives the focus (becoming the active application) or loses the focus (another application becomes active).

Syntax

```
Declare Sub ForegroundTaskSwitchHandler

Sub ForegroundTaskSwitchHandler
    statement_list
End Sub
```

statement_list is a list of statements.

Description

If the user runs an application containing a procedure named **ForegroundTaskSwitchHandler**, MapInfo Professional calls the procedure automatically whenever MapInfo Professional receives or loses the focus. Within the procedure, call the **CommandInfo( ) function** to determine whether MapInfo Professional received or lost the focus.

Example

```
Sub ForegroundTaskSwitchHandler

   If CommandInfo(CMD_INFO_TASK_SWITCH)
        = SWITCHING_INTO_MAPINFO Then

   ' ... then MapInfo just became active
   Else
   ' ... another app just became active
   End If

End Sub
```

See Also:

CommandInfo( ) function

Format$( ) function

Purpose

Returns a string representing a custom-formatted number. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Format$(*value*, *pattern*)

value is a numeric expression.

pattern is a string which specifies how to format the results.

Return Value

String

Description

The **Format$( )** function returns a string representing a formatted number. Given a numeric value such as 12345.67, **Format$( )** can produce formatted results such as “$12,345.67”.

The *value* parameter specifies the numeric value that you want to format.

The *pattern* parameter is a string of code characters, chosen to produce a particular type of formatting. The pattern string should include one or more special format characters, such as `#`, `0`, `%`, the comma character (`,`), the period (`.`), or the semi-colon (`;`); these characters control how the results will look. The table below summarizes the format characters.

pattern character	Role in formatting results:
`#`	The result will include one or more digits from the value.
	If the pattern string contains one or more `#` characters to the left of the decimal place, and if the value is between zero and one, the formatted result string will not include a zero before the decimal place.
`0`	A digit placeholder similar to the `#` character. If the pattern string contains one or more `0` characters to the left of the decimal place, and the value is between zero and one, the formatted result string will include a zero before the decimal place. See examples below.
`.` (period)	The pattern string must include a period if you want the result string to include a “decimal separator.” The result string will include the decimal separator currently in use on the user's computer. To force the decimal separator to be a period, use the **Set Format statement**.

pattern character	Role in formatting results:
, (comma)	The pattern string must include a comma if you want the result string to include “thousand separators.” The result string will include the thousand separator currently set up on the user's computer. To force the thousand separator to be a comma, use the Set Format statement.
%	The result will represent the value multiplied by one hundred; thus, a value of 0.75 will produce a result string of “75%”. If you wish to include a percent sign in your result, but you do not want MapBasic to multiply the value by one hundred, place a \ (back slash) character before the percent sign (see below).
E+	The result is formatted with scientific notation. For example, the value 1234 produces the result “1.234e+03”. If the exponent is positive, a plus sign appears after the “e”. If the exponent is negative (which is the case for fractional numbers), the results include a minus sign after the “e”.
E-	This string of control characters functions just as the “E+” string, except that the result will never show a plus sign following the “e”.
; (semi-colon)	By including a semicolon in your pattern string, you can specify one format for positive numbers and another format for negative numbers. Place the semicolon after the first set of format characters, and before the second set of format characters. The second set of format characters applies to negative numbers. If you want negative numbers to appear with a minus sign, include “-” in the second set of format characters.
\	If the back slash character appears in a pattern string, MapBasic does not perform any special processing for the character which follows the back slash. This lets you include special characters (for example, %) in the results, without causing the special formatting actions described above.

Error Conditions

ERR_FCN_INVALID_FMT (643) error generated if the pattern string is invalid

Examples

The following examples show the results you can obtain by using various pattern strings. The results are shown as comments in the code.

You will obtain slightly different results if your computer is set up with non-US number formatting.

```
Format$( 12345, ",#") ' returns "12,345"
Format$(-12345, ",#") ' returns "-12,345"
Format$( 12345, "$#") ' returns "$12345"
Format$(-12345, "$#") ' returns "-$12345"
```

```
Format$( 12345.678, "$,#.##") ' returns "$12,345.68"
Format$(-12345.678, "$,#.##") ' returns "-$12,345.68"

Format$( 12345.678, "$,#.##;($,#.##)") 'returns "$12,345.68"
Format$(-12345.678, "$,#.##;($,#.##)") 'returns "($12,345.68)"
Format$(12345.6789, ",#.###") ' returns "12,345.679"
Format$(12345.6789, ",#.#") ' returns "12,345.7"

Format$(-12345.6789, "#.###E+00") ' returns "-1.235e+04"
Format$( 0.054321, "#.###E+00") ' returns "5.432e-02"

Format$(-12345.6789, "#.###E-00") ' returns "-1.235e04"
Format$( 0.054321, "#.###E-00") ' returns "5.432e-02"

Format$(0.054321, "#.##%") ' returns "5.43%"
Format$(0.054321, "#.##\%") ' returns ".05%"
Format$(0.054321, "0.##\%") ' returns "0.05%"
```

See Also:

Str$( ) function

FormatDate$( ) function

Purpose

Returns a date formatted in the short date style specified by the Control Panel. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

FormatDate$(*value*)

value is a number or string representing the date in a YYYYMMDD format.

Return Value

String

Description

The **FormatDate$( )** function returns a string representing a date in the local system format as specified by the Control Panel.

If you specify the year as a two-digit number (for example, 96), MapInfo Professional uses the current century or the century as determined by the **Set Date Window statement**.

Year can take two-digit year expressions. Use the Date window to determine which century should be used. See **DateWindow( ) function**.

Examples

Assuming Control Panel settings are d/m/y for date order, '-' for date separator, and “dd-MMM-yyyy” for short date format:

```
Dim d_Today As Date
d_Today = CurDate( )
Print d_Today 'returns "19970910"
Print FormatDate$( d_Today ) 'returns "10-Sep-1997"
Dim s_EnteredDate As String
s_EnteredDate = "03-02-61"
Print FormatDate$( s_EnteredDate ) 'returns "03-Feb-1961"
s_EnteredDate = "12-31-61"
Print FormatDate$( s_EnteredDate ) ' returns ERROR: not d/m/y ordering
s_EnteredDate = "31-12-61"
Print FormatDate$( s_EnteredDate ) ' returns 31-Dec-1961"
```

See Also:

DateWindow( ) function, **Set Date Window statement**

FormatNumber$( ) function

Purpose

Returns a string representing a number, including thousands separators and decimal-place separators that match the user's system configuration. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

FormatNumber$(*num*)

num is a numeric value or a string that represents a numeric value, such as “1234.56”.

Return Value

String

Description

Returns a string that represents a number. If the number is large enough to need a thousands separators, this function inserts thousands separators. MapInfo Professional reads the user's system configuration to determine which characters to use as the thousands separator and decimal separator.

Examples

The following table demonstrates how the **FormatNumber$( )** function with a comma as the thousands separator and period as the decimal separator (United States defaults):

Function Call	Result returned
`FormatNumber$("12345.67")`	“12,345.67” (inserted a thousands separator)
`FormatNumber$("12,345.67")`	“12,345.67” (no change)

If the user's computer is set up to use period as the thousands separator and comma as the decimal separator, the following table demonstrates the results:

Function Call	Result returned
`FormatNumber$("12345.67")`	“12.345,67” (inserted a thousands separator, and changed the decimal separator to match user's setup)
`FormatNumber$("12,345.67")`	“12.345,67” (changed both characters to match the user's setup)

See Also:

DeformatNumber$( ) function

FormatTime$ function

Purpose

Returns a string representing a time using the format specified in the second argument. You can call this function from the MapBasic Window in MapInfo Professional.

The format string should follow the same Microsoft standards as for setting the locale time format:

Hours	Meaning
h	Hours without leading zeros for single-digit hours (12-hour clock).
hh	Hours with leading zeros for single-digit hours (12-hour clock).
H	Hours without leading zeros for single-digit hours (24-hour clock).
HH	Hours with leading zeros for single-digit hours (24-hour clock).
Minutes	**Meaning**
m	Minutes without leading zeros for single-digit minutes.
mm	Minutes with leading zeros for single-digit minutes.
Seconds	**Meaning**

s	Seconds without leading zeros for single-digit seconds.
ss	Seconds with leading zeros for single-digit seconds.
Time marker	**Meaning**
t	One-character time marker string. ⓘ Do not to use this format for certain languages, for example, Japanese (Japan). With this format, the application always takes the first character from the time marker string, defined by LOCALE_S1159 (AM) and LOCALE_S2359 (PM). Because of this, the application can create incorrect formatting with the same string used for both AM and PM.
tt	Multi-character time marker string.

Source: **http://msdn2.microsoft.com/en-us/library/ms776320.aspx**

ⓘ In the preceding formats, the letters m, s, and t must be lowercase, and the letter h must be lowercase to denote the 12-hour clock or uppercase to denote the 24-hour clock.

Our code follows the rules for specifying the system local time format. In addition, we also allow the user to specify f, ff, or fff for tenths of a second, hundredths of a second, or milliseconds.

Syntax

```
FormatTime$ (Time, String)
```

Return Value

String

Example

Copy this example into the MapBasic window for a demonstration of this function.

```
dim Z as time
Z = CurTime()
Print FormatTime$(Z, "hh:mm:ss.fff tt")
```

FME Refresh Table statement

Purpose

Refreshes a Universal Data Source (FME) table from the original data source. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

FME Refresh Table *alias*

alias is the an alias for an open registered Universal Data Source (FME) table.

Example

The following example refreshes the local table named watershed.

```
FME Refresh Table watershed
```

FrontWindow( ) function

Purpose

Returns the integer identifier of the active window. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

FrontWindow( )

Return Value

Integer

Description

The **FrontWindow( )** function returns the integer ID of the foremost document window (Map, Browse, Graph, or Layout). Note that immediately following a statement which creates a new window (for example, Map, Browse, Graph, Layout), the new window is the foremost window.

Example

```
Dim map_win_id As Integer
Open Table "states"
Map From states
map_win_id = FrontWindow( )
```

See Also:

NumWindows( ) function, **WindowID( ) function**, **WindowInfo( ) function**

Function…End Function statement

Purpose

Defines a custom function.

Restrictions

You cannot issue a **Function…End Function** statement through the MapBasic window.

Syntax

```
Function name ( [ [ ByVal ] parameter As datatype ]
    [, [ ByVal ] parameter As datatype... ] ) As return_type
   statement_list
End Function
```

name is the function name.

parameter is the name of a parameter to the function.

datatype is a variable type, such as integer; arrays and custom Types are allowed.

return_type is a standard scalar variable type; arrays and custom Types are not allowed.

statement_list is the list of statements that the function will execute.

Description

The **Function…End Function** statement creates a custom, user-defined function. User-defined functions may be called in the same fashion that standard MapInfo Professional functions are called.

Each **Function…End Function** definition must be preceded by a **Declare Function statement**.

A user-defined function is similar to a **Sub** procedure; but a function returns a value. Functions are more flexible, in that any number of function calls may appear within one expression. For example, the following statement performs an assignment incorporating two calls to the **Proper$( ) function**:

```
fullname = Proper$(firstname) + " " + Proper$(lastname)
```

Within a **Function…End Function** definition, the function name parameter acts as a variable. The value assigned to the name “variable” will be the value that is returned when the function is called. If no value is assigned to name, the function will always return a value of zero (if the function has a numeric data type), FALSE (if the function has a logical data type), or a null string (if the function has a string data type).

Restrictions on Parameter Passing

A function call can return only one “scalar” value at a time. In other words, a single function call cannot return an entire array's worth of values, nor can a single function call return a set of values to fill in a custom data Type variable. By default, every parameter to a user-defined function is a by-reference parameter. This means that the function's caller must specify the name of a variable as the parameter. If the function modifies the value of a by-reference parameter, the modified value will be reflected in the caller's variable.

Any or all of a function's parameters may be specified as by-value if the optional **ByVal** keyword precedes the parameter name in the **Function…End Function** definition. When a parameter is declared by-value, the function's caller can specify an expression for that parameter, rather than having to specify the name of a single variable. However, if a function modifies the value of a by-value parameter, there is no way for the function's caller to access the new value. You cannot pass arrays, custom Type variables, or Alias variables as **ByVal** parameters to custom functions. However, you can pass any of those data types as by-reference parameters. If your custom function takes no parameters, your **Function…End Function** statement can either include an empty pair of parentheses, or omit the parentheses entirely. However, every function call must include a pair of

parentheses, regardless of whether the function takes parameters. For example, if you wish to define a custom function called Foo, your **Function…End Function** statement could either look like this:

```
Function Foo( )
   ' ... statement list goes here ...
End Function
```

or like this:

```
Function Foo
   ' ... statement list goes here ...
End Function
```

but all calls to the function would need to include the parentheses, in this fashion:

```
var_name = Foo( )
```

Availability of Custom Functions

The user may not incorporate calls to user-defined functions when filling in standard MapInfo Professional dialog boxes. A custom function may only be called from within a compiled MapBasic application. Thus, a user may not specify a user-defined function within the SQL Select dialog box; however, a compiled MapBasic program may issue a **Select statement** which does incorporate calls to user-defined functions.

A custom function definition is only available from within the application that defines the function. If you write a custom function which you wish to include in each of several MapBasic applications, you must copy the **Function…End Function** definition to each of the program files.

Function Names

The **Function…End Function** statement's name parameter can match the name of a standard MapBasic function, such as **Abs** or **Chr$**. Such a custom function will replace the standard MapBasic function by the same name (within the confines of that MapBasic application). If a program defines a custom function named **Abs**, any subsequent calls to the **Abs** function will execute the custom function instead of MapBasic's standard **Abs( ) function**.

When a MapBasic application redefines a standard function in this fashion, other applications are not affected. Thus, if you are writing several separate applications, and you want each of your applications to use your own, customized version of the **Distance( ) function**, each of your applications must include the appropriate **Function…End Function** statement.

When a MapBasic application redefines a standard function, the re-definition applies throughout the entire application. In every procedure of that program, all calls to the redefined function will use the custom function, rather than the original.

Example

The following example defines a custom function, CubeRoot, which returns the cube root of a number (the number raised to the one-third power). Because the call to CubeRoot appears earlier in the program than the CubeRoot Function…End Function definition, this example uses the **Declare Function statement** to pre-define the CubeRoot function parameter list.

```
Declare Function CubeRoot(ByVal x As Float) As Float
Declare Sub Main

Sub Main
   Dim f_result As Float
   f_result = CubeRoot(23)
   Note Str$(f_result)
End Sub

Function CubeRoot(ByVal x As Float) As Float
   CubeRoot = x ^ 0.33333333333
End Function
```

See Also:

Declare Function statement, **Declare Sub statement**, **Sub…End Sub statement**

Geocode statement

Purpose

Geocodes a table or individual value using a remote geocode service through a connection created using the **Open Connection statement** and set up using the **Set Connection Geocode statement**. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Geocode connection_number
   Input
      [ Table input_tablename ]
      [ Country = Country_expr
         [ Street = Street_expr,
            [ IntersectingStreet = IntersectingStreet_expr ],
          Municipality = Municipality_expr,
          CountrySubdivision = CountrySubdiv_expr,
          PostalCode = PostalCode_expr,
          CountrySecondarySubdivision= CountrySecondarySubdiv_expr,
          SecondaryPostalCode = SecondaryPostalCode_expr,
          Placename = Placename_expr,
          Street2 = Street2_expr,
          MunicipalitySubdivision = MunicipalitySubdiv_expr ] ]
   Output
      [ Into
         [ Table out_tablename [ Key out_keycolumn = in_keyexpr ] ] |
         [ Variable variable_name ]]
      [ Point [ On | Off ] [ Symbol Symbol_expr ], ]
      [ Street_column = Street, Municipality_column = Municipality,
       CountrySubdiv_column = CountrySubdivision,
       PostalCode_column = PostalCode,
       CountrySecondarySubdiv_column = CountrySecondarySubdivision,
       SecondaryPostalCode_column = SecondaryPostalCode,
```

```
    Placename_column = Placename,
    MunicipalitySubdiv_column = MunicipalitySubdivision,
    Country_column = Country, ResultCode_column = ResultCode,
    Latitude_column = Latitude, Longitude_column = Longitude
    Columns colname = geocoder_keyname [,] ...]
[ Interactive [ On [ Max Candidates candidates_expr | All ]
   [ CloseMatchesOnly [ On | Off ] ]
     | Off [ First | None ]]
```

connection_number is the number returned when the connection was created. See **Open Connection statement**.

input_tablename is a table alias of an open table including result sets and selections.

Country_expr is a string expression representing the three letter ISO code for the country.

Street_expr is an expression that specifies a street address.

IntersectingStreet_expr is an expression that specifies a street that should intersect with the street specified in *Street_expr*.

Municipality_expr is an expression that specifies the name of a municipality.

CountrySubdivision_expr is an expression that specifies the name of a subdivision of a country. For example, in the US this specifies the name of a state. In Canada it specifies the name of a province.

PostalCode_expr is an expression that specifies a postal code.

CountrySecondarySubdiv_expr is an expression that specifies the name of a secondary subdivision for a country. For example, in the US this corresponds to a county, in Canada this corresponds to a census division.

SecondaryPostalCode_expr is an expression that specifies a secondary postal code system. In the US this corresponds to a ZIP+4 extension on a ZIP Code.

Placename_expr is an expression that specifies the name of a well-known place, such as a large building that may contain multiple addresses.

Street2_expr is an expression that specifies a secondary address line.

MunicipalitySubdiv_expr is an expression that specifies the name of a municipality subdivision.

out_tablename is a table alias of a table to be used as the holder of the data resulting from the geocode operation.

out_keycolumn is a string representing the name of a key column in the output table that will be used to hold some identifying “key” from the input records. This is used to identify the record from where the geocode came.

in_keyexpr is an expression from (the input table) whose value is inserted in the output record.

variable_name is the name of a variable that can hold a single geometry.

Symbol_expr is an expression that specifies the symbol to use when displaying a Point from the geometry column. See **Symbol clause** for more information.

Street_column is an alias that represents the name of the column to hold the Street result.

Municipality_column is a string the represents the name of the column to hold the Municipality result.

CountrySubdiv_column is a string the represents the name of the column to hold the Country Subdivision result.

PostalCode_column is a string the represents the name of the column to hold the Postal Code result.

CountrySecondarySubdiv_column is a string the represents the name of the column to hold the Country Secondary Subdivision result.

SecondaryPostalCode_column is a string the represents the name of the column to hold the Secondary Postal Code result.

Placename_column is a string the represents the name of the column to hold the Placename result.

MunicipalitySubdiv_column is a string the represents the name of the column to hold the Municipality Subdivision result.

Country_column is a string the represents the name of the column to hold the Country result.

ResultCode_column is a string the represents the name of the column to hold the Result Code generated by the geocoder.

Latitude_column is a string the represents the name of the float or decimal column to hold the Latitude result.

Longitude_column is a string the represents the name of the float or decimal column to hold the Longitude result.

colname is a string the represents the name of the column for a geocoder-specific result.

geocoder_keyname is a string representing the name of a country-specific geocoder item. These items are documented by the specific geocoder.

candidates_expr is an expression that specifies the number of candidates to be returned in an interactive geocoding session.

Description

Every **Geocode** statement must include an **Input** clause and an **Output** clause. The *input_tablename* is optional, however if a table is not specified, the resulting geocode operation would be performed on a set of string inputs (variables or constants), so that only a single address is geocoded in each request. The *output_tablename* is also optional. See **Table vs. non-table- based input and output** below.

Input clause

The **Input** clause is required as a geocode request needs some input data.

A **Country** must be specified either as an explicit argument or as a column in *input_tablename*. When a single country is used, it can be a constant string if no data is available. ISO standard three letter country codes must be used.

The list of fields to include from *input_table* to be geocoded must include at least one value. The more expressions that are included, the more accurate your geocoding result will be.

Output clause

The **Output** clause is required, as without it, the entire command returns nothing.

Into Table indicates that the **Output** clause refer to columns in *output_table*, which must be writable. If not specified, the clauses refer to the *input_table*. Note that if the input columns are to be updated, they must be specified both for input AND output.

Key is used with **Into Table**. This clause creates a relationship between the key columns in the input and output table.

Variable specifies that the geometry result from the geocode operation is stored in a variable defined in *variable_name*. When using this output option, note that if the input is a table only the first record is processed and the remainder of the records are skipped.

Point specifies that the geographic result of the geocode is to be stored in either the table or the variable. In the case of a table, this requires that the table be mappable.

To store the point stored into the object column, specify **Point** or **Point On** (default is on). The current default symbol is used. To return the same using a specific symbol, specify **Point On Symbol** *symbol_expr*. If you do not want to store the point in the object column, specify **Point Off**. Whether you want the object created or not, you can still store the x and y values in real number columns. To do this specify those columns as **Latitude** = *latitude_column* **Longitude** = *longitude_column*.

The rest of the output data specifies columns in the output table where well known geocoder return values are stored. In general, these may be more specific than the input. For example, it may be possible to geocode an address with just a business name of “MapInfo” and a post code of “12180”. However, much more is returned in the output. The Columns extension allows for data to be returned that is geocoder specific. The user must know the names of the keys as defined by the geocoders.

Table vs. non-table- based input and output

The **Geocode** statement can be used with any combination of table-based and non-table-based inputs and outputs. If you choose to use a table-based input you can have your output placed into either a new or existing table, or into a variable. If the output is a variable then only the first record is processed and the only value stored is the geographic object.

If you choose non-table-based input, the values for the operation must either be expressions (not column names), variables, or constant strings, the output can be placed either into a table or assigned to a variable.

Interactive clause

Interactive [**On** | **Off**] is an optional keyword that controls whether a dialog box to be displayed in the case of multiple candidates returned for each address. When this occurs, the user is prompted to choose, respecify, skip, or cancel the operation.

is asked to decide which of the choices is best given the opportunity to skip this input. When **On**, the dialog box displays in these situations. When **Off**, if multiple matches occur the choices are to accept the first candidate or none, meaning that the record is skipped. The default is skipping the record.

If the **Interactive** keyword is not included, it is equivalent to **Interactive Off None** and no options can be specified. If **Interactive** is specified, the default is **On**.

- **Interactive** is equivalent to **Interactive On**. When no value is provided for Max the default is three (3) candidates to be returned.
- **Interactive On Max Candidates All** returns all candidates
- **Interactive On Max Candidates** `4* myMBVariable/6` returns the number of candidates resulting from the evaluation of the expression.
- **Interactive Off** is equivalent to **Interactive Off None**.
- **Interactive Off First** returns the first candidate in the list.

The **CloseMatchesOnly** setting sets the geocode service to only return close matches as defined by the server. If **CloseMatchesOnly** is set to **Off**, all results are returned up to the number defined in **Max Candidates** with the ones that are considered to be close marked as such.

Examples

The following example shows a geocode request using the nystreets table and specifying the use of the city, Streetname, state, and postalcode.

```
Geocode connectionHandle Input Table nystreets municipality=city,
street=StreetName, countrysubdivision=state, postalcode=zip,
country="usa"
OUTPUT StreetName=street, address=municipality
```

This example shows a geocode request using the nystreets table and specifying a symbol for displaying the output.

```
Geocode connectionHandle Input Table nystreets street=StreetName,
country="usa"
Output Point Symbol MakeFontSymbol(65, 255 ,24,"MapInfo
Cartographic",32,0), StreetName=street
```

This example sends a request with the Interactive set to On with the return value being placed into the street column.

```
Geocode connectionHandle Input Table nystreets street=StreetName,
country="usa"
Output Point Symbol MakeFontSymbol(65, 255 ,24,"MapInfo
Cartographic",32,0),
StreetName=street Interactive on Max Candidates 5
```

The following example shows a Geocode request without using a table and outputting the results into a variable:

```
Geocode connectionHandle Input street="1 Global View", country="usa",
countrysubdivison="NY", municipality="Troy"
Output Variable outvar
```

See Also:

Open Connection statement

GeocodeInfo( ) function

Purpose

Returns any and all attributes that were set on a connection using the **Set Connection Geocode statement**. In addition, **GeocodeInfo( )** can also return some status values from the last geocode command issued using each connection. There is also an attribute to handle the maximum number of addresses that the server will permit to be sent to the service at a time. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

GeoCodeInfo(*connection_handle, attribute*)

connection_handle is an Integer.

attribute is an Integer code, indicating which type of information should be returned.

Return Value

Float, Integer, SmallInt, Logical, or String, depending on the attribute parameter.

Description

The **GeoCodeInfo( )** function returns the properties defaulted by the connection or the properties that have been changed using Set GeoCode. Like many functions of this type in MapBasic, the return values vary according to the *attribute* parameter. All the codes for these values are listed in MAPBASIC.DEF.

attribute Value	ID	GeoCodeInfo( ) Return Value
GEOCODE_STREET_NAME	1	Logical representing whether or not a match for StreetName is set.
GEOCODE_STREET_NUMBER	2	Logical representing whether or not a match for StreetNumber is set.
GEOCODE_MUNICIPALITY	3	Logical representing whether or not a match for municipality is set.
GEOCODE_MUNICIPALITY2	4	Logical representing whether or not a match for MunicipalitySubdivision is set.
GEOCODE_COUNTRY_SUBDIVISION	5	Logical representing whether a match for CountrySubdivision is set.
GEOCODE_COUNTRY_SUBDIVISION2	6	Logical representing whether or not a match for CountrySecondarySubdivision is set.

attribute Value	ID	GeoCodeInfo( ) Return Value
GEOCODE_POSTAL_CODE	7	Logical representing whether or not a match for PostalCode is set.
GEOCODE_DICTIONARY	9	SmallInt value representing one of these five values: • DICTIONARY_ALL • DICTIONARY_ADDRESS_ONLY • DICTIONARY_USER_ONLY • DICTIONARY_PREFER_ADDRESS • DICTIONARY_PREFER_USER
GEOCODE_BATCH_SIZE	10	Integer value representing the batch size.
GEOCODE_FALLBACK_GEOGRAPHIC	11	Logical representing whether or not the geocoder should fall back to a geographic centroid when other options fail.
GEOCODE_FALLBACK_POSTAL	12	Logical representing whether or not the geocoder should fall back to a postal centroid when other options fail.
GEOCODE_OFFSET_CENTER	13	Float value representing the distance from the center of the road that the point is returned.
GEOCODE_OFFSET_CENTER_UNITS	14	String value representing the units of the center of the road values.
GEOCODE_OFFSET_END	15	Float value representing the distance from the end of the road that the point is returned.
GEOCODE_OFFSET_END_UNITS	16	String value representing the units of the offset from end of street value
GEOCODE_MIXED_CASE	17	Logical representing whether MapInfo Professional should format the strings returned in mixed case or leave them as uppercase. This option may not be available for all countries. The option uses a country specific algorithm that has knowledge of what address parts and what items should be capitalized and what should be made lower case.

attribute Value	ID	GeoCodeInfo( ) Return Value
GEOCODE_RESULT_MARK_MULTIPLE	18	Logical representing whether MapInfo Professional should change the result code returned from the server by adding an indicator to the result code that the result was based on an arbitrary choice between multiple close matches. This flag only affects the behavior under the following circumstances: **1.** The geocoding was not interactive so no possibility of presenting the candidates dialog was possible. **1.** The non-interactive command flag was to pick the first candidate returned rather than none. (see Geocode command “First”). This forces MapInfo Professional to pick one of the candidates. **2.** The actual request returned more than one close match for a particular record.
GEOCODE_COUNT_GEOCODED	19	Integer value representing the number of records geocoded during the last operation.
GEOCODE_COUNT_NOTGEOCODED	20	Integer value representing the number of records not geocoded during the last operation.
GEOCODE_UNABLE_TO_CONVERT_DATA	21	Logical representing whether a column was not updated during the last operation because of a data type problem. The case where this occurs is when integer columns are erroneously specified for non-numeric postal codes.
GEOCODE_PASSTHROUGH	100	Integer specifying the number of passthrough items set on this connection. There are two items for each pair. This value is used to know when to stop the enumeration of these values without error.

attribute Value	ID	GeoCodeInfo( ) Return Value
GEOCODE_PASSTHROUGH + *n*		String values alternately representing name and value for each passthrough pair. *n* is valid up to the value returned via GEOCODE_INFO_PASSTHROUGH.
GEOCODE_MAX_BATCH_SIZE	22	Integer value representing the maximum number of records (for example, addresses) that the server will permit to be sent to the service at one time.

Example

The following MapBasic snippet will print the Envinsa Location Utility Constraints to the message window in MapInfo Professional:

```
Include "MapBasic.Def"
declare sub main
sub main
dim iConnect as integer

Open Connection Service Geocode Envinsa
     URL
"http://envinsa_server:8066/LocationUtility/services/LocationUtility"
     User "john"
     Password "green"
     into variable iConnect

Print "Geocode Max Batch Size: " +
GeoCodeInfo(iConnect,GEOCODE_MAX_BATCH_SIZE)
end sub
```

See Also:

Open Connection statement, **Set Connection Geocode statement**

Get statement

Purpose

Reads from a file opened in Binary or Random access mode.

Syntax

Get [**#**] *filenum*, [*position*], *var_name*

filenum is the number of a file opened through an **Open File statement**.

position is the file position to read from.

var_name is the name of a variable where MapBasic will store results.

Description

The **Get** statement reads from an open file. The behavior of the **Get** statement and the set of parameters which it expects are affected by the options specified in the preceding **Open File statement**.

If the **Open File statement** specified **Random** file access, the **Get** statement's Position clause can be used to indicate which record of data to read. When the file is opened, the file position points to the first record of the file (record 1). A **Get** automatically increments the file position, and thus the Position clause does not need to be used if sequential access is being performed. However, you can use the Position clause to set the record position before the record is read.

If the **Open File statement** specified **Binary** file access, one variable can be read at a time. What data is read depends on the byte-order format of the file and the *var_name* variable being used to store the results. If the variable type is integer, then 4 bytes of the binary file will be read, and converted to a MapBasic variable. Variables are stored the following way:

Variable Type	Storage In File
Logical	One byte, either 0 or non-zero.
SmallInt	Two byte integer.
Integer	Four byte integer.
Float	Eight byte IEEE format.
String	Length of string plus a byte for a 0 string terminator.
Date	Four bytes: SmallInt year, byte month, byte day.
Other data types	Cannot be read.

With Binary file access, the *position* parameter is used to position the file pointer to a specific offset in the file. When the file is opened, the *position* is set to one (the beginning of the file). As a **Get** is performed, the position is incremented by the same amount read. If the Position clause is not used, the **Get** reads from where the file pointer is positioned.

The **Get** statement requires two commas, even if the optional *position* parameter is omitted.

If a file was opened in Binary mode, the **Get** statement cannot specify a variable-length string variable; any string variable used in a **Get** statement must be fixed-length.

See Also:

Open File statement, **Put statement**

GetCurrentPath( ) function

Purpose

Returns the path of a special MapInfo Professional directory defined initially in the Preferences dialog box to access specific MapInfo files. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

GetCurrentPath$(*current_path_id*)

current_path_id is one of the following values:

```
PREFERENCE_PATH_TABLE (0)
PREFERENCE_PATH_WORKSPACE (1)
PREFERENCE_PATH_MBX (2)
PREFERENCE_PATH_IMPORT (3)
PREFERENCE_PATH_SQLQUERY (4)
PREFERENCE_PATH_THEMETHEMPLATE (5)
PREFERENCE_PATH_MIQUERY (6)
PREFERENCE_PATH_NEWGRID (7)
PREFERENCE_PATH_CRYSTAL (8)
PREFERENCE_PATH_GRAPHSUPPORT (9)
PREFERENCE_PATH_REMOTETABLE (10)
PREFERENCE_PATH_SHAPEFILE (11)
PREFERENCE_PATH_WFSTABLE (12)
PREFERENCE_PATH_WMSTABLE (13)
```

Return Value

String

Description

Given the ID of a special MapInfo Preference directory, the GetCurrentPath$( ) function returns the path of the directory. An example of a special MapInfo directory is the default location to which MapInfo Professional writes out new native MapInfo tables.

Example

Copy this example into the MapBasic window for a demonstration of this function.

```
include "mapbasic.def"
declare sub main
sub main
dim sMiPrfFile as string
sMiPrfFile = GetCurrentPath$( PREFERENCE_PATH_WORKSPACE)
Print sMiPrfFile
end sub
```

See Also:

GetPreferencePath$( ) function

GetDate( ) function

Purpose

Returns the Date component of a DateTime. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

```
GetDate( DateTime )
```

Return Value

Date

Example

Copy this example into the MapBasic window for a demonstration of this function.

```
dim dtX as datetime
dim Z as date
dtX = "03/07/2007 12:09:09.000 AM"
Z = GetDate(dtX)
Print FormatDate$(Z)
```

GetFolderPath$( ) function

Purpose

Returns the path of a special MapInfo Professional or Windows directory. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

GetFolderPath$(*folder_id*)

folder_id is one of the following values:

```
FOLDER_MI_APPDATA (-1)
FOLDER_MI_LOCAL_APPDATA (-2)
FOLDER_MI_PREFERENCE (-3)
FOLDER_MI_COMMON_APPDATA (-4)
FOLDER_APPDATA (26)
FOLDER_LOCAL_APPDATA (28)
FOLDER_COMMON_APPDATA (35)
FOLDER_COMMON_DOCS (46)
FOLDER_MYDOCS (5)
FOLDER_MYPICS (39)
```

Return Value

String

Description

Given the ID of a special MapInfo or Windows directory, **GetFolderPath$( )** function returns the path of the directory. An example of a special Windows directory is the My Documents directory. An example of a special MapInfo directory is the preference directory; the default location to which MapInfo Professional writes out the preference file.

The location of many of these directories varies between versions of Windows. They can also vary depending on which user is logged in. Note that FOLDER_MI_APPDATA (-1), FOLDER_MI_LOCAL_APPDATA (-2), and FOLDER_MI_COMMON_APPDATA (-4) may not exist. Before attempting to access those directories, test for their existence by using **FileExists( ) function**. FOLDER_MI_PREFERENCE (-3) always exists.

IDs beginning in FOLDER_MI return the path for directories specific to MapInfo Professional. The rest of the IDs return the path for Windows directories and correspond to the IDs defined for WIN32 API function SHGetFolderPath. The most common of these IDs have been defined for easy use in MapBasic applications. Any ID valid to SHGetFolderPath will work with **GetFolderPath$( )**.

Example

```
include "mapbasic.def"
declare sub main
sub main
dim sMiPrfFile as string
sMiPrfFile = GetFolderPath$(FOLDER_MI_PREFERENCE)
Print sMiPrfFile
end subet128
```

See Also:

LocateFile$( ) function

GetGridCellValue( ) function

Purpose:

Determines the value of a grid cell if the cell is non-null.

Syntax:

GetGridCellValue(*table_id*, *x_pixel*, *y_pixel*)

table_id is a string representing a table name, a positive integer table number, or 0 (zero). The table must be a grid table.

x_pixel is the integer number of the X coordinate of the grid cell. Pixel numbers start at 0. The maximum pixel value is the (pixel_width-1), determined by calling

`RasterTableInfo(...RASTER_TAB_INFO_WIDTH).`

y_pixel is the integer number of the Y coordinate of the grid cell. Pixel numbers start at 0. The maximum pixel value is the (pixel_height-1), determined by calling

```
RasterTableInfo(...RASTER_TAB_INFO_HEIGHT).
```

Return Value

A Float is returned, representing the value of a specified cell in the table if the cell is non-null. The IsGridCellNull() function should be used before calling this function to determine if the cell is null or if it contains a value.

GetMetadata$( ) function

Purpose

Retrieves metadata from a table. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

GetMetadata$(*table_name*, *key_name*)

table_name is the name of an open table, specified either as an explicit table name (for example, World) or as a string representing a table name (for example, “World”).

key_name is a string representing the name of a metadata key.

Return Value

String, up to 239 bytes long. If the key does not exist, or if there is no value for the key, MapInfo Professional returns an empty string.

Description

This function returns a metadata value from a table. For more information about querying a table's metadata, see **Metadata statement**, or see the *MapBasic User Guide*.

Example

If the Parcels table has a metadata key called “\Copyright” then the following statement reads the key's value:

```
Print GetMetadata$(Parcels, "\Copyright")
```

See Also:

Metadata statement

GetPreferencePath$( ) function

Purpose

Returns the path of a special MapInfo Professional directory defined in the Preferences dialog to access specific MapInfo files. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

GetPreferencePath$(*preference_path_id*)

preference_path_id is one of the following values:

- PREFERENCE_PATH_TABLE (0)
- PREFERENCE_PATH_WORKSPACE (1)
- PREFERENCE_PATH_MBX (2)
- PREFERENCE_PATH_IMPORT (3)
- PREFERENCE_PATH_SQLQUERY (4)
- PREFERENCE_PATH_THEMETHEMPLATE (5)
- PREFERENCE_PATH_MIQUERY (6)
- PREFERENCE_PATH_NEWGRID (7)
- PREFERENCE_PATH_CRYSTAL (8)
- PREFERENCE_PATH_GRAPHSUPPORT (9)
- PREFERENCE_PATH_REMOTETABLE (11)
- PREFERENCE_PATH_WFSTABLE (12)
- PREFERENCE_PATH_WMSTABLE (13)

Return Value

String

Description

Given the ID of a special MapInfo Preference directory, the GetPreferencePath$( ) function returns the path of the directory. An example of a special MapInfo directory is the default location to which MapInfo Professional writes out new native MapInfo tables.

Example

```
include "mapbasic.def"
declare sub main
sub main
dim sMiPrfFile as string
sMiPrfFile = GetPreferencePath$( PREFERENCE_PATH_WORKSPACE)
Print sMiPrfFile
end sub
```

See Also:

LocateFile$( ) function

GetSeamlessSheet( ) function

Purpose

Prompts the user to select one sheet from a seamless table, and then returns the name of the chosen sheet. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

GetSeamlessSheet(*table_name*)

table_name is the name of a seamless table that is open.

Return Value

String, representing a table name (or an empty string if user cancels).

Description

This function displays a dialog box listing all of the sheets that make up a seamless table. If the user chooses a sheet and clicks **OK**, this function returns the table name the user selected. If the user cancels, this function returns an empty string.

Example

```
Sub Browse_A_Table(ByVal s_tab_name As String)
   Dim s_sheet As String

   If TableInfo(s_tab_name, TAB_INFO_SEAMLESS) Then
      s_sheet = GetSeamlessSheet(s_tab_name)
      If s_sheet <> "" Then
         Browse * From s_sheet
      End If
   Else
      Browse * from s_tab_name
   End If

End Sub
```

See Also:

Set Table statement, TableInfo( ) function

GetTime() function

Purpose

Returns the Time component of a DateTime. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

```
GetTime( DateTime )
```

Return Value

Time

Example

Copy this example into the MapBasic window for a demonstration of this function.

```
dim dtX as datetime
dim Z as time
dtX = "03/07/2007 12:09:09.000 AM"
Z = GetTime(dtX)
Print FormatTime$(Z,"hh:mm:ss.fff tt")
```

Global statement

Purpose

Defines one or more global variables.

Syntax

Global *var_name* [, *var_name*...] **As** var_type
   [, *var_name*...] **As** *var_type*...]

var_name is the name of a global variable to define.

var_type is integer, float, date, logical, string, or a custom variable Type.

Description

A **Global** statement defines one or more global variables. **Global** statements may only appear outside of a sub procedure.

The syntax of the **Global** statement is identical to the syntax of the **Dim statement**; the difference is that variables defined through a **Global** statement are global in scope, while variables defined through a **Dim statement** are local. A local variable may only be examined or modified by the sub procedure which defined it, whereas any sub procedure in a program may examine or modify any global variable. A sub procedure may define local variables with names which coincide with the names of global variables. In such a case, the sub procedure's own local variables take precedence (for example, within the sub procedure, any references to the variable name will utilize the local

variable, not the global variable by the same name). Global array variables may be re-sized with the **ReDim statement**. Windows, global variables are “visible” to other Windows applications through DDE conversations.

Example

```
Declare Sub testing( )
Declare Sub Main( )
Global gi_var As Integer
Sub Main( )
   Call testing
   Note Str$(gi_var) ' this displays "23"
End Sub

Sub testing( )
   gi_var = 23
End Sub
```

See Also:

Dim statement, **ReDim statement**, **Type statement**, **UBound( ) function**

Goto statement

Purpose

Jumps to a different spot (in the same procedure), identified by a label.

Restrictions

You cannot issue a **Goto** statement through the MapBasic window.

Syntax

Goto *label*

label is a label appearing elsewhere in the same procedure.

Description

The **Goto** statement performs an unconditional jump. Program execution continues at the statement line identified by the *label*. The *label* itself should be followed by a colon; however, the *label* name should appear in the **Goto** statement without the colon.

Generally speaking, the **Goto** statement should not be used to exit a loop prematurely. The **Exit Do statement** and **Exit For statement** provide the ability to exit a loop. Similarly, you should not use a **Goto** statement to jump into the body of a loop.

A **Goto** statement may only jump to a *label* within the same procedure.

Example

```
Goto endproc

...

endproc: End Program
```

See Also:

Do Case…End Case statement, **Do…Loop statement**, **For…Next statement**, **OnError statement**, **Resume statement**

Graph statement

Purpose

Opens a new Graph window. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Graph
   label_column, expr ["label_text"][ , ... ]
   From table
   [ Position ( x, y ) [ Units paperunits ] ]
   [ Width window_width [ Units paperunits ] ]
   [ Height window_height [ Units paperunits ] ]
   [ Min | Max ]
   [ Using template_file [ Restore ] [ Series In Columns ] ]
```

label_column is the name of the column to use for labelling the y-axis.

expr is an expression providing values to be graphed.

label_text is the text that displays for each label column instead of the column name

table is the name of an open table.

paperunits is the name of a paper unit (for example, “in”).

x, *y* specifies the position of the upper left corner of the Grapher, in paper units.

window_width and *window_height* specify the size of the Grapher, in paper units.

template_file is a valid graph template file.

Description

If the **Using** clause is present and *template_file* specifies a valid graph template file, then a graph is created based on the specified template file. Otherwise a 5.0 graph is created. If the **Restore** clause is included, then title text in the template file is used in the graph window. Otherwise default text is used for each title in the graph. The **Restore** keyword is included when writing the Graph command to a workspace, so when the workspace is opened the title text is restored exactly as is was when

the workspace was saved. The **Restore** keyword is not used in the Graph command constructed by the Create Graph wizard, so the default text is used for each title. If **Series In Columns** is included, then the graph series are based on the table columns. Otherwise the series are based on the table rows.

The **Graph** statement adds a new Grapher window to the screen, displaying the specified table. The graph will appear as a rotated bar chart; subsequent **Set Graph statement**s can re-configure the specifics of the graph (for example, the graph rotation, graph type, title, etc.).

MapInfo Professional 's **Window > Graph** dialog box is limited in that it only allows the user to choose column names to graph. MapBasic's **Graph** statement, however, is able to graph full expressions which involve column names. Similarly, although the Graph dialog box only allows the user to choose four columns to graph, the **Graph** statement can construct a graph with up to 255 columns.

If the **Graph** statement includes the optional **Max** keyword, the resultant Grapher window is maximized, taking up all of the screen space available to MapInfo Professional. Conversely, if the **Graph** statement includes the **Min** keyword, the window is minimized.

Example (5.5 and later graphs)

```
Graph State_Name, Pop_1980, Pop_1990, Num_Hh_80 From States Using
"C:\Program Files\MapInfo\GRAPHSUPPORT\Templates\Column\Percent.3tf"
Graph City, Tot_hu, Tot_pop From City_125 Using "C:\Program
Files\MapInfo\GRAPHSUPPORT\Templates\Bar\Clustered.3tf" Series In Columns
```

Example (pre-5.5 graphs)

```
Graph Country, Population From Selection
```

See Also:

Set Graph statement

GridTableInfo( ) function

Purpose

Returns information about a grid table.

Syntax

GridTableInfo(table_id, attribute)

table_id is a string representing a table name, a positive integer table number, or 0 (zero). The table must be a grid table.

attribute is an integer code indicating which aspect of the grid table to return.

Return Value

String, SmallInt, Integer or Logical, depending on the attribute parameter specified.

The attribute parameter can be any value from the table below. Codes in the left column (for example, GRID_TAB_INFO_) are defined in MAPBASIC.DEF.

attribute code	ID	GridTableInfo() returns:
GRID_TAB_INFO_MIN_VALUE	1	Float result, representing the minimum grid cell value in the file
GRID_TAB_INFO_MAX_VALUE	2	Float result, representing the maximum grid cell value in the file
GRID_TAB_INFO_HAS_HILLSHADE	3	Logical result, TRUE if the grid file has hillshade/relief shade information. This flag does not depend on whether the file is displayed using the hillshading.

GroupLayerInfo function

Purpose

This function returns information about a specific group layer in the map.

Syntax

GroupLayerInfo (*map_window_id*, *group_layer_id*, *attribute*)

map_window_id is a Map window identifier.

group_layer_id is the number of a group layer in the Map window (for example, 1 for the top group layer) or a name of a group layer in the map. To determine the number of group layers in a Map window, call the **MapperInfo( ) function**.

attribute is a code indicating the type of information to return; see table below.

Return Value

Depends on the *attribute* parameter.

Description

The attributes are:

Value of window_id, attribute	ID	Description
GROUPLAYER_INFO_NAME	1	Returns a string value, which is the name of the group layer.
GROUPLAYER_INFO_LAYERLIST_ID	2	Returns a numeric value, the ID of the group layer in the layer list (position of the group layer in the layer list).
GROUPLAYER_INFO_DISPLAY	3	Returns the boolean value • GROUPLAYER_INFO_DISPLAY_ON (true if the layer is visible (0)) • GROUPLAYER_INFO_DISPLAY_OFF (false if the layer is not visible (non-zero))
GROUPLAYER_INFO_LAYERS	4	Returns the count of graphical layers in the group. It will ignore group layers but include all nested graphical layers.
GROUPLAYER_INFO_ALL_LAYERS	5	Returns the count of layers and group layers (includes all nested layers and group layers).
GROUPLAYER_INFO_TOPLEVEL_LAYERS	6	Returns the count of graphical or group layers at the top level of the group's layer list.
GROUPLAYER_INFO_PARENT_GROUP_ID	7	Returns the group layer ID of the immediate group containing this group, will return zero (0) if group layer is in the top level list.

Group layer ID's are from zero (0) to n, where n is the number of group layers in the list and zero (0) refers to top level, or "root" of the layer list. All the group layer info attributes will apply to the root of the list with the exception of GROUPLAYER_INFO_DISPLAY (3). GROUPLAYER_INFO_NAME (1) will return the map's name (same as its window title). The cosmetic layer will be included in any of the attributes that count graphical layers.

Specifying a map window ID of zero (0) returns the name of the map window, and returns the name of the Cosmetic Layer as "cosmetic1", "cosmetic2".

See Also:

LayerInfo( ) function, **MapperInfo( ) function**

HomeDirectory$( ) function

Purpose

Returns a string indicating the user's home directory path. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

HomeDirectory$( )

Return Value

String

Description

The **HomeDirectory$( )** function returns a string which indicates the user's home directory path.

The significance of a home directory path depends on the hardware platform on which the user is running. The table below summarizes the platform-dependent home directory path definitions.

Environment	Definition of "Home Directory"
Windows	The directory path to the user's Windows directory.

Example

```
Dim s_home_dir As String
s_home_dir = HomeDirectory$( )
```

See Also:

ApplicationDirectory$( ) function, **ProgramDirectory$( ) function**, **SystemInfo( ) function**

HotlinkInfo function

Purpose

Returns information about a HotLink definition in a map layer. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

HotlinkInfo (*map_window_id, layer_number, hotlink_number, attribute*)

map_window_id is a Map window identifier.

layer_number is the number of a layer in the current Map window (for example, 1 for the top layer); to determine the number of layers in a Map window, call the **MapperInfo( ) function**.

hotlink_number - the index of the hotlink definition being queried. The first hotlink definition in a layer has index of 1.

attribute - the following attribute values are allowed:

Hotlink Name	ID	Description
HOTLINK_INFO_EXPR	1	Returns the filename expression for this hotlink definition.
HOTLINK_INFO_MODE	2	Returns the mode for this hotlink definition, one of the following predefined values: • HOTLINK_MODE_LABEL (0) • HOTLINK_MODE_OBJ (1) HOTLINK_MODE_BOTH (2)
HOTLINK_INFO_ENABLED	3	Returns TRUE if the relative path option is on for this hotlink definition.
HOTLINK_INFO_ENABLED	4	Returns TRUE if this hotlink definition is enabled.

See Also:

Set Map statement, **LayerInfo( ) function**,

Hour function

Purpose

Returns the hour component of a Time. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

```
Hour (Time)
```

Return Value

Number

Example

Copy this example into the MapBasic window for a demonstration of this function.

```
dim Z as time
dim iHour as integer
Z = CurDateTime()
iHour = Hour(Z)
Print iHour
```

If…Then statement

Purpose

Decides which block of statements to execute (if any), based on the current value of one or more expressions.

Syntax

```
If if_condition Then
   if_statement_list
   [ ElseIf elseif_condition Then
   elseif_statement_list ]
   [ ElseIf ... ]
   [ Else
    else_statement_list ]
End If
```

condition is a condition which will evaluate to TRUE or FALSE

statement_list is a list of zero or more statements.

Restrictions

You cannot issue an **If…Then** statement through the MapBasic window.

Description

The **If…Then** statement allows conditional execution of different groups of statements.

In its simplest form, the **If** statement does not include an **ElseIf** clause, nor an **Else** clause:

```
If if_condition Then
    if_statement_list
End If
```

With this arrangement, MapBasic evaluates the *if_condition* at run-time. If the *if_condition* is TRUE, MapBasic executes the *if_statement_list*; otherwise, MapBasic skips the *if_statement_list*.

An **If** statement may also include the optional **Else** clause:

```
If if_condition Then
    if_statement_list
Else
    else_statement_list
End If
```

With this arrangement, MapBasic will either execute the *if_statement_list* (if the condition is TRUE) or the *else_statement_list* (if the condition is FALSE).

Additionally, an **If** statement may include one or more **ElseIf** clauses, following the **If** clause (and preceding the optional **Else** clause):

```
If if_condition Then
    if_statement_list
```

```
ElseIf elseif_condition Then
    elseif_statement_list
Else
    else_statement_list
End If
```

With this arrangement, MapBasic tests a series of two or more conditions, continuing until either one of the conditions turns out to be TRUE or until the **Else** clause or the **End If** is reached. If the *if_condition* is TRUE, MapBasic will perform the *if_statement_list*, and then jump down to the statement which follows the **End If**. But if that condition is FALSE, MapBasic then evaluates the *else_if_condition*; if that condition is TRUE, MapBasic will execute the *elseif_statement_list*.

An **If** statement may include two or more **ElseIf** clauses, thus allowing you to test any number of possible conditions. However, if you are testing for one out of a large number of possible conditions, the **Do Case…End Case statement** is more elegant than an **If** statement with many **ElseIf** clauses.

Example

```
Dim today As Date
Dim today_mon, today_day, yearcount As Integer

today = CurDate( ) ' get current date
today_mon = Month(today) ' get the month value
today_day = Day(today) ' get the day value (1-31)

If today_mon = 1 And today_day = 1 Then
   Note "Happy New Year!"
   yearcount = yearcount + 1
ElseIf today_mon = 2 And today_day = 14 Then
   Note "Happy Valentine's Day!"
ElseIf today_mon = 12 And today_day = 25 Then
   Note "Merry Christmas!"
Else
   Note "Good day."
End If
```

See Also:

Do Case…End Case statement

Import statement

Purpose

Creates a new MapInfo Professional table by importing an exported file, such as a GML or DXF file. You can issue this statement from the MapBasic Window in MapInfo Professional.

See **Importing MIF/MID, PICT, or MapInfo for DOS Files**, **Importing DXF Files**, **Importing GML Files**, or **Importing GML 2.1 Files**.

Importing MIF/MID, PICT, or MapInfo for DOS Files

Syntax

```
Import file_name
   [ Type file_type ]
   [ Into table_name ]
   [ Overwrite ]
```

file_name is a string that specifies the name of the file to import.

file_type is a string that specifies the import file format (MIF, MBI, MMI, IMG, or PICT).

table_name specifies the name of the new table to create.

Description

The **Import** statement creates a new MapInfo table by importing the contents of an existing file.

> ⓘ To create a MapInfo table based on a spreadsheet or database file, use the **Register Table statement**, not the **Import** statement.

The optional **Type** clause specifies the format of the file you want to import. The **Type** clause can take one of the following forms:

Type clause	File Format Specified
Type “DXF”	DXF file (a format supported by CAD packages, such as AutoCAD). See **Importing DXF Files**.
Type “MIF”	MIF/MID file pair, created by exporting a MapInfo table.
Type “MBI”	MapInfo Boundary Interchange, created by MapInfo for DOS.
Type “MMI”	MapInfo Map Interchange, created by MapInfo for DOS.
Type “IMG”	MapInfo Image file, created by MapInfo for DOS.
Type “GML”	GML files. See **Importing GML Files**.
Type “GML21”	GML 2.1 files. See **Importing GML 2.1 Files**.

If you omit the **Type** clause, MapInfo Professional assumes that the file's extension indicates the file format. For example, a file named “PARCELS.DXF” is assumed to be a DXF file. (For more about DXF, see **Importing DXF Files**.)

The **Into** clause lets you override the name and location of the MapInfo table that is created. If no Into clause is specified, the new table is created in the same directory location as the original file, with a corresponding file name. For example, on Windows, if you import the text file “WORLD.MIF”, the new table's default name is “WORLD.TAB”.

If you include the optional **Overwrite** keyword, MapInfo Professional creates a new table, regardless of whether a table by that name already exists; the new table replaces the existing table. If you omit the **Overwrite** keyword, and the table already exists, MapInfo Professional does not overwrite the table.

Example

Sample importing using current MapInfo style:

```
Import "D:\midata\GML\test.gml" Type "GML" layer "TopographicLine" style
auto off Into "D:\midata\GML\test_TopographicLine.TAB" Overwrite
```

The following example imports a MIF (MapInfo Interchange Format) file:

```
Import "WORLD.MIF"
   Type "MIF"
   Into "world_2.tab"

   Map From world_2
```

Importing DXF Files

Syntax

```
Import file_name
   [ Type "DXF" ]
   [ Into table_name ]
   [ Overwrite ]
   [ Warnings { On | Off } ]
   [ Preserve
      [ AttributeData ] [ Preserve ] [ Blocks As MultiPolygonRgns ] ]
   [ CoordSys... ]
   [ Autoflip ]
   [ Transform
      ( DXF_x1, DXF_y1 ) ( DXF_x2, DXF_y2 )
      ( MI_x1, MI_y1 ) ( MI_x2, MI_y2 ) ]
   [ Read
      [ Integer As Decimal ] [ Read ] [ Float As Decimal ] ]
   [ Store [ Handles ] [ Elevation ] [ VisibleOnly ] ]
   [ Layer DXF_layer_name
      [ Into table_name ]
      [ Preserve
         [ AttributeData ] [ Preserve ] [ Blocks As MultiPolygonRgns ] ]
      ]
   [ Layer... ]
```

file_name is a string that specifies the name of the file to import.

table_name specifies the name of the new table to create.

DXF_x1, *DXF_y1*, etc. are numbers that represent coordinates in the DXF file.

MI_x1, *MI_y1*, etc. are numbers that represent coordinates in the MapInfo table.

DXF_layer_name is a string representing the name of a layer in the DXF file.

Description

If you import a DXF file, the **Import** statement can include the following DXF-specific clauses.

> ⓘ The order of the clauses is important; placing the clauses in the wrong order can cause compilation errors.

Warnings On or Warnings Off – Controls whether warning messages are displayed during the import operation. By default, warnings are off.

Preserve AttributeData – Include this clause if you want MapInfo Professional to preserve the attribute data from the DXF file.

Preserve Blocks As MultiPolygonRgns – Include this clause if you want MapInfo Professional to store all of the polygons from a DXF block record into one multiple-polygon region object. If you omit this clause, each DXF polygon becomes a separate MapInfo Professional region object.

CoordSys – Controls the projection and coordinate system of the table. For details, see **CoordSys clause**.

Autoflip – Include this option if you want the map's x-coordinates to be flipped around the center line of the map. This option is only allowed if you specify a non-Earth coordinate system.

Transform – Specifies a coordinate transformation. In the **Transform** clause, you specify the minimum and maximum x- and y-coordinates of the imported file, and you specify the minimum and maximum coordinates that you want to have in the MapInfo table.

Read Integer As Decimal – Include this clause if you want to store whole numbers from the DXF file in a decimal column in the new table. This clause is only allowed when you include the **Preserve AttributeData** clause.

Read Float As Decimal – Include this clause if you want to store floating-point numbers from the DXF file in a decimal column in the new table. This clause is only allowed when you include the **Preserve AttributeData** clause.

Store [Handles] [Elevation] [VisibleOnly] – If you include **Handles**, the MapInfo table stores handles (unique ID numbers of objects in the drawing) in a column called _DXFHandle. If you include **Elevation**, MapInfo Professional stores each object's center elevation in a column called _DXFElevation. (For lines, MapInfo Professional stores the elevation at the center of the line; for regions, MapInfo Professional stores the average of the object's elevation values.) If you include **VisibleOnly**, MapInfo Professional ignores invisible objects.

Layer clause – If you do not include any **Layer** clauses, all objects from the DXF file are imported into a single MapInfo table. If you include one or more **Layer** clauses, each DXF layer that you name becomes a separate MapInfo table.

If your DXF file contains multiple layers, and if your **Import** statement includes one or more **Layer** clauses, MapInfo Professional only imports the layers that you name. For example, suppose your DXF file contains four layers (layers 0, 1, 2, and 3). The following **Import** statement imports all four layers into a single MapInfo table:

```
Import "FLOORS.DXF"
   Into "FLOORS.TAB"
   Preserve AttributeData
```

The following statement imports layers 1 and 3, but does not import layers 0 or 2:

```
Import "FLOORS.DXF"
   Layer "1"
      Into "FLOOR_1.TAB"
      Preserve AttributeData
   Layer "3"
      Into "FLOOR_3.TAB"
      Preserve AttributeData
```

Importing GML Files

Syntax

```
Import file_name
   [ Type  "GML"  ]
   [ Layer layer_name ]
   [ Into  table_name  ]
   [ Style Auto [ On | Off ] ]
```

file_name is a string that specifies the name of the file to import.

layer_name is a string representing the name of a layer in the GML file.

table_name specifies the name of the new table to create.

Description

Type is “GML” for GML files.

MapInfo Professional supports importing OSGB (Ordnance Survey of Great Britain) GML files. Cartographic Symbol, Topographic Point, Topographic Line, Topographic Area, and Boundary Line are supported; Cartographic Text is not supported. Topographic Area can be distributed in two forms; MapInfo Professional supports the non-topological form. If the files contains XLINKS, MapInfo Professional only imports attribute data, and does not import spatial objects. These XLINKs are stored in the GML file as “xlink:href=”. If topological objects are included in the file, a warning displays indicating that spatial objects cannot be imported. Access the Browser view to see the display of attribute data.

Example

Sample importing using GML style:

```
Import "D:\midata\GML\est.gml" Type "GML" layer "LandformArea" style auto
on Into "D:\midata\GML\est_LandformArea.TAB" Overwrite
```

Importing GML 2.1 Files

Syntax

```
Import file_name
   [ Type "GML21" ]
   [ Layer layer_name]
   [ Into table_name ]
   [ Overwrite ]
   [ CoordSys... ]
```

file_name is the name of the GML 2.1 file to import.

layer_name is the name of the GML layer.

table_name is the MapInfo table name.

Description

Type is “GML21” for GML 2.1 files.

Overwrite causes the TAB file to be automatically overwritten. If **Overwrite** is not specified, an error will result if the TAB file already exists.

The **Coordsys** clause is optional. If the GML file contains a supported projection and the **Coordsys** clause is not specified, the projection from the GML file will be used. If the GML file contains a supported projection and the **Coordsys** clause is specified, the projection from the **Coordsys** clause will be used. If the GML file does not contain a supported projection, the **Coordsys** clause must be specified.

If the **Coordsys** clause does not match the projection of the GML file, your data may not import correctly. The coordinate system must match the coordinate system of the data in the GML file. It will not transform the data from one projection to another.

Example

Sample importing using GML21 style:

```
Import "D:\midata\GML\GML2.1\mi_usa.xml" Type "GML21" layer "USA" Into
"D:\midata\GML\GML2.1\mi_usa_USA.TAB" Overwrite CoordSys Earth Projection
1, 104
```

See Also:

Export statement

Include statement

Purpose

Incorporates the contents of a separate text file as part of a MapBasic program. Issuing this statement from the MapBasic Window in MapInfo Professional does not work.

Syntax

Include "*filename*"

filename is the name of an existing text file.

Restrictions

You cannot issue an **Include** statement through the MapBasic window.

Description

When MapBasic is compiling a program file and encounters an **Include** statement, the entire contents of the included file are inserted into the program file. The file specified by an **Include** statement should be a text file, containing only legitimate MapBasic statements.

If the *filename* parameter does not specify a directory path, and if the specified file does not exist in the current directory, the MapBasic compiler looks for the file in the program directory. This arrangement allows you to leave standard definitions files, such as MAPBASIC.DEF, in one directory, rather than copying the definitions files to the directories where you keep your program files.

The most common use of the **Include** statement is to include the file of standard MapBasic definitions, MAPBASIC.DEF. This file, which is provided with MapBasic, defines a number of important identifiers, such as TRUE and FALSE.

Whenever you change the contents of a file that you use through an **Include** statement, you should then recompile any MapBasic programs which Include that file.

Example

```
Include "MAPBASIC.DEF"
```

Input # statement

Purpose

Reads data from a file, and stores the data in variables.

Syntax

Input # *filenum*, *var_name* [, *var_name*...]

filenum is the number of a file opened through the **Open File statement**.

var_name is the name of a variable.

Description

The **Input #** statement reads data from a file which was opened in a sequential mode (for example, INPUT mode), and stores the data in one or more MapBasic variables.

The **Input #** statement reads data (up to the next end-of-line) into the variable(s) indicated by the *var_name* parameter(s). MapInfo Professional treats commas and end-of-line characters as field delimiters. To read an entire line of text into a single string variable, use **Line Input statement**.

MapBasic automatically converts the data to the type of the variable(s). When reading data into a string variable, the **Input #** statement treats a blank line as an empty string. When reading data into a numeric variable, the **Input #** statement treats a blank line as a zero value.

After issuing an **Input #** statement, call the **EOF( ) function** to determine if MapInfo Professional was able to read the data. If the input was successful, the **EOF( ) function** returns FALSE; if the end-of-file was reached before the input was completed, the **EOF( ) function** returns TRUE.

For an example of the **Input #** statement, see the sample program NVIEWS (Named Views).

The following data types are not available with the **Input #** statement:

- Alias
- Pen
- Brush
- Font
- Symbol
- Object

See Also:

EOF( ) function, **Line Input statement**, **Open File statement**, **Write # statement**

Insert statement

Purpose

Appends new rows to an open table. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Insert Into table
   [ ( columnlist ) ]
   { Values ( exprlist ) | Select columnlist From table }
```

table is the name of an open table.

columnlist is a list of column expressions, comma-separated.

exprlist is a list of one or more expressions, comma-separated.

Description

The **Insert** statement inserts new rows into an open table. There are two main forms of this statement, allowing you to either add one row at a time, or insert groups of rows from another table (via the **Select** clause). In either case, the number of column values inserted must match the number of columns in the column list. If no column list is specified, all fields are assumed. Note that you must use a **Commit Table statement** if you want to permanently save newly-inserted records to disk.

If you know exactly how many columns are in the table you are modifying, and if you have values to store in each of those columns, then you do not need to specify the optional *columnlist* clause.

In the following example, we know that the table has four columns (Name, Address, City, and State), and we provide MapBasic with a value for each of those columns.

```
Insert Into customers
   Values ("Mary Ryan", "23 Main St", "Dallas", "TX")
```

The preceding statement would generate an error at run-time if it turned out that the table had fewer than (or more than) four columns. In cases where you do not know exactly how many columns are in a table or the exact order in which the columns appear, you should use the optional *columnlist* clause.

Examples

The following example inserts a new row into the customer table, while providing only one column value for the new row; thus, all other columns in the new row will initially be blank. Here, the one value specified by the **Values** clause will be stored in the “Name” column, regardless of how many columns are in the table, and regardless of the position of the “Name” column in the table structure.

```
Insert Into customers (Name)
   Values ("Steve Harris")
```

The following statement creates a point object and inserts the object into a new row of the Sites table. Note that Obj is a special column name representing the table's graphical objects.

```
Insert Into sites (Obj)
   Values ( CreatePoint(-73.5, 42.8) )
```

The following example illustrates how the **Insert** statement can append records from one table to another. In this example, we assume that the table NY_ZIPS contains ZIP Code boundaries for New York state, and NJ_ZIPS contains ZIP Code boundaries for New Jersey. We want to put all ZIP Code boundaries into a single table, for convenience's sake (since operations such as Find can only work with one table at a time).

Accordingly, the **Insert** statement below appends all of the records from the New Jersey table into the New York table.

```
Insert Into NY_ZIPS
   Select * From NJ_ZIPS
```

In the following example, we select the graphical objects from the table World, then insert each object as a new record in the table Outline.

```
Open Table "world"
Open Table "outline"
```

```
Insert Into outline (Obj)
   Select Obj From World
```

See Also:

Commit Table statement, **Delete statement**, **Rollback statement**

InStr( ) function

Purpose

Returns a character position, indicating where a substring first appears within another string. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

InStr(*position*, *string*, *substring*)

position is a positive integer, indicating the start position of the search.

string is a string expression.

substring is a string expression which we will try to locate in string.

Return Value

Integer

Description

The **InStr( )** function tests whether the string expression *string* contains the string expression *substring*. MapBasic searches the string expression, starting at the position indicated by the *position* parameter; thus, if the *position* parameter has a value of one, MapBasic will search from the very beginning of the *string* parameter.

If *string* does not contain *substring*, the **InStr( )** function returns a value of zero.

If *string* does contain *substring*, the **InStr( )** function returns the character position where the substring appears. For example, if the *substring* appears at the very start of the *string*, **InStr( )** will return a value of one.

If the *substring* parameter is a null string, the **InStr( )** function returns zero.

The **InStr( )** function is case-sensitive. In other words, the **InStr( )** function cannot locate the substring “BC” within the larger string “abcde”, because “BC” is upper-case.

Error Conditions

ERR_FCN_ARG_RANGE (644) error generated if an argument is outside of the valid range

Example

```
Dim fullname As String, pos As Integer
fullname = "New York City"
```

```
pos = InStr(1, fullname, "York")
' pos will now contain a value of 5 (five)

pos = InStr(1, fullname, "YORK")
' pos will now contain a value of 0;
' YORK is uppercase, so InStr will not locate it
' within the string "New York City"
```

See Also:

Mid$( ) function

Int( ) function

Purpose

Returns an integer value obtained by removing the fractional part of a decimal value. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Int(*num_expr*)

num_expr is a numeric expression.

Return Value

Integer

Description

The **Int( )** function returns the nearest integer value that is less than or equal to the specified *num_expr* expression. The **Fix( ) function** is similar to, but not identical to, the **Int( )** function. The two functions differ in the way that they treat negative fractional values. When passed a negative fractional number, **Fix( ) function** will return the nearest integer value greater than or equal to the original value; so, the function call `Fix(-2.3)` will return a value of -2. But when the **Int( )** function is passed a negative fractional number, it returns the nearest integer value that is less than or equal to the original value. So, the function call `Int(-2.3)` returns a value of -3.

Example

```
Dim whole As Integer
whole = Int(5.999)
' whole now has the value 5

whole = Int(-7.2)
' whole now has the value -8
```

See Also:

Fix( ) function, **Round( ) function**

IntersectNodes( ) function

Purpose

Calculates the set of points at which two objects intersect, and returns a polyline object that contains each of the points of intersection. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

IntersectNodes(*object1*, *object2*, *points_to_include*)

object1 and *object2* are object expressions; may not be point or text objects.

points_to_include is one of the following SmallInt values:

- INCL_CROSSINGS returns points where segments cross.
- INCL_COMMON returns end-points of segments that overlap.
- INCL_ALL returns points where segments cross and points where segments overlap.

Return Value

A polyline object that contains the specified points of intersection.

Description

The **IntersectNodes( )** function returns a polyline object that contains all nodes at which two objects intersect.

IsGridCellNull( ) function

Purpose

Returns a Logical. Returns TRUE if the cell value location (x, y) is valid for the table, and is a null cell (a cell that does not have an assigned value). Returns FALSE if the cell contains a value that is non-null. The GetCellValue() function can be used to retrieve the value.

Syntax

IsGridCellNull(*table_id*, *x_pixel*, *y_pixel*)

table_id is a string representing a table name, a positive integer table number, or 0 (zero). The table must be a grid table.

x_pixel is the integer pixel number of the X coordinate of the grid cell. Pixel numbers start at 0. The maximum pixel value is the (pixel_width-1), determined by calling

```
RasterTableInfo(...RASTER_TAB_INFO_WIDTH)
```

y_pixel is the integer pixel number of the Y coordinate of the grid cell. Pixel numbers start at 0. The maximum pixel value is the (pixel_height-1), determined by calling

```
RasterTableInfo(...RASTER_TAB_INFO_HEIGHT).
```

Return Value

A Logical is returned, representing whether the specified cell in the table is null, or non-null. If the grid cell is non-null (IsGridCellNull() returns FALSE), then the GetGridCellValue() function can be called to retrieve the value for that grid pixel.

IsogramInfo( ) function

Purpose

Returns any and all attributes that were set on a connection using the **Set Connection Isogram statement**. Includes attributes to handle the maximum number of records for server, time, and distance values. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

IsogramInfo(*connection_handle, attribute*)

connection_handle is an integer signifying the number of the connection returned from the **Open Connection statement**.

attribute is an Integer code, indicating which type of information should be returned.

Return Value

Float, Logical, or String, depending on the *attribute* parameter.

Description

This function returns the properties defaulted by the connection or the properties that have been changed using the **Set Connection Isogram statement**.

There are several attributes that **IsogramInfo( )** can return. Codes are defined in MAPBASIC.DEF.

attribute setting	ID	IsogramInfo( ) Return Value
ISOGRAM_BANDING	1	Logical representing the Banding option
ISOGRAM_MAJOR_ROADS_ONLY	2	Logical representing the MajorRoadsOnly option
ISOGRAM_RETURN_HOLES	3	Logical representing the choice of returning regions with holes or not.
ISOGRAM_MAJOR_POLYGON_ONLY	4	Logical representing the choice of returning only the main polygon of a region.

attribute setting	ID	IsogramInfo( ) Return Value
ISOGRAM_MAX_OFF_ROAD_DISTANCE	5	Float value representing the Maximum off Road Distance value
ISOGRAM_MAX_OFF_ROAD_DISTANCE_UNITS	6	The unit string associated with the value
ISOGRAM_SIMPLIFICATION_FACTOR	7	Float value representing the Simplification Factor. (a percent value represented as a value between 0 and 1
ISOGRAM_DEFAULT_AMBIENT_SPEED	8	Float value representing the default ambient speed.
ISOGRAM_DEFAULT_AMBIENT_SPEED_DISTANCE_UNIT	9	String value representing the distance unit (“mi”, “km”).
ISOGRAM_DEFAULT_AMBIENT_SPEED_TIME_UNIT	10	String value representing the time unit (“hr”, “min”, “sec”).
ISOGRAM_DEFAULT_PROPAGATION_FACTOR	11	Determines the off-road network percentage of the remaining cost (distance) for which off network travel is allowed when finding the Distance boundary. Roads not identified in the network can be driveways or access roads, among others. The propagation factor is a percentage of the cost used to calculate the distance between the starting point and the Distance. The default value for this property is 0.16.
ISOGRAM_BATCH_SIZE	12	Integer value representing the maximum number of records that are sent to the service at one time.
ISOGRAM_POINTS_ONLY	13	Logical representing the whether or not records that contain non-point objects should be skipped.
ISOGRAM_RECORDS_INSERTED	14	Integer value representing the number of records inserted in the last command.
ISOGRAM_RECORDS_NOTINSERTED	15	Integer value representing the number of records NOT inserted in the last command.
ISOGRAM_MAX_BATCH_SIZE	16	Integer value representing the maximum number of records (for example, points) that the server will permit to be sent to the service at one time.

attribute setting	ID	IsogramInfo( ) Return Value
ISOGRAM_MAX_BANDS	17	Integer value representing the maximum number of Iso bands (for example, distances or times) allowed.
ISOGRAM_MAX_DISTANCE	18	Float value representing the maximum distance permitted for an Isodistance request. The distance units are specified by ISOGRAM_MAX_DISTANCE_UNITS.
ISOGRAM_MAX_DISTANCE_UNITS	19	String value representing the units for ISOGRAM_MAX_DISTANCE.
ISOGRAM_MAX_TIME	20	Float value representing the maximum time permitted for an Isochrone request. The time units are specified by ISOGRAM_MAX_TIME_UNITS.
ISOGRAM_MAX_TIME_UNITS	21	String value representing the units for ISOGRAM_MAX_TIME.

Example

The following MapBasic snippet will print the Envinsa Routing Constraints to the message window in MapInfo Professional:

```
Include "MapBasic.Def"
declare sub main
sub main
dim iConnect as integer
Open Connection Service Isogram
     URL "http://envinsa_server:8062/Route/services/Route"
     User "john"
     Password "green"
     into variable iConnect

Print "Isogram_Max_Batch_Size: " +
IsogramInfo(iConnect,Isogram_Max_Batch_Size)
Print "Isogram_Max_Bands: " + IsogramInfo(iConnect, Isogram_Max_Bands)
Print "Isogram_Max_Distance: " + IsogramInfo(iConnect,
Isogram_Max_Distance)
Print "Isogram_Max_Distance_Units: " + IsogramInfo(iConnect,
Isogram_Max_Distance_Units)
Print "Isogram_Max_Time: " + IsogramInfo(iConnect,Isogram_Max_Time)
Print "Isogram_Max_Time_Units: " +
IsogramInfo(iConnect,Isogram_Max_Time_Units)
Close Connection iConnect
end sub
```

See Also:

Create Object statement, **Open Connection statement**, **Set Connection Isogram statement**

IsPenWidthPixels( ) function

Purpose

The IsPenWidthPixels function determines if a pen width is in pixels or in points. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

IsPenWidthPixels(*penwidth*)

penwidth is a small integer representing the pen width.

Return Value

True if the width value is in pixels. False if the width value is in points.

Description

The **IsPenWidthPixels( )** function will return TRUE if the given pen width is in pixels. The pen width for a line may be determined using the **StyleAttr( ) function**.

Example

```
Include "MAPBASIC.DEF"
Dim CurPen As Pen
Dim Width As Integer
Dim PointSize As Float
CurPen = CurrentPen( )
Width = StyleAttr(CurPen, PEN_WIDTH)
If Not IsPenWidthPixels(Width) Then
   PointSize = PenWidthToPoints(Width)
End If
```

See Also:

CurrentPen( ) function, **MakePen( ) function**, **Pen clause**, **PenWidthToPoints( ) function**, **StyleAttr( ) function**

Kill statement

Purpose

Deletes a file. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Kill *filespec*

filespec is a string which specifies a filename (and, optionally, the file's path).

Return Value

String

Description

The **Kill** statement deletes a file from the disk. There is no “undo” operation for a **Kill** statement. Therefore, the **Kill** statement should be used with caution.

Example

```
Kill "C:\TEMP\JUNK.TXT"
```

See Also:

Open File statement

LabelFindByID( ) function

Purpose

Initializes an internal label pointer, so that you can query the label for a specific row in a map layer. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

LabelFindByID(*map_window_id*, *layer_number*, *row_id*, *table*, *b_mapper*)

map_window_id is an integer window id, identifying a Map window.

layer_number is the number of a layer in the current Map window (for example, 1 for the top layer).

row_id is a positive integer value, indicating the row number of the row whose label you wish to query.

table is a table name or an empty string (""): when you query a table that belongs to a seamless table, specify the name of the member table; otherwise, specify an empty string.

b_mapper is a logical value. Specify TRUE to query the labels that appear when the Map is active; specify FALSE to query the labels that appear when the map is inside a Layout.

Return Value

Logical value: TRUE means that a label exists for the specified row.

Description

Call **LabelFindByID( )** when you want to query the label for a specific row in a map layer. If the return value is TRUE, then a label exists for the row, and you can query the label by calling the **LabelInfo( ) function**.

Example

The following example maps the World table, displays automatic labels, and then determines whether a label was drawn for a specific row in the table.

```
Include "mapbasic.def"
Dim b_morelabels As Logical
Dim i_mapid As Integer
Dim obj_mytext As Object
Open Table "World" Interactive As World
Map From World
i_mapid = FrontWindow( )
Set Map Window i_mapid Layer 1 Label Auto On
' Make sure all labels draw before we continue...
Update Window i_mapid
' Now see if row # 1 was auto-labeled
b_morelabels = LabelFindByID(i_mapid, 1, 1, "", TRUE)
If b_morelabels Then
   ' The object was labeled; now query its label.
   obj_mytext = LabelInfo(i_mapid, 1, LABEL_INFO_OBJECT)
   ' At this point, you could save the obj_mytext object
   ' in a permanent table; or you could query it by
   ' calling ObjectInfo( ) or ObjectGeography( ).
End If
```

See Also:

LabelFindFirst( ) function, **LabelFindNext( ) function**, **LabelInfo( ) function**

LabelFindFirst( ) function

Purpose

Initializes an internal label pointer, so that you can query the first label in a map layer. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

LabelFindFirst(*map_window_id*, *layer_number*, *b_mapper*)

map_window_id is an integer window id, identifying a Map window.

layer_number is the number of a layer in the current Map window (for example, 1 for the top layer).

b_mapper is a logical value. Specify TRUE to query the labels that appear when the Map is active; specify FALSE to query the labels that appear when the map is inside a Layout.

Return Value

Logical value: TRUE means that labels exist for the specified layer (either labels are currently visible, or the user has edited labels, and those edited labels are not currently visible).

Description

Call **LabelFindFirst( )** when you want to loop through a map layer's labels to query the labels. Querying labels is a two-step process:

1. Set MapBasic's internal label pointer by calling the **LabelFindFirst( )** function, the **LabelFindNext( ) function**, or the **LabelFindByID( ) function**.
2. If the function you called in step 1 did not return FALSE, you can query the current label by calling the **LabelInfo( ) function**.

To continue querying additional labels, return to step 1.

Example

For an example, see **LabelInfo( ) function**.

See Also:

LabelFindByID( ) function, **LabelFindNext( ) function**, **LabelInfo( ) function**

LabelFindNext( ) function

Purpose

Advances the internal label pointer, so that you can query the next label in a map layer. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

LabelFindNext(*map_window_id*, *layer_number*)

map_window_id is an integer window id, identifying a Map window.

layer_number is the number of a layer in the current Map window (for example, 1 for the top layer).

Return Value

Logical value: TRUE means the label pointer was advanced to the next label; FALSE means there are no more labels for this layer.

Description

After you call the **LabelFindFirst( ) function** to begin querying labels, you can call **LabelFindNext( )** to advance to the next label in the same layer.

Example

For an example, see **LabelInfo( ) function**.

See Also:

LabelFindByID( ) function, **LabelFindFirst( ) function**, **LabelInfo( ) function**

LabelInfo( ) function

Purpose

Returns information about a label in a map. LabelInfo can return a label as text object and the text object returned can be curved or can be returned as rotated straight text. However, if the label is curved, it will be returned as rotated flat text. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Labelinfo(*map_window_id, layer_number, attribute*)

map_window_id is an integer window id, identifying a Map window.

layer_number is the number of a layer in the current Map window (for example, 1 for the top layer).

attribute is a code indicating the type of information to return; see table below.

Return Value

Return value depends on attribute.

Description

The **Labelinfo( )** function returns information about a label in a Map window.

Labels are different than text objects. To query a text object, call functions such as **ObjectInfo( ) function** or **ObjectGeography( ) function**.

Before calling **Labelinfo( )**, you must initialize MapBasic's internal label pointer by calling the **LabelFindFirst( ) function**, the **LabelFindNext( ) function**, or the **LabelFindByID( ) function**. See the example below.

The attribute parameter must be one of the codes from the following table; codes are defined in MAPBASIC.DEF.

attribute code	ID	Labelinfo( ) Return Value
LABEL_INFO_OBJECT	1	Text object is returned, which is an approximation of the label. This feature allows you to convert a label into a text object, which you can save in a permanent table. ⓘ LABEL_INFO_OBJECT returns a text object, but if the label is curved, it will return a label with a Parallel orientation. MapBasic does not support curved labels as text objects.
LABEL_INFO_POSITION	2	Integer value between 0 and 8, indicating the label's position relative to its anchor location. The return value will match one of these codes: • LAYER_INFO_LBL_POS_CC (0), • LAYER_INFO_LBL_POS_TL (1), • LAYER_INFO_LBL_POS_TC (2), • LAYER_INFO_LBL_POS_TR (3), • LAYER_INFO_LBL_POS_CL (4), • LAYER_INFO_LBL_POS_CR (5), • LAYER_INFO_LBL_POS_BL (6), • LAYER_INFO_LBL_POS_BC (7), • LAYER_INFO_LBL_POS_BR (8). For example, if the label is Below and to the Right of the anchor, its position is 8; if the label is Centered horizontally and vertically over its anchor, its position is zero.
LABEL_INFO_ANCHORX	3	Float value, indicating the x-coordinate of the label's anchor location.
LABEL_INFO_ANCHORY	4	Float value, indicating the y-coordinate of the label's anchor location.
LABEL_INFO_OFFSET	5	Integer value between 0 and 200, indicating the distance (in points) the label is offset from its anchor location.
LABEL_INFO_ROWID	6	Integer value, representing the ID number of the row that owns this label; returns zero if no label exists.

attribute code	ID	Labelinfo( ) Return Value
LABEL_INFO_TABLE	7	String value, representing the name of the table that owns this label. Useful if you are using seamless tables and you need to know which member table owns the label.
LABEL_INFO_EDIT	8	Logical value; TRUE if label has been edited.
LABEL_INFO_EDIT_VISIBILITY	9	Logical value; TRUE if label visibility has been set to OFF.
LABEL_INFO_EDIT_ANCHOR	10	Logical value; TRUE if label has been moved.
LABEL_INFO_EDIT_OFFSET	11	Logical value; TRUE if label's offset has been modified.
LABEL_INFO_EDIT_FONT	12	Logical value; TRUE if label's font has been modified.
LABEL_INFO_EDIT_PEN	13	Logical value; TRUE if callout line's Pen style has been modified.
LABEL_INFO_EDIT_TEXT	14	Logical value; TRUE if label's text has been modified.
LABEL_INFO_EDIT_TEXTARROW	15	Logical value; TRUE if label's text arrow setting has been modified.
LABEL_INFO_EDIT_ANGLE	16	Logical value; TRUE if label's rotation angle has been modified.
LABEL_INFO_EDIT_POSITION	17	Logical value; TRUE if label's position (relative to anchor) has been modified.
LABEL_INFO_EDIT_TEXTLINE	18	Logical value; TRUE if callout line has been moved.
LABEL_INFO_SELECT	19	Logical value; TRUE if label is selected.

attribute code	ID	LabelInfo( ) Return Value
LABEL_INFO_DRAWN	20	Logical value; TRUE if label is currently visible.
LABEL_INFO_ORIENTATION	21	Returns Smallint value indicating the 'current' label's orientation. The current label is initialized by using one of the following Label functions: LabelFindFirst, LabelFindByID, or LabelFindNext. The Return value will be one of these: • LAYER_INFO_LABEL_ORIENT_HORIZONTAL (label has angle equal to 0) • LAYER_INFO_LABEL_ORIENT_PARALLEL (label has non-zero angle) • LAYER_INFO_LABEL_ORIENT_CURVED (label is curved)

Example

The following example shows how to loop through all of the labels for a row, using the **LabelInfo( )** function to query each label.

```
Dim b_morelabels As Logical
Dim i_mapid, i_layernum As Integer
Dim obj_mytext As Object
' Here, you would assign a Map window's ID to i_mapid,
' and assign a layer number to i_layernum.
b_morelabels = LabelFindFirst(i_mapid, i_layernum, TRUE)
Do While b_morelabels
   obj_mytext = LabelInfo(i_mapid, i_layernum, LABEL_INFO_OBJECT)
   ' At this point, you could save the obj_mytext object
   ' in a permanent table; or you could query it by
   ' calling ObjectInfo( ) or ObjectGeography( ).
   b_morelabels = LabelFindNext(i_mapid, i_layernum)
Loop
```

See Also:

LabelFindByID( ) function, **LabelFindFirst( ) function**, **LabelFindNext( ) function**

LabelOverrideInfo( ) function

Returns information about a specific label override.

Syntax

LabelOverrideInfo (
   window_id, *layer_number*, *labeloverride_index*, *attribute*)

window_id is the integer window identifier of a Map window.

layer_number is the number of a layer in the current Map window (for example, 1 for the top layer); to determine the number of layers in a Map window, call the **MapperInfo( ) function**.

labeloverride_index is an integer index (1-based) for the override definition within the layer. Each label override is tied to an zoom range and is ordered so that the smallest zoom range value is on top (index 1).

attribute is a code indicating the type of information to return; see table below.

Return Value

Return value depends on attribute parameter.

Description

The **LabelOverrideInfo( ) function** returns label information for a specific label override for one layer in an existing Map window. The *layer_number* must be a valid layer (1 is the topmost table layer, and so on). The *attribute* parameter must be one of the codes from the following table; codes are defined in MAPBASIC.DEF.

Attribute Code	ID	LabelOverrideInfo( ) Return Value
LBL_OVR_INFO_NAME	1	Label override name.
LBL_OVR_INFO_VISIBILITY	2	Smallint value, indicating whether the override label are visible. The return value will be one of: • LBL_OVR_INFO_VIS_OFF (0) override label is disabled/off; never visible • LBL_OVR_INFO_VIS_ON (1) override label is currently visible in the map • LBL_OVR_INFO_VIS_OFF_ZOOM (2) override label is currently not visible because it's outside the map zoom range
LBL_OVR_INFO_ZOOM_MIN	3	Float value, indicating the minimum zoom value (in MapBasic's current distance units) at which the label override displays.
LBL_OVR_INFO_ZOOM_MAX	4	Float value, indicating the maximum zoom value at which the label override displays.
LBL_OVR_INFO_EXPR	5	String value: the expression used in labels.

Attribute Code	ID	LabelOverrideInfo( ) Return Value
LBL_OVR_INFO_LT	6	SmallInt value indicating what type of line, if any, connects a label to its original location after you move the label. The return value will match one of these values: • LAYER_INFO_LBL_LT_NONE (0) no line • LAYER_INFO_LBL_LT_SIMPLE (1) simple line • LAYER_INFO_LBL_LT_ARROW (2) line with an arrowhead
LBL_OVR_INFO_FONT	7	Font style used in labels.
LBL_OVR_INFO_PARALLEL	8	Logical value: TRUE if layer is set for parallel labels.
LBL_OVR_INFO_POS	9	SmallInt value, indicating label position. Return value will match one of these values (T=Top, B=Bottom, C=Center, R=Right, L=Left): • LAYER_INFO_LBL_POS_CC (0) • LAYER_INFO_LBL_POS_TL (1) • LAYER_INFO_LBL_POS_TC (2) • LAYER_INFO_LBL_POS_TR (3) • LAYER_INFO_LBL_POS_CL (4) • LAYER_INFO_LBL_POS_CR (5) • LAYER_INFO_LBL_POS_BL (6) • LAYER_INFO_LBL_POS_BC (7) • LAYER_INFO_LBL_POS_BR (8)
LBL_OVR_INFO_OVERLAP	10	Logical value; TRUE if overlapping labels are allowed.
LBL_OVR_INFO_DUPLICATES	11	Logical value; TRUE if duplicate labels are allowed.
LBL_OVR_INFO_OFFSET	12	SmallInt value from 0 to 50, indicating how far the labels are offset from object centroids. The offset value represents a distance, in points.
LBL_OVR_INFO_MAX	13	Integer value, indicating the maximum number of labels allowed for this label layer override. If no maximum has been set, return value is 2,147,483,647.

Attribute Code	ID	LabelOverrideInfo( ) Return Value
LBL_OVR_INFO_PARTIALSEGS	14	Logical value; TRUE if the Label Partial Objects check box is checked for this layer.
LBL_OVR_INFO_ORIENTATION	15	Returns Smallint value indicating the setting for the layer's auto label orientation. Return value will be one of these values: • LAYER_INFO_LABEL_ORIENT_HORIZONTAL labels have angle equal to 0 • LAYER_INFO_LABEL_ORIENT_PARALLEL labels have non-zero angle • LAYER_INFO_LABEL_ORIENT_CURVED labels are curved If LAYER_INFO_LABEL_ORIENT_PARALLEL is returned then LBL_OVR_INFO_PARALLEL returns TRUE.
LBL_OVR_INFO_ALPHA	16	SmallInt value, representing the alpha factor for the labels of the specified layer. • 0=fully transparent. • 255=fully opaque. To turn set the translucency or alpha for a layer, use the **Set Map** label clause statement, see **Managing Individual Label Properties**.
LBL_OVR_INFO_AUTODISPLAY	17	Logical value: TRUE if this label override is set to display labels automatically.
LBL_OVR_INFO_POS_RETRY	18	Logical value: TRUE if label overlaps with others, try multiple label positions until a position is found that does not overlap any other labels, or until all position are exhausted.
LBL_OVR_INFO_LINE_PEN	19	Pen style used for displaying the label line.
LBL_OVR_INFO_PERCENT_OVER	20	SmallInt value, max percentage curved label can overhang polyline.

Example

```
LabelOverrideInfo(nMID, nLayer, nOverride, LBL_OVR_INFO_ORIENTATION)
```

See Also:

StyleOverrideInfo( ) function, **LayerStyleInfo( ) function**, **Set Map statement**, **LayerInfo( ) function**

LayerControlInfo( ) function

Purpose

Returns information about the Layer Control window.

Syntax

LayerControlInfo (*attribute*)

attribute is a code indicating the type of information to return; see table below.

Description

The *attribute* parameter is a value from the table below. Codes in the left column are defined in MAPBASIC.DEF.

attribute code	ID	TableInfo( ) returns
LC_INFO_SEL_COUNT	1	Smallint result, indicating the number of selected items.

Example

```
LayerControlInfo(LC_INFO_SEL_COUNT)
```

See Also:

LayerControlSelectionInfo( ) function

LayerControlSelectionInfo( ) function

Purpose

Returns information about a selected item in the Layer Control window.

Syntax

LayerControlSelectionInfo (*selection_index*, *attribute*)

selection_id is the index of a selected item in Layer Control.

attribute is a code indicating the type of information to return; see table below.

Description

The *attribute* parameter can be any value from the table below. Codes in the left column are defined in MAPBASIC.DEF.

attribute code	ID	TableInfo( ) returns
LC_SEL_INFO_NAME	1	String result, representing the name of the selected.
LC_SEL_INFO_TYPE	2	Smallint result, indicating the type of selected item. Return value will be one of the values: • LC_SEL_INFO_TYPE_MAP (0) • LC_SEL_INFO_TYPE_LAYER (1) • LC_SEL_INFO_TYPE_GROUPLAYER (2) • LC_SEL_INFO_TYPE_STYLE_OVR (3) • LC_SEL_INFO_TYPE_LABEL_OVR (4)
LC_SEL_INFO_MAPWIN_ID	3	Integer value, representing the window id of the mapper associated with the selected item.
LC_SEL_INFO_LAYER_ID	4	Smallint value, indicating the ID of the layer associated with the selected item. If you query this value when a map item is selected, the return value is -1.
LC_SEL_INFO_OVR_ID	5	Smallint value, indicating the index of the override associated with the selected item. If you query this value when a map, layer, or grouplayer item is selected, the return value is -1.

Example

```
LayerControlSelectionInfo(layer_number, LC_SEL_INFO_NAME)
```

See Also:

LayerControlInfo( ) function

LayerInfo( ) function

Purpose

Returns information about a layer in a Map window. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

LayerInfo(*window_id*, *layer_number*, *attribute*)

window_id is the integer window identifier of a Map window.

layer_number is the number of a layer in the current Map window (for example, 1 for the top layer); to determine the number of layers in a Map window, call the **MapperInfo( ) function**.

attribute is a code indicating the type of information to return; see table below.

Return Value

Return value depends on attribute parameter.

Restrictions

Many of the settings that you can query using **LayerInfo( ) function** only apply to conventional map layers (as opposed to Cosmetic map layers, thematic map layers, and map layers representing raster image tables). See example below.

Description

The **LayerInfo( ) function** returns information about one layer in an existing Map window. The *layer_number* must be a valid layer (1 is the topmost table layer, and so on). The *attribute* parameter must be one of the codes from the following table; codes are defined in MAPBASIC.DEF. From here you can also query the Hotlink options using the LAYER_HOTLINK_* attributes.

Attribute Code	ID	LayerInfo( ) Return Value
LAYER_INFO_NAME	1	String indicating the name of the table associated with this map layer. If the specified layer is the map's Cosmetic layer, the string will be a table name such as “Cosmetic1”; this table name can be used with other statements (for example, **Select statement**).
LAYER_INFO_EDITABLE	2	Logical value; TRUE if the layer is editable.
LAYER_INFO_SELECTABLE	3	Logical value; TRUE if the layer is selectable.
LAYER_INFO_ZOOM_LAYERED	4	Logical; TRUE if zoom-layering is enabled.
LAYER_INFO_ZOOM_MIN	5	Float value, indicating the minimum zoom value (in MapBasic's current distance units) at which the layer displays. (To set MapBasic's distance units, use **Set Distance Units statement**.)
LAYER_INFO_ZOOM_MAX	6	Float value, indicating the maximum zoom value at which the layer displays.
LAYER_INFO_COSMETIC	7	Logical; TRUE if this is the Cosmetic layer.
LAYER_INFO_PATH	8	String value representing the full directory path of the table associated with the map layer.

Attribute Code	ID	LayerInfo( ) Return Value
LAYER_INFO_DISPLAY	9	SmallInt indicating how and whether this layer is displayed; return value will be one of these values: • LAYER_INFO_DISPLAY_OFF (0) the layer is not displayed • LAYER_INFO_DISPLAY_GRAPHIC (1) objects in this layer appear in their “default” style—the style saved in the table • LAYER_INFO_DISPLAY_GLOBAL (2) objects in this layer are displayed with a “style override” specified in Layer Control • LAYER_INFO_DISPLAY_VALUE (3) objects in this layer appear as thematic shading
LAYER_INFO_OVR_LINE	10	Pen style used for displaying linear objects. If the base set of layer properties includes a stacked style, the pen returned is the first pass of the stacked style.
LAYER_INFO_OVR_PEN	11	Pen style used for displaying the borders of filled objects. If the base set of layer properties includes a stacked style, the pen returned is the first pass of the stacked style.
LAYER_INFO_OVR_BRUSH	12	Brush style used for displaying filled objects. If the base set of layer properties includes a stacked style, the brush returned is the first pass of the stacked style.
LAYER_INFO_OVR_SYMBOL	13	Symbol style used for displaying point objects. If the base set of layer properties includes a stacked style, the symbol returned is the first pass of the stacked style.
LAYER_INFO_OVR_FONT	14	Font style used for displaying text objects. If the base set of layer properties includes a stacked style, the font returned is the first pass of the stacked style.
LAYER_INFO_LBL_EXPR	15	String value: the expression used in labels.

Attribute Code	ID	LayerInfo( ) Return Value
LAYER_INFO_LBL_LT	16	SmallInt value indicating what type of line, if any, connects a label to its original location after you move the label. The return value will match one of these values: • LAYER_INFO_LBL_LT_NONE (0) no line • LAYER_INFO_LBL_LT_SIMPLE (1) simple line • LAYER_INFO_LBL_LT_ARROW (2) line with an arrowhead
LAYER_INFO_LBL_CURFONT	17	For applications compiled with MapBasic 3.x, this query returns the following values: Logical value: TRUE if layer is set to use the current font, or FALSE if layer is set to use the custom font (see LAYER_INFO_LBL_FONT). For applications compiled with MapBasic 4.0 or later, this query always returns FALSE.
LAYER_INFO_LBL_FONT	18	Font style used in labels.
LAYER_INFO_LBL_PARALLEL	19	Logical value: TRUE if layer is set for parallel labels.
LAYER_INFO_LBL_POS	20	SmallInt value, indicating label position. Return value will match one of these values (**T**=Top, **B**=Bottom, **C**=Center, **R**=Right, **L**=Left): • LAYER_INFO_LBL_POS_**TL** (1) • LAYER_INFO_LBL_POS_**TC** (2) • LAYER_INFO_LBL_POS_**TR** (3) • LAYER_INFO_LBL_POS_**CL** (4) • LAYER_INFO_LBL_POS_**CC** (0) • LAYER_INFO_LBL_POS_**CR** (5) • LAYER_INFO_LBL_POS_**BL** (6) • LAYER_INFO_LBL_POS_**BC** (7) • LAYER_INFO_LBL_POS_**BR** (8)
LAYER_INFO_ARROWS	21	Logical value; TRUE if layer displays direction arrows on linear objects.
LAYER_INFO_NODES	22	Logical value; TRUE if layer displays object nodes.

Attribute Code	ID	LayerInfo( ) Return Value
LAYER_INFO_CENTROIDS	23	Logical value; TRUE if layer displays object centroids.
LAYER_INFO_TYPE	24	SmallInt value, indicating this layer's file type: • LAYER_INFO_TYPE_NORMAL (0) for a normal layer • LAYER_INFO_TYPE_COSMETIC (1) for the Cosmetic layer; • LAYER_INFO_TYPE_IMAGE (2) for a raster image layer • LAYER_INFO_TYPE_THEMATIC (3) for a thematic layer • LAYER_INFO_TYPE_GRID (4) for a grid image layer • LAYER_INFO_TYPE_WMS (5) for a layer from a Web Service Map • LAYER_INFO_TYPE_TILESERVER (6) for a layer from a Tile Server
LAYER_INFO_LBL_VISIBILITY	25	SmallInt value, indicating whether labels are visible; see the **Visibility** clause of the **Set Map statement**. Return value will be one of these values: • LAYER_INFO_LBL_VIS_ON (3) labels always visible • LAYER_INFO_LBL_VIS_OFF (1) labels never visible • LAYER_INFO_LBL_VIS_ZOOM (2) labels visible when in zoom range
LAYER_INFO_LBL_ZOOM_MIN	26	Float value, indicating the minimum zoom distance for this layer's labels.
LAYER_INFO_LBL_ZOOM_MAX	27	Float value, indicating the maximum zoom distance for this layer's labels.
LAYER_INFO_LBL_AUTODISPLAY	28	Logical value: TRUE if this layer is set to display labels automatically. See the **Auto** clause of the **Set Map statement**.
LAYER_INFO_LBL_OVERLAP	29	Logical value; TRUE if overlapping labels are allowed.
LAYER_INFO_LBL_DUPLICATES	30	Logical value; TRUE if duplicate labels are allowed.

Attribute Code	ID	LayerInfo( ) Return Value
LAYER_INFO_LBL_OFFSET	31	SmallInt value from 0 to 50, indicating how far the labels are offset from object centroids. The offset value represents a distance, in points.
LAYER_INFO_LBL_MAX	32	Integer value, indicating the maximum number of labels allowed for this layer. If no maximum has been set, return value is 2,147,483,647.
LAYER_INFO_LBL_PARTIALSEGS	33	Logical value; TRUE if the **Label Partial Objects** check box is checked for this layer.
LAYER_INFO_HOTLINK_EXPR	34	Returns the layer's Hotlink filename expression. Can return empty string (““)
LAYER_INFO_HOTLINK_MODE	35	Returns the layer's Hotlink mode, one of the following predefined values: • HOTLINK_MODE_LABEL (0) default • HOTLINK_MODE_OBJ (1) • HOTLINK_MODE_BOTH (2)
LAYER_INFO_HOTLINK_RELATIVE	36	Returns TRUE if the relative path option is on, FALSE otherwise. FALSE is default.
LAYER_INFO_HOTLINK_COUNT	37	Allows you to query the number of hotlink definitions in a layer.
LAYER_INFO_LBL_ORIENTATION*	38	Returns Smallint value indicating the setting for the layer's auto label orientation. Return value will be one of these values: • LAYER_INFO_LABEL_ORIENT_HORIZONTAL labels have angle equal to 0 • LAYER_INFO_LABEL_ORIENT_PARALLEL labels have non-zero angle • LAYER_INFO_LABEL_ORIENT_CURVED labels are curved If LAYER_INFO_LABEL_ORIENT_PARALLEL is returned then LBL_OVR_INFO_PARALLEL returns TRUE.
LAYER_INFO_LAYER_ALPHA	39	SmallInt value, representing the alpha factor for the specified layer. • 0=fully transparent. • 255=fully opaque. To set the translucency or alpha for a layer, use the Set Map statement.

Attribute Code	ID	LayerInfo( ) Return Value
LAYER_INFO_LAYER_TRANSLUCENCY	40	SmallInt value, representing the translucency percentage for the specified layer. • 100=fully transparent. • 0=fully opaque. To set the translucency or alpha for a layer, use the Set Map statement.
LAYER_INFO_LABEL_ALPHA	41	SmallInt value, representing the alpha factor for the labels of the specified layer. • 0=fully transparent. • 255=fully opaque. To set the translucency or alpha for a layer, use the Set Map LABELCLAUSE statement.
LAYER_INFO_LAYERLIST_ID	42	Returns the overall numeric ID of the layer in the current layer list. For example, a layer may be the first group layer from the top down in the map layer list (its group layer ID would be 1), but it may be the 4th layer from the top. Thus its layer list ID would be 4. This ID can be used with the **LayerListInfo function**.
LAYER_INFO_PARENT_GROUP_ID	43	Returns the group layer ID of the immediate group containing this layer, returns 0 if layer is in the top level list.
LAYER_INFO_OVR_STYLE_COUNT	44	Returns a smallint; indicates the number of display style overrides.
LAYER_INFO_OVR_LBL_COUNT	45	Returns a smallint; indicates the number of label overrides.
LAYER_INFO_OVR_STYLE_CURRENT	46	Returns a smallint; indicates display style override index in current zoom range, 0 means no override.
LAYER_INFO_OVR_LBL_CURRENT	47	Returns a smallint; indicates label override index in current zoom range, 0 means no override.
LAYER_INFO_OVR_LINE_COUNT	48	Returns a smallint; indicates the number of Pen styles defined for displaying linear objects for the layer's base set of properties.

Attribute Code	ID	LayerInfo( ) Return Value
LAYER_INFO_OVR_PEN_COUNT	49	Returns a smallint; indicates the number of Pen styles defined for displaying borders of filled objects for the layer's base set of properties.
LAYER_INFO_OVR_BRUSH_COUNT	50	Returns a smallint; indicates the number of brush styles defined for displaying filled objects for the layer's base set of properties.
LAYER_INFO_OVR_SYMBOL_COUNT	51	Returns a smallint; indicates the number of symbol styles defined for displaying point objects for the layer's base set of properties.
LAYER_INFO_OVR_FONT_COUNT	52	Returns a smallint; indicates the number of font styles defined for displaying text objects for the layer's base set of properties. This always returns 1, because font style is not supported by stacked styles.

Hotlinks

For backwards compatibility, the original set of attributes before version 10.0 still work, and will return the values for the layer's first hotlink definition. If no hotlinks are defined when the function is called, then the following values are returned:

LAYER_INFO_HOTLINK_EXPR – empty string ("")

LAYER_INFO_HOTLINK_MODE – returns default value HOTLINK_MODE_LABEL

LAYER_INFO_HOTLINK_RELATIVE – returns default value FALSE

Example

Many of the settings that you can query using **LayerInfo( )** only apply to conventional map layers (as opposed to cosmetic map layers, thematic map layers, and map layers representing raster image tables).

To determine whether a map layer is a conventional layer, use the LAYER_INFO_TYPE setting, as shown below:

```
i_lay_type = LayerInfo( map_id, layer_number, LAYER_INFO_TYPE)

If i_lay_type = LAYER_INFO_TYPE_NORMAL Then
   '
   ' ... then this is a "normal" layer
   '
End If
```

See Also:

GroupLayerInfo function, **LayerListInfo function**, **MapperInfo( ) function**, **Set Map statement**

LayerListInfo function

Purpose

This function helps to enumerate a map's list of layers and can refer to both group and graphical layers.

Syntax

LayerListInfo(*map_window_id*, *numeric_counter*, *attribute*)

map_window_id is a Map window identifier.

numeric_counter is value from zero (0) to MAPPER_INFO_ALL_LAYERS, which is the number of layers in the Map window excluding the cosmetic layer. For details about MAPPER_INFO_ALL_LAYERS, see **MapperInfo( ) function**.

attribute is a code indicating the type of information to return; see table below.

Return Value

Depends on the *attribute* parameter.

Description

This function can be used to iterate over all the components of the map's layer list where *numeric_counter* goes from zero (0) to MAPPER_INFO_ALL_LAYERS.

The attributes are:

Value of window_id, attribute	ID	Description
LAYERLIST_INFO_TYPE	1	The type of layer in the list: • LAYERLIST_INFO_TYPE_LAYER (0) • LAYERLIST_INFO_TYPE_GROUP (1)
LAYERLIST_INFO_NAME	2	Returns a string value, which is the name of the layer or group layer.
LAYERLIST_INFO_LAYER_ID	3	Returns a numeric value, Layer-ID of the layer. Use this value to query the layer further using the **LayerInfo( ) function**.
LAYERLIST_INFO_GROUPLAYER_ID	4	Returns a numeric value, GroupLayer-ID of GroupLayer. Use this value to query the group layer further using the **GroupLayerInfo function**.

If the type returns a graphical layer, then use LayerInfo to get attributes. If it is a group layer then use GroupLayerInfo to get attributes. To loop through this flattened view of the layer list, use MAPPER_INFO_ALL_LAYERS as the looping limit.

Specifying a map window ID of zero (0) returns information about the Cosmetic Layer.

See Also:

GroupLayerInfo function, **LayerInfo( ) function**, **MapperInfo( ) function**

LayerStyleInfo( ) function

Returns style information for a stacked style (a style composed of one or more style definitions).

Syntax

```
LayerStyleInfo (
   window_id, layer_number, override_index, pass_index, attribute )
```

window_id is the integer window identifier of a Map window.

layer_number is the number of a layer in the current Map window (for example, 1 for the top layer); to determine the number of layers in a Map window, call the **MapperInfo( ) function**.

override_index is an integer index (0-based, where 0 for the layer's base set of properties) and 1 or higher is for a style override.

pass_index is an integer index (1-based) where the index corresponds to a pass within the stacked style. The first pass is the part of the style drawn first, the second pass is the part of the style drawn next, and so on.

attribute is a code indicating the type of information to return; see table below.

Return Value

Return value depends on attribute parameter.

Description

The **LayerStyleInfo( ) function** returns style information for a stacked style. A stacked style is made up of one or more style definitions. For example, a line style drawn with two separate styles; a thin light red line drawn on top of a thicker dark red line would be descried as follows using MapBasic syntax:

```
Line (7,2,12582912), Line (3,2,16736352)
```

The thicker dark red line in this example is drawn first.

The *layer_number* must be a valid layer (1 is the topmost table layer, and so on). The *attribute* parameter must be one of the codes from the following table; codes are defined in MAPBASIC.DEF.

Attribute Code	ID	LayerStyleInfo( ) Return Value
STYLE_OVR _INFO_LINE	10	Pen style used for displaying the specified pass for linear objects.
STYLE_OVR _INFO_PEN	11	Pen style used for displaying the specified pass for the borders of filled objects.
STYLE_OVR_INFO_BRUSH	12	Brush style used for displaying the specified pass for filled objects.
STYLE_OVR_INFO_SYMBOL	13	Symbol style used for displaying the specified pass for point objects.
STYLE_OVR_INFO_FONT	14	Font style used for displaying the specified pass for text objects.

Example

```
LayerStyleInfo(nMID, nLayer, nOverride, nPass, STYLE_OVR_INFO_PEN)
```

See Also:

StyleOverrideInfo( ) function, **LabelOverrideInfo( ) function**, **Set Map statement**, **LayerInfo( ) function**

Layout statement

Purpose

Opens a new layout window. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Layout
   [ Position ( x, y ) [ Units paperunits ] ]
   [ Width window_width [ Units paperunits ] ]
   [ Height window_height [ Units paperunits ] ]
   [ { Min | Max } ]
```

paperunits is a string representing the name of a paper unit (for example, “in” or “mm”).

x, *y* specifies the position of the upper left corner of the Layout, in paper units, where 0,0 represents the upper-left corner of the MapInfo Professional window.

window_width and *window_height* dictate the size of the window, in Paper units.

Description

The **Layout** statement opens a new Layout window. If the statement includes the optional **Min** keyword, the window is minimized before it is displayed. If the statement includes the optional **Max** keyword, the window appears maximized, filling all of MapInfo Professional's screen space.

The **Width** and **Height** clauses control the size of the Layout window, not the size of the page layout itself. The page layout size is controlled by the paper size currently in use and the number of pages included in the Layout.

See **Set Layout statement** for more information on setting the number of pages in a Layout.

MapInfo Professional assigns a special hidden table name to each Layout window. The first Layout window opened has the table name Layout1, the next Layout window that is opened has the table name Layout2, etc.

A MapBasic program can create, select, or modify objects on a Layout window by issuing statements which refer to these table names. For example, the following statement selects all objects from a Layout window:

```
Select * From Layout1
```

Example

The following example creates a Layout window two inches wide by four inches high, located at the upper-left corner of the MapInfo Professional workspace.

```
Layout Position (0, 0) Width 2 Height 4
```

See Also:

Open Window statement, **Set Layout statement**

LCase$( ) function

Purpose

Returns a lower-case equivalent of a string. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

LCase$(*string_expr*)

string_expr is a string expression.

Return Value

String

Description

The **LCase$( )** function returns the string which is the lower-case equivalent of the string expression string_expr.

Conversion from upper- to lower-case only affects alphabetic characters (A through Z); numeric digits, and punctuation marks are not affected. Thus, the function call:

```
LCase$( "A#12a" )
```

returns the string value “a#12a”.

Example

```
Dim regular, lower_case As String
regular = "Los Angeles"
lower_case = LCase$(regular)
'
' Now, lower_case contains the value "los angeles"
'
```

See Also:

Proper$( ) function, **UCase$( ) function**

Left$( ) function

Purpose

Returns part or all of a string, beginning at the left end of the string. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Left$(*string_expr*, *num_expr*)

string_expr is a string expression.

num_expr is a numeric expression, zero or larger.

Return Value

String

Description

The **Left$( )** function returns a string which consists of the leftmost *num_expr* characters of the string expression *string_expr*.

The *num_expr* parameter should be an integer value, zero or larger. If *num_expr* has a fractional value, MapBasic rounds to the nearest integer. If *num_expr* is zero, **Left$( )** returns a null string. If the *num_expr* parameter is larger than the number of characters in the *string_expr* string, **Left$( )** returns a copy of the entire *string_expr* string.

Example

```
Dim whole, partial As String
whole = "Afghanistan"
partial = Left$(whole, 6)

' at this point, partial contains the string: "Afghan"
```

See Also:

Mid$( ) function, **Right$( ) function**

LegendFrameInfo( ) function

Purpose

Returns information about a frame within a legend. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

LegendFrameInfo(*window_id*, *frame_id*, *attribute*)

window_id is a number that specifies which legend window you want to query.

frame_id is a number that specifies which frame within the legend window you want to query. Frames are numbered 1 to *n* where *n* is the number of frames in the legend.

attribute is an integer code indicating which type of information to return.

Return Value

Depends on the *attribute* parameter.

Attribute codes	ID	LegendFrameInfo( ) Return Value
FRAME_INFO_TYPE	1	Returns one of the following predefined constant indicating frame type: • FRAME_TYPE_STYLE (1) • FRAME_TYPE_THEME (2)
FRAME_INFO_MAP_LAYER_ID	2	Returns the ID of the layer to which the frame corresponds.
FRAME_INFO_REFRESHABLE	3	Returns TRUE if the frame was created without the **Norefresh** keyword. Always returns TRUE for theme frames.

Attribute codes	ID	LegendFrameInfo( ) Return Value
FRAME_INFO_POS_X	4	Returns the distance of the frame's upper left corner from the left edge of the legend canvas (in paper units).
FRAME_INFO_POS_Y	5	Returns the distance of the frame's upper left corner from the top edge of the legend canvas (in paper units).
FRAME_INFO_WIDTH	6	Returns the width of the frame (in paper units).
FRAME_INFO_HEIGHT	7	Returns the height of the frame (in paper units).
FRAME_INFO_TITLE	8	Returns the title of a style frame or theme frame.
FRAME_INFO_TITLE_FONT	9	Returns the font of a style frame title. Returns the default title font if the frame has no title or if it is a theme frame.
FRAME_INFO_SUBTITLE	10	Returns the subtitle of a style frame or theme frame.
FRAME_INFO_SUBTITLE_FONT	11	Same as FRAME_INFO_TITLE_FONT (9)
FRAME_INFO_BORDER_PEN	12	Returns the pen used to draw the border.
FRAME_INFO_NUM_STYLES	13	Returns the number of styles in a frame. Zero if theme frame.
FRAME_INFO_VISIBLE	14	Returns TRUE if the frame is visible (theme frames can be invisible).
FRAME_INFO_COLUMN	15	Returns the legend attribute column name as a string if there is one. Returns an empty string for a theme frame.
FRAME_INFO_LABEL	16	Returns the label expression as a string if there is one. Returns an empty string for a theme frame.

LegendInfo( ) function

Purpose

Returns information about a legend. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

LegendInfo(*window_id*, *attribute*)

window_id is a number that specifies which legend window you want to query.

attribute is an integer code indicating which type of information to return.

Return Value

Depends on the attribute parameter.

Attribute Code	ID	LegendInfo( ) Return Value
LEGEND_INFO_MAP_ID	1	Returns the ID of the parent map window (can also get this value by calling the **WindowInfo( ) function** with the WIN_INFO_TABLE code).
LEGEND_INFO_ORIENTATION	2	Returns predefined value to indicate the layout of the legend: • ORIENTATION_PORTRAIT (1) • ORIENTATION_LANDSCAPE (2) • ORIENTATION_CUSTOM (3)
LEGEND_INFO_NUM_FRAMES	3	Returns the number of frames in the legend.
LEGEND_INFO_STYLE_SAMPLE_SIZE	4	Returns 0 for small legend sample size style or 1 for large legend sample size style.

Example

```
LegendInfo(FrontWindow( ) LEGEND_INFO_STYLE_SAMPLE_SIZE)
```

See Also:

LegendStyleInfo( ) function

LegendStyleInfo( ) function

Purpose

Returns information about a style item within a legend frame. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

LegendStyleInfo(*window_id*, *frame_id*, *style_id*, *attribute*)

window_id is a number that specifies which legend window you want to query.

frame_id is a number that specifies which frame within the legend window you want to query. Frames are numbered 1 to *n* where *n* is the number of frames in the legend.

style_id is a number that specifies which style within a frame you want to query. Styles are numbered 1 to *n* where *n* is the number of styles in the frame.

attribute is an integer code indicating which type of information to return.

Return Value

Attribute Code	ID	LegendStyleInfo( ) Return Values
LEGEND_STYLE_INFO_TEXT	1	Returns the text of the style.
LEGEND_STYLE_INFO_FONT	2	Returns the font of the style.
LEGEND_STYLE_INFO_OBJ	3	Returns the object of the style.

Error Conditions

Generates an error when issued on a frame that has no styles (theme frame).

See Also:

LegendInfo( ) function

Len( ) function

Purpose

Returns the number of characters in a string or the number of bytes in a variable. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Len(*expr*)

expr is a variable expression. expr cannot be a Pen, Brush, Symbol, Font, or Alias.

Return Value

SmallInt

Description

The behavior of the **Len( )** function depends on the data type of the *expr* parameter.

If the *expr* expression represents a character string, the **Len( )** function returns the number of characters in the string.

Otherwise, if *expr* is a MapBasic variable, **Len( )** returns the size of the variable, in bytes. Thus, if you pass an integer variable, **Len( )** will return the value 4 (because each integer variable occupies 4 bytes), while if you pass a SmallInt variable, **Len( )** will return the value 2 (because each SmallInt variable occupies 2 bytes).

Example

```
Dim name_length As SmallInt
name_length = Len("Boswell")

' name_length now has the value: 7
```

See Also:

ObjectLen( ) function

LibraryServiceInfo( ) function

Purpose

Returns information about the Library Services, such as the current mode of operation, version, or default URL for the Library Service. It also gives the list of CSW URL's exposed by the MapInfo Manager sever.

Syntax

LibraryServiceInfo(*attribute*)

attribute is a code indicating the type of information to return; see table below.

Description

The **LibraryServiceInfo( )** function returns one piece of information about the Library Services.

The attribute parameter is a value from the table below. Codes in the left column are defined in MAPBASIC.DEF.

attribute code	ID	LibraryServiceInfo( ) returns
LIBSRVC_INFO_LIBSRVCMODE	1	Integer result, indicating the current mode of operation of the Library Service.
LIBSRVC_INFO_LIBVERSION	2	String result, indicating the version of the Library Service. The default Library Service URL should be set before calling this function.
LIBSRVC_INFO_DEFURLPATH	3	String result, indicating the default URL for the Library Service. The default value for the Library URL is an empty string.
LIBSRVC_INFO_LISTCSWURL	4	String result, gives the list of CSW URL's exposed by the MapInfo Manager sever as a single string delimited by a semi-colon (;).

Example

The following example shows how to use this function:

```
include "mapbasic.def"
declare sub main
sub main
dim liburlpath as string
dim libversion as string
liburlpath = LibraryServiceInfo(LIBSRVC_INFO_DEFURLPATH)if
StringCompare(liburlpath, "") == 0 then
Set LibraryServiceInfo URL
"http://localhost:8080/LibraryService/LibraryService"
endif
libversion = LibraryServiceInfo(LIBSRVC_INFO_LIBVERSION)
end sub
```

See Also:

Set LibraryServiceInfo statement

Like( ) function

Purpose

Returns TRUE or FALSE to indicate whether a string satisfies pattern-matching criteria. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Like(*string*, *pattern_string*, *escape_char*)

string is a string expression to test.

pattern_string is a string that contains regular characters or special wild-card characters.

escape_char is a string expression defining an escape character. Use an escape character (for example, "\") if you need to test for the presence of one of the wild-card characters ("%" and "_") in the string expression. If no escape character is desired, use an empty string ("").

Return Value

Logical value (TRUE if string matches *pattern_string*).

Description

The **Like( )** function performs string pattern-matching. This string comparison is case-sensitive; to perform a comparison that is case-insensitive, use the Like operator.

The pattern_string parameter can contain the following wildcard characters:

_ (underscore)	matches a single character.
% (percent)	matches zero or more characters.

To search for instances of the underscore or percent characters, specify an *escape_char* parameter, as shown in the table below.

To determine if a string…	Specify these parameters:
starts with “South”	`Like( string_var, "South%", "")`
ends with “America”	`Like( string_var, "%America", "")`
contains “ing” at any point	`Like( string_var, "%ing%", "" )`
starts with an underscore	`Like( string_var, "\_%", "\")`

See Also:

Len( ) function, **StringCompare( ) function**

Line Input statement

Purpose

Reads a line from a sequential text file into a variable.

Syntax

`Line Input` [**#**] *filenum*, *var_name*

filenum is an integer value, indicating the number of an open file.

var_name is the name of a string variable.

Description

The **Line Input** statement reads an entire line from a text file, and stores the results in a string variable. The text file must already be open, in Input mode.

The **Line Input** statement treats each line of the file as one long string. If each line of a file contains a comma-separated list of expressions, and you want to read each expression into a separate variable, use the **Input # statement** instead of **Line Input**.

Example

The following program opens an existing text file, reads the contents of the text file one line at a time, and copies the contents of the file to a separate text file.

```
Dim str As String
Open File "original.txt" For Input As #1
Open File "copy.txt" For Output As #2
   Do While Not EOF(1)
      Line Input #1, str
      If Not EOF(1) Then
         Print #2, str
      End If
   Loop
Close File #1
Close File #2
```

See Also:

Input # statement, **Open File statement**, **Print # statement**

LocateFile$( ) function

Purpose

Return the path to one of the MapInfo application data files. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

LocateFile$(*file_id*)

file_id is one of the following values

Value	ID	Description
LOCATE_PREF_FILE	0	Preference file (MAPINFOW.PRF).
LOCATE_DEF_WOR	1	Default workspace file (MAPINFOW.WOR).
LOCATE_CLR_FILE	2	Color file (MAPINFOW.CLR).
LOCATE_PEN_FILE	3	Pen file (MAPINFOW.PEN).
LOCATE_FNT_FILE	4	Symbol file (MAPINFOW.FNT).
LOCATE_ABB_FILE	5	Abbreviation file (MAPINFOW.ABB).
LOCATE_PRJ_FILE	6	Projection file (MAPINFOW.PRJ).
LOCATE_MNU_FILE	7	Menu file (MAPINFOW.MNU).

Value	ID	Description
LOCATE_CUSTSYMB_DIR	8	Custom symbol directory (CUSTSYMB).
LOCATE_THMTMPLT_DIR	9	Theme template directory (THMTMPL).
LOCATE_GRAPH_DIR	10	Graph support directory (GRAPHSUPPORT).
LOCATE_WMS_SERVERLIST	11	XML list of WMS servers (MIWMSSERVERS.XML).
LOCATE_WFS_SERVERLIST	12	XML list of WFS servers (MIWFSSERVERS.XML).
LOCATE_GEOCODE_SERVERLIST	13	XML list of geocode servers (MIGEOCODESERVERS.XML).
LOCATE_ROUTING_SERVERLIST	14	XML list of routing servers (MIROUTINGSERVERS.XML).
LOCATE_LAYOUT_TEMPLATE_DIR	15	Layout template directory (LAYOUTTEMPLATE)

Return Value

String

Description

Given the ID of a MapInfo Professional application data file, this function returns the location where MapInfo Professional found that file. MapInfo Professional installs these files under the user's Application Data directory, but there are several valid locations for these files, including the program directory. MapBasic applications should not assume the location of these files, instead **LocateFile$( )** should be used to determine the actual location.

Example

```
include "mapbasic.def"
declare sub main
sub main
dim sGraphLocations as string
sGraphLocations = LocateFile$(LOCATE_GRAPH_DIR)
Print sGraphLocations
end sub
```

See Also:

GetFolderPath$( ) function

LOF( ) function

Purpose

Returns the length of an open file.

Syntax

LOF(*filenum*)

filenum is the number of an open file.

Return Value

Integer

Description

The **LOF( )** function returns the length of an open file, in bytes.

The file parameter represents the number of an open file; this is the same number specified in the **As** clause of the **Open File statement**.

Error Conditions

ERR_FILEMGR_NOTOPEN (366) error generated if the specified file is not open.

Example

```
Dim size As Integer
Open File "import.txt" For Binary As #1
size = LOF(1)
' size now contains the # of bytes in the file
```

See Also:

Open File statement

Log( ) function

Purpose

Returns the natural logarithm of a number. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Log(*num_expr*)

num_expr is a numeric expression.

Return Value

Float

Description

The **Log( )** function returns the natural logarithm of the numeric expression specified by the *num_expr* parameter.

The natural logarithm represents the number to which the mathematical value e must be raised in order to obtain *num_expr*. e has a value of approximately 2.7182818.

The logarithm is only defined for positive numbers; accordingly, the **Log( )** function will generate an error if *num_expr* has a negative value.

You can calculate logarithmic values in other bases (for example, base 10) using the natural logarithm. To obtain the base-10 logarithm of the number *n*, divide the natural log of *n* (**Log(** *n* **)**) by the natural logarithm of 10 (*Log(10)*).

Example

```
Dim original_val, log_val As Float
original_val = 2.7182818
log_val = Log(original_val)

' log_val will now have a value of 1 (approximately),
' since E raised to the power of 1 equals
' 2.7182818 (approximately)
```

See Also:

Exp( ) function

LTrim$( ) function

Purpose

Trims space characters from the beginning of a string and returns the results. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

LTrim$(*string_expr*)

string_expr is a string expression.

Return Value

String

Description

The **LTrim$( )** function removes any spaces from the beginning of the *string_expr* string, and returns the resultant string.

Example

```
Dim name As String
name = " Mary Smith"
name = LTrim$(name)

' name now contains the string "Mary Smith"
```

See Also:

RTrim$( ) function

Main procedure

Purpose

The first procedure called when an application is run.

Syntax

```
Declare Sub Main
Sub Main
    statement_list
End Sub
```

statement_list is a list of statements to execute when an application is run.

Description

Main is a special-purpose MapBasic procedure name. If an application contains a sub procedure called **Main**, MapInfo Professional runs that procedure automatically when the application is first run. The **Main** procedure can then take actions (for example, issuing **Call statement**s) to cause other sub procedures to be executed.

However, you are not required to explicitly declare the **Main** procedure. Instead of declaring a procedure named **Main**, you can simply place one or more statements at or near the top of your program file, outside of any procedure declaration. MapBasic will then treat that group of statements as if they were in a **Main** procedure. This is known as an “implicit” **Main** procedure (as opposed to an “explicit” **Main** procedure).

Example

A MapBasic program can be as short as a single line. For example, you could create a MapBasic program consisting only of the following statement:

```
Note "Testing, one two three."
```

If the statement above comprises your entire program, MapBasic considers that program to be in an implicit Main procedure. When you run that application, MapBasic will execute the **Note statement**.

Alternately, the following example explicitly declares the **Main** procedure, producing the same results (for example, a **Note statement**).

```
Declare Sub Main
Sub Main
   Note "Testing, one two three."
End Sub
```

The next example contains an implicit **Main** procedure, and a separate sub procedure called Talk. The implicit **Main** procedure calls the Talk procedure through the **Call statement**.

```
Declare Sub Talk(ByVal msg As String)
Call Talk("Hello")
Call Talk("Goodbye")
Sub Talk(ByVal msg As String)
   Note msg
End Sub
```

The next example contains an explicit **Main** procedure, and a separate sub procedure called Talk. The **Main** procedure calls the Talk procedure through the Call statement.

```
Declare Sub Main
Declare Sub Talk(ByVal msg As String)

Sub Main
   Call Talk("Hello")
   Call Talk("Goodbye")
End Sub

Sub Talk(ByVal msg As String)
   Note msg
End Sub
```

See Also:

EndHandler procedure, **RemoteMsgHandler procedure**, **SelChangedHandler procedure**, **Sub…End Sub statement**, **ToolHandler procedure**, **WinClosedHandler procedure**

MakeBrush( ) function

Purpose

Returns a Brush value. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

MakeBrush(*pattern*, *forecolor*, *backcolor*)

pattern is an integer value from 1 to 8 or from 12 to 71, dictating a fill pattern. See **Brush clause** for a listing of the patterns.

forecolor is the integer RGB color value of the foreground of the pattern. See **RGB( ) function** for details.

backcolor is the integer RGB color value of the background of the pattern. To make the background transparent, specify -1 as the background color, and specify a pattern of 3 or greater.

Return Value

Brush

Description

The **MakeBrush( )** function returns a Brush value. The return value can be assigned to a Brush variable, or may be used as a parameter within a statement that takes a Brush setting as a parameter (such as Create Ellipse, Set Map, Set Style, or Shade).

See **Brush clause** for more information about Brush settings.

Example

```
Include "mapbasic.def"
Dim b_water As Brush
b_water = MakeBrush(64, CYAN, BLUE)
```

See Also:

Brush clause, **CurrentBrush( ) function**, **RGB( ) function**, **StyleAttr( ) function**

MakeCustomSymbol( ) function

Purpose

Returns a Symbol value based on a bitmap file. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

MakeCustomSymbol(*filename*, *color*, *size*, *customstyle*)

filename is a string up to 31 characters long, representing the name of a bitmap file. The file must be in the CustSymb directory inside the user's MapInfo directory.

color is an integer RGB color value; see **RGB( ) function** for details.

size is an integer point size, from 1 to 48.

customstyle is an integer code controlling color and background attributes. See table below.

Return Value

Symbol

Description

The **MakeCustomSymbol( )** function returns a Symbol value based on a bitmap file. See **Symbol clause** for information about other symbol types.

The following table describes how the customstyle argument controls the symbol's style:

customstyle value	Symbol Style
0	The Show Background, the Apply Color, and the Display at Actual Size settings are off; the symbol appears in its default state at the point size specified by the size parameter. White pixels in the bitmap are displayed as transparent, allowing whatever is behind the symbol to show through.
1	The Show Background setting is on; white pixels in the bitmap are opaque.
2	The Apply Color setting is on; non-white pixels in the bitmap are replaced with the symbol's color setting.
3	Both Show Background and Apply Color are on.
4	The Display at Actual Size setting is on; the bitmap image is rendered at its native width and height in pixels.
5	The Show Background and Display at Actual Size settings are on.
7	The Show Background, the Apply Color, and the Display at Actual Size settings are on.

Example

```
Include "mapbasic.def"
Dim sym_marker As Symbol
sym_marker = MakeCustomSymbol("CAR1-64MP", BLUE, 18, 0)
```

See Also:

CurrentSymbol( ) function, **MakeFontSymbol( ) function**, **MakeSymbol( ) function**, **StyleAttr( ) function**, **Symbol clause**

MakeDateTime function

Purpose

Returns a DateTime made from the specified Date and Time. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

```
MakeDateTime (Date, Time)
```

Return Value

DateTime

Example

Copy this example into the MapBasic window for a demonstration of this function.

```
dim tX as time
dim dX as date
dim dtX as datetime
tX = 105604123
dX = 20070908
dtX = MakeDateTime(dX,tX)
Print FormatDate$(GetDate(dtX))
Print FormatTime$(GetTime(dtX), "hh:mm:ss.fff tt")
```

MakeFont( ) function

Purpose

Returns a Font value. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

MakeFont(*fontname*, *style*, *size*, *forecolor*, *backcolor*)

fontname is a text string specifying a font (for example, “Arial”). This argument is case sensitive.

style is a positive integer expression; 0 = plain text, 1 = bold text, etc. See Font clause for details.

size is an integer point size, one or greater.

forecolor is the RGB color value for the text. See **RGB( ) function**.

backcolor is the RGB color value for the background (or the halo color, if the style setting specifies a halo). To make the background transparent, specify -1 as the background color.

Return Value

Font

Description

The **MakeFont( )** function returns a Font value. The return value can be assigned to a Font variable, or may be used as a parameter within a statement that takes a Font setting as a parameter (such as **Create Text statement** or **Set Style statement**).

See **Font clause** for more information about Font settings.

Example

```
Include "mapbasic.def"
Dim big_title As Font
big_title = MakeFont("Arial", 1, 20,BLACK,WHITE)
```

See Also:

CurrentFont( ) function, **Font clause**, **StyleAttr( ) function**

MakeFontSymbol( ) function

Purpose

Returns a Symbol value, using a character from a TrueType font as the symbol. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

MakeFontSymbol(*shape*, *color*, *size*, *fontname*, *fontstyle*, *rotation*)

shape is a SmallInt value, 31 or larger (31 is invisible), specifying a character code from a TrueType font.

color is an integer RGB color value; see **RGB( ) function** for details.

size is a SmallInt value from 1 to 48, dictating the point size of the symbol.

fontname is a string representing the name of a TrueType font (for example, “WingDings”). This argument is case sensitive.

fontstyle is a numeric code controlling bold, outline, and other attributes; see below.

rotation is a floating-point number indicating the symbol's rotation angle, in degrees.

Return Value

Symbol

Description

The **MakeFontSymbol( )** function returns a Symbol value based on a character in a TrueType font. See **Symbol clause** for information about other symbol types.

The following table describes how the *fontstyle* parameter controls the symbol's style:

fontstyle value	Symbol Style
0	Plain
1	Bold
16	Border (black outline)
32	Drop Shadow
256	Halo (white outline)

To specify two or more style attributes, add the values from the left column. For example, to specify both the Bold and the Drop Shadow attributes, use a fontstyle value of 33. Border and Halo are mutually exclusive.

Example

```
Include "mapbasic.def"
Dim sym_marker As Symbol
sym_marker = MakeFontSymbol(65,RED,24,"WingDings",32,0)
```

See Also:

CurrentSymbol( ) function, **MakeCustomSymbol( ) function**, **MakeSymbol( ) function**, **StyleAttr( ) function**, **Symbol clause**

MakePen( ) function

Purpose

Returns a Pen value. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

MakePen(*width*, *pattern*, *color*)

width specifies a pen width.

pattern specifies a line pattern; see Pen clause for a listing.

color is the RGB color value; see **RGB( ) function** for details.

Return Value

Pen

Description

The **MakePen( )** function returns a Pen value, which defines a line style. The return value can be assigned to a Pen variable, or may be used as a parameter within a statement that takes a Pen setting as a parameter (such as **Create Line statement**, **Create Pline statement**, **Set Style statement**, or **Set Map statement**).

See **Pen clause** for more information about Pen settings.

Example

```
Include "mapbasic.def"
Dim p_bus_route As Pen
p_bus_route = MakePen(3, 9, RED)
```

See Also:

CurrentPen( ) function, **Pen clause**, **StyleAttr( ) function**, **RGB( ) function**

MakeSymbol( ) function

Purpose

Returns a Symbol value, using a character from the MapInfo 3.0 symbol set. The MapInfo 3.0 symbol set is the symbol set that was originally published with MapInfo for Windows 3.0 and has been maintained in subsequent versions of MapInfo Professional. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

MakeSymbol(*shape*, *color*, *size*)

shape is a SmallInt value, 31 or larger (31 is invisible), specifying a symbol shape; standard symbol set provides symbols 31 through 67; see **Symbol clause** for a listing.

color is an integer RGB color value; see **RGB( ) function** for details.

size is a SmallInt value from 1 to 48, dictating the point size of the symbol.

Return Value

Symbol

Description

The **MakeSymbol( )** function returns a Symbol value. The return value can be assigned to a Symbol variable, or may be used as a parameter within a statement that takes a **Symbol clause** as a parameter (such as **Create Point statement**, **Set Map statement**, **Set Style statement**, or **Shade statement**).

To create a symbol from a character in a TrueType font, call the **MakeFontSymbol( ) function**.

To create a symbol from a bitmap file, call the **MakeCustomSymbol( ) function**.

See **Symbol clause** for more information about Symbol settings.

Example

```
Include "mapbasic.def"
Dim sym_marker As Symbol
sym_marker = MakeSymbol(44, RED, 16)
```

See Also:

CurrentSymbol( ) function, **MakeCustomSymbol( ) function**, **MakeFontSymbol( ) function**, **StyleAttr( ) function**, **Symbol clause**

Map statement

Purpose

Opens a new Map window. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Map From item [ , item ... ]
   [ Position ( x, y ) [ Units paperunits ] ]
   [ Width window_width [ Units paperunits ] ]
   [ Height window_height [ Units paperunits ] ]
   [ { Min | Max } ]
```

where *item* = *table* | [**GroupLayer** (“*friendly_name*” [, *item* …])

item is either the name of an open table, or a group layer

*friendly_name f*or each group layer is required but does not have to be unique, group layers may contain other group layers and/or tables, or be empty (no tables).

paperunits is the name of a paper unit (for example, “in”).

x, *y* specifies the position of the upper left corner of the Map window, in paper units.

window_width and *window_height* specify the size of the Map window, in paper units.

Description

The **Map** statement opens a new Map window. After you open a Map window, you can modify the window by issuing **Set Map statement**.

A GroupLayer keyword has been added to create nested group layers. Group layers are a special type of layer that allow users to organize other map layers into groups, similar to the way that folders and subfolders allow users to organize files. Group layers will make it easier to manage maps that have many layers. There are two main benefits to using groups:

1. Organizational benefits - layer lists are more manageable if they are organized into meaningful groups.
2. Efficiency benefits - once layers are organized into groups, subsequent operations such as “turn off all the street layers” can be performed in fewer clicks / fewer steps.

The table name specified must already be open. The table must also be mappable; in other words, the table must be able to have graphic objects associated with the records. The table does not need to actually contain any graphical objects, but the structure of the table must specify that objects may be attached.

The **Map** statement must specify at least one table, regardless of whether it is part of a group layer or not, since any Map window must contain at least one layer. Optionally, the **Map** statement can specify multiple table names (separated by commas) to open a multi-layer Map window. The first table name in the **Map** statement will be drawn last whenever the Map window is redrawn; thus, the

first table in the **Map** statement will always appear on top. Typically, tables with point objects appear earlier in **Map** statements, and tables with region (boundary) objects appear later in **Map** statements.

The default size of the resultant Map window is roughly a quarter of the screen size; the default position of the window depends on how many windows are currently on the screen. Optional **Position**, **Height**, and **Width** clauses allow you to control the size and position of the new Map window. The **Height** and **Width** clauses dictate the window size, in inches. Note that the **Position** clause specifies a position relative to the upper left corner of the MapInfo Professional application, not relative to the upper left corner of the screen.

If the **Map** statement includes the optional **Max** keyword, the new Map window is maximized, taking up all of the screen space available to MapInfo Professional. Conversely, if the **Map** statement includes the **Min** keyword, the window is minimized immediately.

Each Map window can have its own projection. MapInfo Professional decides a Map window's initial projection based on the native projection of the first table mapped. A user can change a map's projection by choosing the **Map > Options** command. A MapBasic program can change the projection by issuing a **Set Map statement**.

Examples

The following example opens a Map window three inches wide by two inches high, inset one inch from the upper left corner of the MapInfo Professional application. The map has two layers.

```
Open Table "world"
Open Table "cust1994" As customers
Map from customers, world
   Position (1,1) Width 3 Height 2
```

The following example opens a Map window that has group layers, some of which are nested (assume all tables have been opened first).

```
Map From
   GroupLayer (
      "Grid",
      GroupLayer ("Tropics", Tropic_Of_Capricorn, Tropic_Of_Cancer),
      Wgrid15
   ),
   GroupLayer (
      "World Places", WorldPlaces, WorldPlacesMajor, WorldPlaces_Capitals
   ),
   Airports,
   GroupLayer ("World Boundaries", world_Border),
   GroupLayer (
      "Roads", Roads, US_Primary_Roads, US_Secondary_Roads, US_Major_Roads
   ),
   GroupLayer ("Countries" Countries_small, Countries_large),
   Ocean
```

Groups layers have unique IDs like layers. Layer IDs may be numeric or table names. When numeric, they represent the order (reverse draw order) of the layer in the list from the top down. Group layers have numeric IDs that are part of a different sequence, but will also increase sequentially from the top down. In the example above the group layer and layer IDs would be as follows:

Group Layer	Layer ID
GroupLayer “Grid”	group 1
GroupLayer “Tropics”	group 2
Tropic_Of_Capricorn	layer 1
Tropic_Of_Cancer	layer 2
Wgrid15	layer 3
GroupLayer “World Places”	group 3
WorldPlaces	layer 4
WorldPlacesMajor	layer 5
WorldPlaces_Capitals	layer 6
Airports	layer 7
GroupLayer “World Boundaries”	group 4
world_Border	layer 8
GroupLayer “Roads”	group 5
Roads	layer 19
US_Primary_Roads	layer 10
US_Secondary_Roads	layer 11
US_Major_Roads	layer 12
GroupLayer “Countries”	group 6
Countries_small	layer 13
Countries_large	layer 14
Ocean	layer 15

See Also:

Add Map statement, **Remove Map statement**, **Set Map statement**, **Set Shade statement**, **Shade statement**

Map3DInfo( ) function

Purpose

Returns properties of a 3DMap window. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Map3DInfo(*window_id, attribute*)

window_id is an integer window identifier.

attribute is an integer code, indicating which type of information should be returned.

Return Value

Float, logical, or string, depending on the attribute parameter.

Description

The **Map3DInfo( )** function returns information about a 3DMap window.

The *window_id* parameter specifies which 3DMap window to query. To obtain a window identifier, call the **FrontWindow( ) function** immediately after opening a window, or call the **WindowID( ) function** at any time after the window's creation.

There are several numeric attributes that **Map3DInfo( )** can return about any given 3DMap window. The attribute parameter tells the **Map3DInfo( )** function which Map window statistic to return. The *attribute* parameter should be one of the codes from the following table; codes are defined in MAPBASIC.DEF.

Attribute	ID	Return Value
MAP3D_INFO_SCALE	1	Float result representing the 3DMaps scale factor.
MAP3D_INFO_RESOLUTION_X	2	Integer result representing the X resolution of the grid(s) in the 3DMap window.
MAP3D_INFO_RESOLUTION_Y	3	Integer result representing the Y resolution of the grid(s) in the 3DMap window.
MAP3D_INFO_BACKGROUND	4	Integer result representing the background color, see the RGB function.

Attribute	ID	Return Value
MAP3D_INFO_UNITS	5	String representing the map's abbreviated area unit name, for example, “mi” for miles.
MAP3D_INFO_LIGHT_X	6	Float result representing the x-coordinate of the Light in the scene.
MAP3D_INFO_LIGHT_Y	7	Float result representing the y-coordinate of the Light in the scene.
MAP3D_INFO_LIGHT_Z	8	Float result representing the z-coordinate of the Light in the scene.
MAP3D_INFO_LIGHT_COLOR	9	Integer result representing the Light color, see **RGB( ) function**.
MAP3D_INFO_CAMERA_X	10	Float result representing the x-coordinate of the Camera in the scene.
MAP3D_INFO_CAMERA_Y	11	Float result representing the y-coordinate of the Camera in the scene.
MAP3D_INFO_CAMERA_Z	12	Float result representing the z-coordinate of the Camera in the scene.
MAP3D_INFO_CAMERA_FOCAL_X	13	Float result representing the x-coordinate of the Cameras FocalPoint in the scene.
MAP3D_INFO_CAMERA_FOCAL_Y	14	Float result representing the y-coordinate of the Cameras FocalPoint in the scene.
MAP3D_INFO_CAMERA_FOCAL_Z	15	Float result representing the z-coordinate of the Cameras FocalPoint in the scene.
MAP3D_INFO_CAMERA_VU_1	16	Float result representing the first value of the ViewUp Unit Normal Vector.
MAP3D_INFO_CAMERA_VU_2	17	Float result representing the second value of the ViewUp Unit Normal Vector.
MAP3D_INFO_CAMERA_VU_3	18	Float result representing the third value of the ViewUp Unit Normal Vector.
MAP3D_INFO_CAMERA_VPN_1	19	Float result representing the first value of the ViewPlane Unit Normal Vector.
MAP3D_INFO_CAMERA_VPN_2	20	Float result representing the second value of the ViewPlane Unit Normal Vector.

Attribute	ID	Return Value
MAP3D_INFO_CAMERA_VPN_3	21	Float result representing the third value of the ViewPlane Unit Normal Vector.
MAP3D_INFO_CAMERA_CLIP_NEAR	22	Float result representing the cameras near clipping plane.
MAP3D_INFO_CAMERA_CLIP_FAR	23	Float result representing the cameras far clipping plane.

Example

Prints out all the state variables specific to the 3DMap window:

```
include "Mapbasic.def"
Print "MAP3D_INFO_SCALE: " + Map3DInfo(FrontWindow( ), MAP3D_INFO_SCALE)
Print "MAP3D_INFO_RESOLUTION_X: " + Map3DInfo(FrontWindow( ),
MAP3D_INFO_RESOLUTION_X)
Print "MAP3D_INFO_RESOLUTION_Y: " + Map3DInfo(FrontWindow( ),
MAP3D_INFO_RESOLUTION_Y)
Print "MAP3D_INFO_BACKGROUND: " + Map3DInfo(FrontWindow( ),
MAP3D_INFO_BACKGROUND)
Print "MAP3D_INFO_UNITS: " + Map3DInfo(FrontWindow( ), MAP3D_INFO_UNITS)
Print "MAP3D_INFO_LIGHT_X : " + Map3DInfo(FrontWindow( ),
MAP3D_INFO_LIGHT_X )
Print "MAP3D_INFO_LIGHT_Y : " + Map3DInfo(FrontWindow( ),
MAP3D_INFO_LIGHT_Y )
Print "MAP3D_INFO_LIGHT_Z: " + Map3DInfo(FrontWindow( ),
MAP3D_INFO_LIGHT_Z)
Print "MAP3D_INFO_LIGHT_COLOR: " + Map3DInfo(FrontWindow( ),
MAP3D_INFO_LIGHT_COLOR)
Print "MAP3D_INFO_CAMERA_X: " + Map3DInfo(FrontWindow( ),
MAP3D_INFO_CAMERA_X)
Print "MAP3D_INFO_CAMERA_Y : " + Map3DInfo(FrontWindow( ),
MAP3D_INFO_CAMERA_Y )
Print "MAP3D_INFO_CAMERA_Z : " + Map3DInfo(FrontWindow( ),
MAP3D_INFO_CAMERA_Z )
Print "MAP3D_INFO_CAMERA_FOCAL_X: " + Map3DInfo(FrontWindow( ),
MAP3D_INFO_CAMERA_FOCAL_X)
Print "MAP3D_INFO_CAMERA_FOCAL_Y: " + Map3DInfo(FrontWindow( ),
MAP3D_INFO_CAMERA_FOCAL_Y)
Print "MAP3D_INFO_CAMERA_FOCAL_Z: " + Map3DInfo(FrontWindow( ),
MAP3D_INFO_CAMERA_FOCAL_Z)
Print "MAP3D_INFO_CAMERA_VU_1: " + Map3DInfo(FrontWindow( ),
MAP3D_INFO_CAMERA_VU_1)
Print "MAP3D_INFO_CAMERA_VU_2: " + Map3DInfo(FrontWindow( ),
MAP3D_INFO_CAMERA_VU_2)
Print "MAP3D_INFO_CAMERA_VU_3: " + Map3DInfo(FrontWindow( ),
MAP3D_INFO_CAMERA_VU_3)
Print "MAP3D_INFO_CAMERA_VPN_1: " + Map3DInfo(FrontWindow( ),
MAP3D_INFO_CAMERA_VPN_1)
```

```
Print "MAP3D_INFO_CAMERA_VPN_2: " + Map3DInfo(FrontWindow( ),
MAP3D_INFO_CAMERA_VPN_2)
Print "MAP3D_INFO_CAMERA_VPN_3: " + Map3DInfo(FrontWindow( ),
MAP3D_INFO_CAMERA_VPN_3)
Print "MAP3D_INFO_CAMERA_CLIP_NEAR: " + Map3DInfo(FrontWindow( ),
MAP3D_INFO_CAMERA_CLIP_NEAR)
Print "MAP3D_INFO_CAMERA_CLIP_FAR: " + Map3DInfo(FrontWindow( ),
MAP3D_INFO_CAMERA_CLIP_FAR)
```

See Also:

Create Map3D statement, **Set Map3D statement**

MapperInfo( ) function

Purpose

Returns coordinate or distance information about a Map window. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

MapperInfo(*window_id*, *attribute*)

window_id is an integer window identifier.

attribute is an integer code, indicating which type of information should be returned. See table below for values.

Return Value

Float, logical, or string, depending on the attribute parameter.

Description

The **MapperInfo( )** function returns information about a Map window.

The *window_id* parameter specifies which Map window to query. To obtain a window identifier, call the **FrontWindow( ) function** immediately after opening a window, or call the **WindowID( ) function** at any time after the window's creation.

There are several numeric attributes that **MapperInfo( )** can return about any given Map window. The attribute parameter tells the **MapperInfo( )** function which Map window statistic to return. The *attribute* parameter should be one of the codes from the following table; codes are defined in MAPBASIC.DEF.

attribute setting	ID	MapperInfo( ) Return Value
MAPPER_INFO_ZOOM	1	The Map window's current zoom value (for example, the East-West distance currently displayed in the Map window), specified in MapBasic's current distance units; see **Set Distance Units statement**.
MAPPER_INFO_SCALE	2	The Map window's current scale, defined in terms of the number of map distance units (for example, Miles) per paper unit (for example, Inches) displayed in the window. This returns a value in MapBasic's current distance units.
MAPPER_INFO_CENTERX	3	The x-coordinate of the Map window's center.
MAPPER_INFO_CENTERY	4	The y-coordinate of the Map window's center.
MAPPER_INFO_MINX	5	The smallest x-coordinate shown in the window.
MAPPER_INFO_MINY	6	The smallest y-coordinate shown in the window.
MAPPER_INFO_MAXX	7	The largest x-coordinate shown in the window.
MAPPER_INFO_MAXY	8	The largest y-coordinate shown in the window.
MAPPER_INFO_LAYERS	9	Returns number of layers in the Map window as a SmallInt (excludes the cosmetic layer and group layers).
MAPPER_INFO_EDIT_LAYER	10	A SmallInt indicating the number of the currently-editable layer. A value of zero means that the Cosmetic layer is editable. A value of -1 means that no layer is editable.
MAPPER_INFO_XYUNITS	11	String representing the map's abbreviated coordinate unit name, for example, “degree”.
MAPPER_INFO_DISTUNITS	12	String representing the map's abbreviated distance unit name, for example, “mi” for miles.
MAPPER_INFO_AREAUNITS	13	String representing the map's abbreviated area unit name, for example, “sq mi” for square miles.
MAPPER_INFO_SCROLLBARS	14	Logical value indicating whether the Map window shows scrollbars.

attribute setting	ID	MapperInfo( ) Return Value
MAPPER_INFO_DISPLAY	15	Small integer, indicating what aspect of the map is displayed on the status bar. Corresponds to **Set Map Display**. Return value will be one of these: • MAPPER_INFO_DISPLAY_SCALE (0) • MAPPER_INFO_DISPLAY_ZOOM (1) • MAPPER_INFO_DISPLAY_POSITION (2)
MAPPER_INFO_NUM_THEMATIC	16	Small integer, indicating the number of thematic layers in this Map window.
MAPPER_INFO_COORDSYS_CLAUSE	17	string result, indicating the window's **CoordSys clause**.
MAPPER_INFO_COORDSYS_NAME	18	String result, representing the name of the map's CoordSys as listed in MAPINFOW.PRJ (but without the optional “\p…” suffix that appears in MAPINFOW.PRJ). Returns empty string if CoordSys is not found in MAPINFOW.PRJ.
MAPPER_INFO_MOVE_DUPLICATE_NODES	19	Small integer, indicating whether duplicate nodes should be moved when reshaping objects in this Map window. If the value is 0, duplicate nodes are not moved. If the value is 1, any duplicate nodes within the same layer will be moved. To return to using the default from the map preferences, call **Set Map Move Nodes Default**.
MAPPER_INFO_DIST_CALC_TYPE	20	Small integer, indicating type of calculation to use for distance, length, perimeter, and area calculations for mapper. Corresponds to **Set Map Distance Type**. Return values include: • MAPPER_INFO_DIST_SPHERICAL (0) • MAPPER_INFO_DIST CARTESIAN (1)

attribute setting	ID	MapperInfo( ) Return Value
MAPPER_INFO_DISPLAY_DMS	21	Small integer, indicating whether the map displays coordinates in decimal degrees, DMS (degrees, minutes, seconds), or Military Grid Reference System or USNG (US National Grid) format. Return value is one of the following: • MAPPER_INFO_DISPLAY_DECIMAL (0) • MAPPER_INFO_DISPLAY_DEGMINSEC (1) • MAPPER_INFO_DISPLAY_MGRS (2) Military Grid Reference System • MAPPER_INFO_DISPLAY_USNG_WGS84 (3) US National Grid NAD 83/WGS 84 • MAPPER_INFO_DISPLAY_USNG_NAD27 (4) US National Grid NAD 27
MAPPER_INFO_COORDSYS_CLAUSE_WITH_BOUNDS	22	String result, indicating the window's **CoordSys clause** including the bounds.
MAPPER_INFO_CLIP_TYPE	23	The type of clipping being implemented. Choices include: • MAPPER_INFO_CLIP_DISPLAY_ALL (0) • MAPPER_INFO_CLIP_DISPLAY_POLYOBJ (1) • MAPPER_INFO_CLIP_OVERLAY (2)
MAPPER_INFO_CLIP_REGION	24	Returns a string to indicate if a clip region is enabled. Returns the string “on” if a clip region is enabled in the Mapper window. Otherwise, it returns the string “off”.
MAPPER_INFO_REPROJECTION	25	String value indicating the current value of the reprojection mode. The value can be either: • None - Never reproject the map. • Always - Always reproject the map. • Auto - Optimize whether or not to reproject the map; allow MapInfo Professional to decide.
MAPPER_INFO_RESAMPLING	26	String value indicating the method for calculating the pixel values of the source image being reprojected. The value can be either: • CubicConvolution • NearestNeighbor
MAPPER_INFO_MERGE_MAP	27	String value: the string of MapBasic statements that a user needs to merge one map window into the current map window.

attribute setting	ID	MapperInfo( ) Return Value
MAPPER_INFO_ALL_LAYERS	28	This will return the count of layers and group layers (includes all nested layers and group layers)
MAPPER_INFO_GROUPLAYERS	29	This will return the count of all group layers (includes nested group layers)
MAPPER_INFO_NUM_ADORNMENTS	200	This will return an integer representing the number of adornments associated with a mapper. Use some value suitably outside the normal range for MapperInfo, such as 100.
MAPPER_INFO_ADORNMENT+n	200	This will return the WindowID of a given adornment associated with the Mapper.

When you call **MapperInfo( )** to obtain coordinate values (for example, by specifying MAPPER_INFO_CENTERX as the attribute), the value returned represents a coordinate in MapBasic's current coordinate system, which may be different from the coordinate system of the Map window. Use the **Set CoordSys statement** to specify a different coordinate system.

A setting for each Map window and providing MapBasic support to set and get the current setting for each mapper. During Reshape, the move duplicate nodes can be set to none or move all duplicates within the same layer.

Whenever a new Map window is created, the initial move duplicate nodes setting will be retrieved from the mapper preference (Options / Preference / Map Window / Move Duplicate Nodes in).

An existing Map window can be queried for its current Move Duplicate Nodes setting using a new attribute in **MapperInfo( )** function.

The current state can be changed for a mapper window using the **Set Map statement**.

Coordinate Value Returns

MapperInfo( ) does not return coordinates (for example MINX, MAXX, MINY, MAXY) in the units set for the map window. Instead, the coordinate values are returned in the units of the internal coordinate system of the MapInfo Professional session or the MapBasic application that calls the function (if the coordinate system was changed within the application). Also, the MAPPER_INFO_XYUNITS attribute returns the units that are used to display the cursor location in the Status Bar (set by using **Set Map Window Frontwindow( ) XY Units**).

Clip Region Information

Beginning with MapInfo Professional 6.0, there are three methods that are used for Clip Region functionality. The MAPPER_INFO_CLIP_OVERLAY (2) method is the method that has been the only option until MapInfo Professional 6.0. Using this method, the **Overlap( ) function** (**Object > Erase Outside**) is used internally. Since the **Overlap( ) function** cannot produce result with Text objects, text objects are never clipped. For Point objects, a simple point in region test is performed to

either include or exclude the Point. Label objects are treated similar to Point objects and are either completely displayed (is the label point is inside the clip region object) or ignored. Since the clipping is done at the spatial object level, styles (wide lines, symbols, text) are never clipped.

The MAPPER_INFO_CLIP_DISPLAY_ALL (0) method uses the Windows Display to perform the clipping. All object types are clipped. Thematics, rasters, and grids are also clipped. Styles (wide lines, symbols, text) are always clipped. This is the default clipping type.

The MAPPER_INFO_CLIP_DISPLAY_POLYOBJ (1) uses the Windows Display to selectively perform clipping which mimics the functionality produced by MAPPER_INFO_CLIP_OVERLAY (2). Windows Display Clipping is used to clip all Poly Objects (Regions and Polylines) and objects than can be converted to Poly Objects (rectangles, rounded rectangles, ellipses, and arcs). These objects will always have their symbology clipped. Points, Labels, and Text are treated as they would be in the MAPPER_INFO_CLIP_OVERLAY (2) method. In general, this method should provide better performance that the MAPPER_INFO_CLIP_OVERLAY (2) method.

Error Conditions

ERR_BAD_WINDOW (590) error generated if parameter is not a valid window number.

ERR_FCN_ARG_RANGE (644) error generated if an argument is outside of the valid range.

ERR_WANT_MAPPER_WIN (313) error generated if window id is not a Map window.

LayerInfo( ) function, **Set Distance Units statement**, **Set Map statement**

Maximum( ) function

Purpose

Returns the larger of two numbers. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Maximum(*num_expr*, *num_expr*)

num_expr is a numeric expression.

Return Value

Float

Description

The **Maximum( )** function returns the larger of two numeric expressions.

Example

```
Dim x, y, z As Float
x = 42
y = 27
z = Maximum(x, y)
```

```
' z now contains the value: 42
```

See Also:

Minimum( ) function

MBR( ) function

Purpose

Returns a rectangle object, representing the minimum bounding rectangle (MBR) of another object. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

MBR(*obj_expr*)

obj_expr is an object expression.

Return Value

Object (a rectangle)

Description

The **MBR( )** function calculates the minimum bounding rectangle (MBR) which encompasses the specified *obj_expr* object.

A minimum bounding rectangle is defined as being the smallest rectangle which is large enough to encompass a particular object. In other words, the MBR of the United States extends east to the eastern tip of Maine, south to the southern tip of Hawaii, west to the western tip of Alaska, and north to the northern tip of Alaska.

The MBR of a point object has zero width and zero height.

Example

```
Dim o_mbr As Object
Open Table "world"
Fetch First From world
o_mbr = MBR(world.obj)
```

See Also:

Centroid( ) function, **CentroidX( ) function**, **CentroidY( ) function**

Menu Bar statement

Purpose

Shows or hides the menu bar. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Menu Bar { Hide | Show }
```

Description

The **Menu Bar** statement shows or hides MapInfo Professional's menu bar. An application might hide the menu bar in order to provide more screen room for windows.

Following a **Menu Bar Hide** statement, the menu bar remains hidden until a **Menu Bar Show** statement is executed. Since users can be severely handicapped without the menu bar, you should be very careful when using the **Menu Bar Hide** statement. Every **Menu Bar Hide** statement should be followed (eventually) by a **Menu Bar Show** statement.

While the menu bar is hidden, MapInfo Professional ignores any menu-related hotkeys. For example, an MapInfo Professional user might ordinarily press Ctrl-O to bring up the Open dialog box; but while the menu bar is hidden, MapInfo Professional ignores the Ctrl-O hotkey.

See Also:

Alter Menu Bar statement, **Create Menu Bar statement**

MenuItemInfoByHandler( ) function

Purpose

Returns information about a MapInfo Professional menu item.

Syntax

```
MenuItemInfoByHandler( handler, attribute )
```

handler is either a string (containing the name of a handler procedure specified in a **Calling** clause) or an integer (which was specified as a constant in a **Calling** clause).

attribute is an integer code indicating which attribute to return; see table below.

Description

The handler parameter can be an integer or a string. If you specify a string (a procedure name), and if two or more menu items call that procedure, MapInfo Professional returns information about the first menu item that calls the procedure. If you need to query multiple menu items that call the same

handler procedure, give each menu item an ID number (for example, using the optional **ID** clause in the **Create Menu statement**), and call **MenuItemInfoByID( ) function** instead of calling **MenuitemInfoByHandler( )**.

The attribute parameter is a numeric code (defined in MAPBASIC.DEF) from the following table:

attribute setting	ID	Return value
MENUITEM_INFO_ENABLED	1	Logical: TRUE if the menu item is enabled.
MENUITEM_INFO_CHECKED	2	Logical: TRUE if the menu item is checkable and currently checked; also return TRUE if the menu item has alternate menu text (for example, if the menu item toggles between **Show…** and **Hide…**), and the menu item is in its “show” state. Otherwise, return FALSE.
MENUITEM_INFO_CHECKABLE	3	Logical: TRUE if this menu item is checkable (specified by the “!” prefix in the menu text).
MENUITEM_INFO_SHOWHIDEABLE	4	Logical: TRUE if this menu item has alternate menu text (for example, if the menu item toggles between **Show…** and **Hide…**). An item has alternate text if it was created with “!” at the beginning of the menu item text (in a **Create Menu statement** or **Alter Menu statement**) and it has a caret (^) in the string.
MENUITEM_INFO_ACCELERATOR	5	String: The code sequence for the menu item's accelerator (for example, “/W^Z” or “/W#%119”) or an empty string if the menu item has no accelerator. For details on menu accelerators, see **Create Menu statement**.
MENUITEM_INFO_TEXT	6	String: the full text used (for example, in a **Create Menu statement**) to create the menu item.
MENUITEM_INFO_HELPMSG	7	String: the menu item's help message (as specified in the **HelpMsg** clause in **Create Menu statement**) or empty string if the menu item has no help message.

attribute setting	ID	Return value
MENUITEM_INFO_HANDLER	8	Integer: The menu item's handler number. If the menu item's **Calling** clause specified a numeric constant (for example, `Calling M_FILE_SAVE`), this call returns the value of the constant. If the **Calling** clause specified "OLE", "DDE", or the name of a procedure, this call returns a unique integer (an internal handler number) which can be used in subsequent calls to **MenuitemInfoByHandler( )** or in the **Run Menu Command statement**.
MENUITEM_INFO_ID	9	Integer: The menu ID number (specified in the optional **ID** clause in a **Create Menu statement**), or 0 if the menu item has no ID.

See Also:

MenuItemInfoByID( ) function

MenuItemInfoByID( ) function

Purpose

Returns information about a MapInfo Professional menu item.

Syntax

MenuItemInfoByID(*menuitem_ID*, *attribute*)

menuitem_ID is an integer menu ID (specified in the ID clause in Create Menu).

attribute is an integer code indicating which attribute to return.

Description

This function is identical to the **MenuItemInfoByHandler( ) function**, except that the first argument to this function is an integer ID.

Call this function to query the status of a menu item when you know the ID of the menu item you need to query. Call the **MenuItemInfoByHandler( ) function** to query the status of a menu item if you would rather identify the menu item by its handler.

The *attribute* argument is a code from MAPBASIC.DEF, such as MENUITEM_INFO_CHECKED (2). For a listing of codes you can use, see **MenuItemInfoByHandler( ) function**.

See Also:

MenuItemInfoByHandler( ) function

Metadata statement

Purpose

Manages a table's metadata. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax 1

```
Metadata Table table_name
   { SetKey key_name To key_value |
     DropKey key_name [ Hierarchical ] |
     SetTraverse starting_key_name [ Hierarchical ]
     Into ID traverse_ID_var }
```

table_name is the name of an open table.

key_name is a string, representing the name of a metadata key. The string must start with a backslash ("\"), and it cannot end with a backslash.

key_value is a string up to 239 characters long, representing the value to assign to the key.

starting_key_name is a string representing the first key name to retrieve from the table. To set up the traversal at the very beginning of the list of keys, specify "\" (backslash).

traverse_ID_var is the name of an integer variable; MapInfo Professional stores a traversal ID in the variable, which you can use in subsequent **Metadata Traverse…** statements.

Syntax 2

```
Metadata Traverse traverse_ID
   { Next Into Key key_name_var In key_value_var |
    Destroy }
```

traverse_ID is an integer value (such as the value of the *traverse_ID_var* variable described above).

key_name_var is the name of a string variable; MapInfo Professional stores the fetched key's name in this variable.

key_value_var is the name of a string variable; MapInfo Professional stores the fetched key's value in this variable.

Description

The Metadata statement manages the metadata stored in MapInfo tables. Metadata is information that is stored in a table's .TAB file, instead of being stored as rows and columns.

Each table can have zero or more keys. Each key represents an information category, such as an author's name, a copyright notice, etc. Each key has a string value associated with it. For example, a key called "\Copyright" might have the value "Copyright 2001 Pitney Bowes Software Inc. Corporation." For more information about Metadata, see the *MapBasic User Guide*.

Modifying a Table's Metadata

To create, modify, or delete metadata, use Syntax 1. The following clauses apply:

SetKey

Assigns a value to a metadata key. If the key already exists, MapInfo Professional assigns it a new value. If the key does not exist, MapInfo Professional creates a new key. When you create a new key, the changes take effect immediately; you do not need to perform a Save operation.

```
MetaData Table Parcels SetKey "\Info\Date" To Str$(CurDate( ))
```

MapInfo Professional automatically creates a metadata key called “\IsReadOnly” (with a default value of “FALSE”) the first time you add a metadata key to a table. The \IsReadOnly key is a special key, reserved for internal use by MapInfo Professional.

DropKey

Deletes the specified key from the table. If you include the **Hierarchical** keyword, MapInfo Professional deletes the entire metadata hierarchy at and beneath the specified key. For example, if a table has the keys “\Info\Author” and “\Info\Date” you can delete both keys with the following statement:

```
MetaData Table Parcels DropKey "\Info" Hierarchical
```

Reading a Table's Metadata

To read a table's metadata values, use the **SetTraverse** clause to initialize a traversal, and then use the **Next** clause to fetch key values. After you are finished fetching key values, use the **Destroy** clause to free the memory used by the traversal. The following clauses apply:

SetTraverse

Prepares to traverse the table's keys, starting with the specified key. To start at the beginning of the list of keys, specify “\” as the starting key name. If you include the **Hierarchical** keyword, the traversal can hierarchically fetch every key. If you omit the **Hierarchical** keyword, the traversal is flat, meaning that MapInfo Professional will only fetch keys at the root level (for example, the traversal will fetch the “\Info” key, but not the “\Info\Date” key).

Next Into Key… Into Value…

Attempts to read the next key. If there is a key to read, MapInfo Professional stores the key's name in the *key_name_var* variable, and stores the key's value in the *key_value_var* variable. If there are no more keys to read, MapInfo Professional stores empty strings in both variables.

Destroy

Ends the traversal, and frees the memory that was used by the traversal.

> ⓘ A hierarchical metadata traversal can traverse up to ten levels of keys (for example, "\One\Two\Three\Four\Five\Six\Seven\Eight\Nine\Ten") if you begin the traversal at the root level ("\"). If you need to retrieve a key that is more than ten levels deep, begin the traversal at a deeper level (for example, begin the traversal at "\One\Two\Three\Four\Five").

Example

The following procedure reads all metadata values from a table; the table name is specified by the caller. This procedure prints the key names and key values to the Message window.

```
Sub Print_Metadata(ByVal table_name As String)
   Dim i_traversal As Integer
   Dim s_keyname, s_keyvalue As String

   ' Initialize the traversal:
   Metadata Table table_name
      SetTraverse "\" Hierarchical Into ID i_traversal

   ' Attempt to fetch the first key:
   Metadata Traverse i_traversal
   Next Into Key s_keyname Into Value s_keyvalue

   ' Now loop for as long as there are key values;
   ' with each iteration of the loop, retrieve
   ' one key, and print it to the Message window.
   Do While s_keyname <> ""
      Print " "
      Print "Key name: " & s_keyname
      Print "Key value: " & s_keyvalue

   Metadata Traverse i_traversal
      Next Into Key s_keyname Into Value s_keyvalue
   Loop

   ' Release this traversal to free memory:
   MetaData Traverse i_traversal Destroy

End Sub
```

See Also:

GetMetadata$( ) function, **TableInfo( ) function**

MGRSToPoint( ) function

Purpose

Converts a string representing an MGRS (Military Grid Reference System) coordinate into a point object in the current MapBasic coordinate system. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

```
MGRSToPoint( string )
```

string is a string expression representing an MGRS coordinate.

The default Longitude/Latitude coordinate system is used as the initial selection.

Return Value

Object

Description

The returned point will be in the current MapBasic coordinate system, which by default is Long/Lat (no datum). For the most accurate results when saving the resulting points to a table, set the MapBasic coordinate system to match the destination table's coordinate system before calling **MGRSToPoint( )**. This will prevent MapInfo Professional from doing an intermediate conversion to the datumless Long/Lat coordinate system, which can cause a significant loss of precision.

Example

Example 1:

```
dim obj1 as Object
dim s_mgrs As String
dim obj2 as Object
obj1 = CreatePoint(-74.669, 43.263)
s_mgrs = PointToMGRS$(obj1)
obj2 = MGRSToPoint(s_mgrs)
```

Example 2:

```
Open Table "C:\Temp\MyTable.TAB" as MGRSfile
' When using the PointToMGRS$( ) or MGRSToPoint( ) functions,
' it is very important to make sure that the current MapBasic
' coordsys matches the coordsys of the table where the
' point object is being stored.
'Set the MapBasic coordsys to that of the table used
Set CoordSys Table MGRSfile
'Update a Character column (for example COL2) with MGRS strings from
'a table of points
Update MGRSfile
   Set Col2 = PointToMGRS$(obj)
```

```
'Update two float columns (Col3 & Col4) with
'CentroidX & CentroidY information
'from a character column (Col2) that contains MGRS strings.
Update MGRSfile
   Set Col3 = CentroidX(MGRSToPoint(Col2))
Update mgrstestfile ' MGRSfile
   Set Col4 = CentroidY(MGRSToPoint(Col2))
Commit Table MGRSfile
Close Table MGRSfile
```

See Also:

PointToMGRS$( ) function

Mid$( ) function

Purpose

Returns a string extracted from the middle of another string. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Mid$(*string_expr*, *position*, *length*)

string_expr is a string expression.

position is a numeric expression, indicating a starting position in the string.

length is a numeric expression, indicating the number of characters to extract.

Return Value

String

Description

The **Mid$( )** function returns a substring copied from the specified *string_expr* string.

Mid$( ) copies length characters from the *string_expr* string, starting at the character position indicated by position. A position value less than or equal to one tells MapBasic to copy from the very beginning of the *string_expr* string.

If the *string_expr* string is not long enough, there may not be length characters to copy; thus, depending on all of the parameters, the **Mid$( )** may or may not return a string length characters long. If the position parameter represents a number larger than the number of characters in *string_expr*, **Mid$( )** returns a null string. If the length parameter is zero, **Mid$( )** returns a null string. If the length or position parameters are fractional, MapBasic rounds to the nearest integer.

Example

```
Dim str_var, substr_var As String
str_var = "New York City"
```

```
substr_var = Mid$(str_var, 10, 4)

' substr_var now contains the string "City"
```

See Also:

InStr( ) function, **Left$( ) function**, **Right$( ) function**

MidByte$( ) function

Purpose

Accesses individual bytes of a string on a system with a double-byte character system. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

MidByte$(*string_expr*, *position*, *length*)

string_expr is a string expression.

position is an integer numeric expression, indicating a starting position in the string.

length is an integer numeric expression, indicating the number of bytes to return.

Return Value

String

Description

The **MidByte$( )** function returns individual bytes of a string.

Use the **MidByte$( )** function when you need to extract a range of bytes from a string, and the application is running on a system that uses a double-byte character set (DBCS systems). For example, the Japanese version of Microsoft Windows uses a double-byte character system.

On systems with single-byte character sets, the results returned by the **MidByte$( )** function are identical to the results returned by the **Mid$( ) function**.

See Also:

InStr( ) function, **Left$( ) function**, **Right$( ) function**

Minimum( ) function

Purpose

Returns the smaller of two numbers. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Minimum(*num_expr*, *num_expr*)

num_expr is a numeric expression.

Return Value

Float

Description

The **Minimum( )** function returns the smaller of two numeric expressions.

Example

```
Dim x, y, z As Float
x = 42
y = -100
z = Minimum(x, y)

' z now contains the value: -100
```

See Also:

Maximum( ) function

Minute function

Purpose

Returns the minute component of a Time. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

```
Minute (Time)
```

Return Value

Number

Example

Copy this example into the MapBasic window for a demonstration of this function.

```
dim X as time
dim iMin as integer
X = CurDateTime()
iMin = Minute(X)
Print iMin
```

Month( ) function

Purpose

Returns the month component (1 - 12) of a date value. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Month(*date_expr*)

date_expr is a date expression.

Return Value

SmallInt value from 1 to 12, inclusive.

Description

The **Month( )** function returns an integer, representing the month component (one to twelve) of the specified date.

Examples

The following example shows how you can extract just the month component from a particular date value, using the **Month( )** function.

```
If Month(CurDate( )) = 12 Then
   '
   ' ... then it is December...
   '
End If
```

You can also use the **Month( )** function within the SQL Select statement. The following Select statement extracts only particular rows from the Orders table. This example assumes that the Orders table has a Date column, called Order_Date. The Select statement's Where clause tells MapInfo Professional to only select the orders from December of 1993.

```
Open Table "orders"
Select *
   From orders
   Where Month(orderdate) = 12 And Year(orderdate) = 1993
```

See Also:

CurDate( ) function, **Day( ) function**, **Weekday( ) function**, **Year( ) function**

Nearest statement

Purpose

Find the object in a table that is closest to a particular object. The result is a 2-point Polyline object representing the closest distance. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Nearest [ N | All ]
   From { Table fromtable | Variable fromvar }
   To totable Into intotable
   [ Type { Spherical | Cartesian }]
   [ Ignore [ Contains ] [ Min min_value ] [ Max max_value ]
      Units unitname ] [ Data clause ]
```

N is an optional parameter representing the number of “nearest” objects to find. The default is 1. If **All** is used, then a distance object is created for every combination.

fromtable represents a table of objects that you want to find closest distances from.

fromvar represents a MapBasic variable representing an object that you want to find the closest distances from.

totable represents a table of objects that you want to find closest distances to.

intotable represents a table to place the results into.

min_value is the minimum distance to include in the results.

max_value is the maximum distance to include in the results.

unitname is string representing the name of a distance unit (for example, “km”) used for *min_value* and/or *max_value*.

clause is an expression that specifies the tables that the results come from.

Description

The **Nearest** statement finds all the objects in the *fromtable* that are nearest to a particular object. Every object in the *fromtable* is considered. For each object in the *fromtable*, the nearest object in the *totable* is found. If *N* is defined, then the *N* nearest objects in *totable* are found. A two-point Polyline object representing the closest points between the *fromtable* object and the chosen *totable* object is placed in the *intotable*. If **All** is specified, then an object is placed in the *intotable* representing the distance between the *fromtable* object and each *totable* object.

If there are multiple objects in the *totable* that are the same distance from a given *fromtable* object, then only one of them may be returned. If multiple objects are requested (for example, if *N* is greater than 1), then objects of the same distance will fill subsequent slots. If the tie exists at the second closest object, and three objects are requested, then the object will become the third closest object.

The types of the objects in the *fromtable* and *totable* can be anything except Text objects. For example, if both tables contain Region objects, then the minimum distance between Region objects is found, and the two-point Polyline object produced represents the points on each object used to calculate that distance. If the Region objects intersect, then the minimum distance is zero, and the two-point Polyline returned will be degenerate, where both points are identical and represent a point of intersection.

The distances calculated do not take into account any road route distance. It is strictly a “as the bird flies” distance.

Type is the method used to calculate the distances between objects. It can either be **Spherical** or **Cartesian**. The type of distance calculation must be correct for the coordinate system of the *intotable* or an error will occur. If the coordinate system of the *intotable* is NonEarth and the distance method is Spherical, then an error will occur. If the coordinate system of the *intotable* is Latitude/Longitude, and the distance method is Cartesian, then an error will occur.

The **Ignore** clause limits the distances returned. Any distances found which are less than or equal to *min_value* or greater than *max_value* are ignored. *min_value* and *max_value* are in the distance unit signified by *unitname*. If *unitname* is not a valid distance unit, an error will occur. One use of the **Min** distance could be to eliminate distances of zero. This may be useful in the case of two point tables to eliminate comparisons of the same point. For example, if there are two point tables representing Cities, and we want to find the closest cities, we may want to exclude cases of the same city. The entire **Ignore** clause is optional, as are the **Min** and **Max** subclauses within it.

The **Max** distance can be used to limit the objects to consider in the *totable*. This may be most useful in conjunction with *N* or **All**. For example, we may want to search for the five airports that are closest to a set of cities (where the *fromtable* is the set of cities and the *totable* is a set of airports), but we don't care about airports that are farther away than 100 miles. This may result in less than five airports being returned for a given city. This could also be used in conjunction with the **All** parameter, where we would find all airports within 100 miles of a city. Supplying a **Max** parameter can improve the performance of the **Nearest** statement, since it effectively limits the number of *totable* objects that are searched.

The effective distances found are strictly greater than the *min_value* and less than or equal to the *max_value*:

```
min_value < distance <= max_value
```

This can allow ranges or distances to be returned in multiple passes using the **Nearest** statement. For example, the first pass may return all objects between 0 and 100 miles, and the second pass may return all objects between 100 and 200 miles, and the results should not contain duplicates (for example, a distance of 100 should only occur in the first pass and never in the second pass).

Normally, if one object is contained within another object, the distance between the objects is zero. For example, if the *fromtable* is WorldCaps and the *totable* is World, then the distance between London and the United Kingdom would be zero. If the **Contains** flag is set within the **Ignore** clause, then the distance will not be automatically be zero. Instead, the distance from London to the boundary of the United Kingdom will be returned. In effect, this will treat all closed objects, such as regions, as polylines for the purpose of this operation.

Data Clause

The **Data** clause can be used to mark which *fromtable* object and which *totable* object the result came from.

```
Data IntoColumn1=column1, IntoColumn2=column2
```

The *IntoColumn* on the left hand side of the equals must be a valid column in *intotable*. The column name on the right hand side of the equals sign must be a valid column name from either *totable* or *fromtable*. If the same column name exists in both *totable* and *fromtable*, then the column in *totable* will be used (e.g., *totable* is searched first for column names on the right hand side of the equals sign). To avoid any conflicts such as this, the column names can be qualified using the table alias:

```
Data name1=states.state_name, name2=county.state_name
```

To fill a column in the *intotable* with the distance, we can either use the **Table > Update Column** functionality from the menu or use the **Update statement**.

Examples

Assume that we have a point table representing locations of ATM machines and that there are at least two columns in this table: Business, which represents the name of the business which contains the ATM; and Address, which represents the street address of that business. Assume that the current selection represents our current location. Then the following will find the closest ATM to where we currently are:

```
Nearest From Table selection To atm Into result Data
where=Business,address=Address
```

If we wanted to find the closest five ATM machines to our current location:

```
Nearest 5 From Table selection To atm Into result Data
where=Business,address=Address
```

If we want to find all ATM machines within a 5 mile radius:

```
Nearest All From Table selection To atm Into result Ignore Max 5 Units
"mi" Data where=buisness,address=address
```

Assume we have a table of house locations (the *fromtable*) and a table representing the coastline (the *totable*). To find the distance from a given house to the coastline:

```
Nearest From Table customer To coastline Into result Data
who=customer.name,
where=customer.address,coast_loc=coastline.county,type=coastline.designat
ion
```

If we don't care about customer locations which are greater than 30 miles from any coastline:

```
Nearest From Table customer To coastline Into result Ignore Max 30 Units
"mi" Data who=customer.name,
where=customer.address,coast_loc=coastline.county,
type=coastline.designation
```

Assume we have a table of cities (the *fromtable*) and another table of state capitals (the *totable*), and we want to find the closest state capital to each city, but we want to ignore the case where the city in the *fromtable* is also a state capital:

```
Nearest From Table uscty_1k To usa_caps Into result Ignore Min 0 Units
"mi" Data city=uscty_1k.name,capital=usa_caps.capital
```

See Also:

Farthest statement, **CartesianObjectDistance( ) function**, **ObjectDistance( ) function**, **SphericalObjectDistance( ) function**, **CartesianConnectObjects( ) function**, **ConnectObjects( ) function**, **SphericalConnectObjects( ) function**

Note statement

Purpose

Displays a simple message in a dialog box. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Note *message*

message is an expression to be displayed in a dialog box.

Description

The **Note** statement creates a dialog box to display a message. The dialog box contains an **OK** button; the message dialog box remains on the screen until the user clicks the **OK** button.

The message expression does not need to be a string expression. If message is an object expression, MapBasic will automatically produce an appropriate string (for example, “Region”) for display in the Note dialog box. If the message expression is a string, the string can be up to 300 characters long, and can occupy up to 6 rows.

Example

```
Note "Total # of records processed: " + Str$( i_count )
```

See Also:

Ask( ) function, **Dialog statement**, **Print statement**

NumAllWindows( ) function

Purpose

Returns the number of windows owned by MapInfo Professional, including special windows such as ButtonPads and the Info window. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

```
NumAllWindows( )
```

Return Value

SmallInt

Description

The **NumAllWindows( )** function returns the number of windows owned by MapInfo Professional.

To determine the number of document windows opened by MapInfo Professional (Map, Browse, Graph, and Layout windows), call **NumWindows( )**.

See Also:

NumWindows( ) function, **WindowID( ) function**

NumberToDate( ) function

Purpose

Returns a Date value, given an integer. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

NumberToDate(*numeric_date*)

numeric_date is an eight-digit integer in the form YYYYMMDD (for example, 19951231).

Return Value

Date

Description

The **NumberToDate( )** function returns a Date value represented by an eight-digit integer. For example, the following function call returns a Date value of December 31, 2006:

```
NumberToDate(20061231)
```

Example

The following example subtracts one Date value from another Date. The result of the subtraction is the number of days between the two dates.

```
Dim i_elapsed As Integer

i_elapsed = CurDate( ) - NumberToDate(20060101)

' i_elapsed now contains the number of days
' since January 1, 2006
```

See Also:

StringToDate( ) function

NumberToDateTime function

Purpose

Returns a DateTime value. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

```
NumberToDateTime( numeric_datetime )
```

numeric_datetime is an seventeen-digit integer in the form YYYYMMDDHHMMSSFFF. For example, 20070301214237582 represents March 1, 2007 9:42:37.582 PM.

Return Value

Date/Time

Example

Copy this example into the MapBasic window for a demonstration of this function.

```
dim fNum as float
dim Y as datetime
fNum = 20070301214237582
Y = NumbertoDateTime (fNum)
Print FormatDate$(Y)
Print FormatTime$(Y,"hh:mm:ss.fff tt")
```

NumberToTime function

Purpose

Returns a Time value. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

```
NumberToTime( numeric_time )
```

numeric_time is an nine-digit integer in the form HHMMSSFFF. For example, 214237582 represents 9:42:37.582 P.M.

Return Value

Time

Example

Copy this example into the MapBasic window for a demonstration of this function.

```
dim fNum as float
dim Y as time
fNum = 214237582
Y = NumberToTime(fNum)
Print FormatTime$(Y,"hh:mm:ss.fff tt")
```

NumCols( ) function

Purpose

Returns the number of columns in a specified table. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

NumCols(*table*)

table is the name of an open table.

Return Value

SmallInt

Description

The **NumCols( )** function returns the number of columns contained in the specified open table.

The number of columns returned by **NumCols( )** does not include the special column known as Object (or Obj for short), which refers to the graphical objects attached to mappable tables. Similarly, the number of columns returned does not include the special column known as RowID.

If a table has temporary columns (for example, because of an **Add Column statement**), the number returned by **NumCols( )** includes the temporary column(s).

Error Conditions

ERR_TABLE_NOT_FOUND (405) error generated if the specified table is not available.

Example

```
Dim i_counter As Integer
Open Table "world"
i_counter = NumCols(world)
```

See Also:

ColumnInfo( ) function, **NumTables( ) function**, **TableInfo( ) function**

NumTables( ) function

Purpose

Returns the number of tables currently open. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

NumTables( )

Return Value

SmallInt

Description

The **NumTables( )** function returns the number of tables that are currently open.

A street-map table may consist of two “companion” tables. For example, when you open the Washington, DC street map named DCWASHS, MapInfo Professional secretly opens the two companion tables DCWASHS1.TAB and DCWASHS2.TAB. However, MapInfo Professional treats the DCWASHS table as a single table; for example, the Layer Control window shows only the table name DCWASHS. Similarly, the **NumTables( )** function counts a street map as a single table, although it may actually be composed of two companion tables.

Example

```
If NumTables( ) < 1 Then
   Note "You must open a table before continuing."
End If
```

See Also:

Open Table statement, **TableInfo( ) function**, **ColumnInfo( ) function**

NumWindows( ) function

Purpose

Returns the number of open document windows (Map, Browse, Graph, Layout). You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

NumWindows( )

Return Value

SmallInt

Description

The **NumWindows( )** function returns the number of Map, Browse, Graph, and Layout windows that are currently open. The result is independent of whether windows are minimized or not.

To determine the total number of windows opened by MapInfo Professional (including ButtonPads and special windows such as the Info window), call **NumAllWindows( )**.

Example

```
Dim num_open_wins As SmallInt
num_open_wins = NumWindows( )
```

See Also:

NumAllWindows( ) function, **WindowID( ) function**

ObjectDistance( ) function

Purpose

Returns the distance between two objects. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

ObjectDistance(*object1*, *object2*, *unit_name*)

object1 and *object2* are object expressions.

unit_name is a string representing the name of a distance unit.

Return Value

Float

Description

ObjectDistance( ) returns the minimum distance between *object1* and *object2* using a spherical calculation method with the return value in *unit_name*. If the calculation cannot be done using a spherical distance method (e.g., if the MapBasic Coordinate System is NonEarth), then a cartesian distance method will be used.

ObjectGeography( ) function

Purpose

Returns coordinate or angle information describing a graphical object. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

ObjectGeography(*object*, *attribute*)

object is an Object expression.

attribute is an integer code specifying which type of information should be returned.

Return Value

Float

Description

The *attribute* parameter controls which type of information will be returned. The table below summarizes the different codes that you can use as the *attribute* parameter; codes in the left column (for example, OBJ_GEO_MINX) are defined in MAPBASIC.DEF.

Some attributes apply only to certain types of objects. For example, arc objects are the only objects with begin-angle or end-angle attributes, and text objects are the only objects with the text-angle attribute. If an object does not support z- or m-values, or a z- or m-value for this node is not defined, then an error is thrown.

attribute setting	ID	Return value (Float)
OBJ_GEO_MINX	1	Minimum x-coordinate of an object's minimum bounding rectangle (MBR), unless the object is a line; if the object is a line, returns same value as OBJ_GEO_LINEBEGX.
OBJ_GEO_MINY	2	Minimum y-coordinate of object's MBR. For lines, returns OBJ_GEO_LINEBEGY value.
OBJ_GEO_MAXX	3	Maximum x-coordinate of object's MBR. Does not apply to Point objects. For lines, returns OBJ_GEO_LINEENDX value.
OBJ_GEO_MAXY	4	Maximum y-coordinate of the object's MBR. Does not apply to Point objects. For lines, returns OBJ_GEO_LINEENDY value.
OBJ_GEO_ARCBEGANGLE	5	Beginning angle of an Arc object.
OBJ_GEO_ARCENDANGLE	6	Ending angle of an Arc object.
OBJ_GEO_LINEBEGX	1	X-coordinate of the starting node of a Line object.
OBJ_GEO_LINEBEGY	2	Y-coordinate of the starting node of a Line object.
OBJ_GEO_LINEENDX	3	X-coordinate of the ending node of a Line object.
OBJ_GEO_LINEENDY	4	Y-coordinate of the ending node of a Line object.
OBJ_GEO_POINTX	1	X-coordinate of a Point object.
OBJ_GEO_POINTY	2	Y-coordinate of a Point object.
OBJ_GEO_POINTZ	8	Z-value of a Point object.
OBJ_GEO_POINTM	9	M-value of a Point object.
OBJ_GEO_ROUNDRADIUS	5	Diameter of the circle that defines the rounded corner of a Rounded Rectangle object, expressed in terms of coordinate units (for example, degrees).
OBJ_GEO_CENTROID	5	Returns a point object for centroid of regions, collections, multipoints, and polylines. This is most commonly used with the **Alter Object statement**.
OBJ_GEO_TEXTLINEX	5	X-coordinate of the end of a Text object's label line.

attribute setting	ID	Return value (Float)
OBJ_GEO_TEXTLINEY	6	Y-coordinate of the end of a Text object's label line.
OBJ_GEO_TEXTANGLE	7	Rotation angle of a Text object.

The **ObjectGeography( )** function has been extended to support Multipoints and Collections. Both types support attributes 1 - 4 (coordinates of object's minimum bounding rectangle (MBR).

OBJ_GEO_MINX	1	Minimum x-coordinate of an object's MBR.
OBJ_GEO_MINY	2	Minimum y-coordinate of an object's MBR.
OBJ_GEO_MAXX	3	Maximum x-coordinate of an object's MBR.
OBJ GEO_MAXY	4	Maximum y-coordinate of an object's MBR.

Example

The following example reads the starting coordinates of a line object from the table City. A **Set Map statement** then uses these coordinates to re-center the Map window.

```
Include "MAPBASIC.DEF"
Dim i_obj_type As Integer, f_x, f_y As Float
Open Table "city"
Map From city
Fetch First From city
' at this point, the expression:
' city.obj
' represents the graphical object that's attached
' to the first record of the CITY table.
i_obj_type = ObjectInfo(city.obj, OBJ_INFO_TYPE)
If i_obj_type = OBJ_LINE Then
   f_x = ObjectGeography(city.obj, OBJ_GEO_LINEBEGX)
   f_y = ObjectGeography(city.obj, OBJ_GEO_LINEBEGY)
   Set Map Center (f_x, f_y)
End If
```

See Also:

Centroid( ) function, **CentroidX( ) function**, **CentroidY( ) function**, **ObjectInfo( ) function**

ObjectInfo( ) function

Purpose

Returns Pen, Brush, or other values describing a graphical object. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

ObjectInfo(*object*, *attribute*)

object is an Object expression.

attribute is an integer code specifying which type of information should be returned.

Return Value

SmallInt, integer, string, float, Pen, Brush, Symbol, or Font, depending on the attribute parameter

OBJ_INFO_NPOLYGONS (21) is an integer that indicates the number of polygons (in the case of a region) or sections (in the case of a polyline) which make up an object.

OBJ_INFO_NPOLYGONS+N (21) is an integer that indicates the number of nodes in the Nth polygon of a region or the Nth section of a polyline.

With region objects, MapInfo Professional counts the starting node twice (once as the start node and once as the end node). For example, **ObjectInfo( )** returns a value of 4 for a triangle-shaped region.

Description

The **ObjectInfo( )** function returns general information about one aspect of a graphical object. The first parameter should be an object value (for example, the name of an Object variable, or a table expression of the form *tablename*.obj).

Each object has several attributes. For example, each object has a “type” attribute, identifying whether the object is a point, a line, or a region, etc. Most types of objects have Pen and/or Brush attributes, which dictate the object's appearance. The **ObjectInfo( )** function returns one attribute of the specified object. Which attribute is returned depends on the value used in the attribute parameter. Thus, if you need to find out several pieces of information about an object, you will need to call **ObjectInfo( )** a number of times, with different attribute values in each call.

The table below summarizes the various attribute settings, and the corresponding return values.

attribute Setting	ID	Return Value
OBJ_INFO_TYPE	1	SmallInt, representing the object type; the return value is one of the values listed in the table below (for example, OBJ_TYPE_LINE). This attribute from the DEF file is 1 (ObjectInfo(Object, 1)).
OBJ_INFO_PEN	2	Pen style is returned; this query is only valid for the following object types: Arc, Ellipse, Line, Polyline, Frame, Regions, Rectangle, and Rounded Rectangle.
OBJ_INFO_BRUSH	3	Brush style is returned; this query is only valid for the following object types: Ellipse, Frame, Region, Rectangle, and Rounded Rectangle.

attribute Setting	ID	Return Value
OBJ_INFO_TEXTFONT	2	Font style is returned; this query is only valid for Text objects. ⓘ If the Text object is contained in a mappable table (as opposed to a Layout window), the Font specifies a point size of zero, and the text height is controlled by the Map window's zoom distance.
OBJ_INFO_SYMBOL	2	Symbol style; this query is only valid for Point objects.
OBJ_INFO_NPNTS	20	Integer, indicating the total number of nodes in a polyline or region object.
OBJ_INFO_SMOOTH	4	Logical, indicating whether the specified Polyline object is smoothed.
OBJ_INFO_FRAMEWIN	4	Integer, indicating the window ID of the window attached to a Frame object.
OBJ_INFO_FRAMETITLE	6	String, indicating a Frame object's title.
OBJ_INFO_NPOLYGONS	21	SmallInt, indicating the number of polygons (in the case of a region) or sections (in the case of a polyline) which make up an object.
OBJ_INFO_NPOLYGONS+*N*	21	Integer, indicating the number of nodes in the *N*th polygon of a region or the *N*th section of a polyline. ⓘ With region objects, MapInfo Professional counts the starting node twice (once as the start node and once as the end node). For example, **ObjectInfo( )** returns a value of 4 for a triangle-shaped region.
OBJ_INFO_TEXTSTRING	3	String, representing the body of a Text object; if the object has multiple lines of text, the string includes embedded line-feeds (`Chr$( 10 )` values).
OBJ_INFO_TEXTSPACING	4	Float value of 1, 1.5, or 2, representing a Text object's line spacing.
OBJ_INFO_TEXTJUSTIFY	5	SmallInt, representing justification of a Text object: 0 = left, 1 = center, 2 = right.
OBJ_INFO_TEXTARROW	6	SmallInt, representing the line style associated with a Text object: 0 = no line, 1 = simple line, 2 = arrow line.

attribute Setting	ID	Return Value
OBJ_INFO_FILLFRAME	7	Logical: TRUE if the object is a frame that contains a Map window, and the frame's “Fill Frame With Map” setting is checked.
OBJ_INFO_NONEMPTY	11	Logical, returns TRUE if a Multipoint object has nodes, or FALSE if the object is empty.
OBJ_INFO_REGION	8	Object value representing the region part of a collection object. If the collection object does not have a region, it returns an empty region. This query is valid only for collection objects.
OBJ_INFO_PLINE	9	Object value representing polyline part of a collection object. If the collection object does not have a polyline, it returns an empty polyline object. This query is valid only for collection objects.
OBJ_INFO_MPOINT	10	Object value representing the Multipoint part of a collection object. If the collection object does not have a Multipoint, it returns an empty Multipoint object. This query is valid only for collection objects.
OBJ_INFO_Z_UNIT_SET	12	Logical, indicating whether z units are defined.
OBJ_INFO_Z_UNIT	13	String result: indicates distance units used for z-values. Returns an empty string if units are not specified.
OBJ_INFO_HAS_Z	14	Logical, indicating whether the object has z-values.
OBJ_INFO_HAS_M	15	Logical, indicating whether the object has m-values.

The codes in the left column (for example, OBJ_INFO_TYPE) are defined through the MapBasic definitions file, MAPBASIC.DEF. Your program should Include “MAPBASIC.DEF” if you intend to call the **ObjectInfo( )** function.

Each graphic attribute only applies to some types of graphic objects. For example, point objects are the only objects with Symbol attributes, and text objects are the only objects with Font attributes. Therefore, the **ObjectInfo( )** function cannot return every type of attribute setting for every type of object.

If you specify OBJ_INFO_TYPE as the attribute setting, the **ObjectInfo( )** function returns one of the object types listed in the table below.

OBJ_INFO_TYPE values

OBJ_INFO_TYPE values	ID	Corresponding object type
OBJ_TYPE_ARC	1	Arc object
OBJ_TYPE_ELLIPSE	2	Ellipse / circle objects
OBJ_TYPE_LINE	3	Line object
OBJ_TYPE_PLINE	4	Polyline object
OBJ_TYPE_POINT	5	Point object
OBJ_TYPE_FRAME	6	Layout window Frame object
OBJ_TYPE_REGION	7	Region object
OBJ_TYPE_RECT	8	Rectangle object
OBJ_TYPE_ROUNDRECT	9	Rounded rectangle object
OBJ_TYPE_TEXT	10	Text object
OBJ_TYPE_MULTIPOINT	11	Collection point object
OBJ_TYPE_COLLECTION	12	Collection text object

Example

```
Include "MAPBASIC.DEF"
Dim counter, obj_type As Integer
Open Table "city"
Fetch First From city
   ' at this point, the expression: city.obj
   ' represents the graphical object that's attached
   ' to the first record of the CITY table.
obj_type = ObjectInfo(city.obj, OBJ_INFO_TYPE)
Do Case obj_type
   Case OBJ_TYPE_LINE
      Note "First object is a line."
   Case OBJ_TYPE_PLINE
      Note "First object is a polyline..."
      counter = ObjectInfo(city.obj, OBJ_INFO_NPNTS)
      Note " ... with " + Str$(counter) + " nodes."
   Case OBJ_TYPE_REGION
      Note "First object is a region..."
      counter = ObjectInfo(city.obj, OBJ_INFO_NPOLYGONS)
      Note ", made up of " + Str$(counter) + " polygons..."
      counter = ObjectInfo(city.obj, OBJ_INFO_NPOLYGONS+1)
```

```
     Note "The 1st polygon has" + Str$(counter) + " nodes"
End Case
```

See Also:

Alter Object statement, **Brush clause**, **Font clause**, **ObjectGeography( ) function**, **Pen clause**, **Symbol clause**

ObjectLen( ) function

Purpose

Returns the geographic length of a line or polyline object. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

ObjectLen(*expr*, *unit_name*)

expr is an object expression.

unit_name is a string representing the name of a distance unit (for example, “mi” for miles).

Return Value

Float

Description

The **ObjectLen( )** function returns the length of an object expression. Note that only line and polyline objects have length values greater than zero; to measure the circumference of a rectangle, ellipse, or region, use the **Perimeter( ) function**.

The **ObjectLen( )** function returns a length measurement in the units specified by the *unit_name* parameter; for example, to obtain a length in miles, specify “mi” as the *unit_name* parameter. See **Set Distance Units statement** for the list of valid unit names.

For the most part, MapInfo Professional performs a Cartesian or Spherical operation. Generally, a Spherical operation is performed unless the coordinate system is nonEarth, in which case, a Cartesian operation is performed.

Example

```
Dim geogr_length As Float
Open Table "streets"
Fetch First From streets
geogr_length = ObjectLen(streets.obj, "mi")
' geogr_length now represents the length of the
' street segment, in miles
```

See Also:

Distance( ) function, **Perimeter( ) function**, **Set Distance Units statement**

ObjectNodeHasM( ) function

Purpose

Returns TRUE if a specific node in a region, polyline or multipoint object has an m-value. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

ObjectNodeHasM(*object*, *polygon_num*, *node_num*)

object is an Object expression.

polygon_num is a positive integer value indicating which polygon or section to query. It is ignored for Multipoint objects (it used for regions and polylines).

node_num is a positive integer value indicating which node to read.

Return Value

Logical

Description

The **ObjectNodeHasM( )** function returns TRUE if the specific node from a region, polyline, or multipoint object has an m-value.

The *polygon_num* parameter must have a value of one or more. This specifies which polygon (if querying a region) or which section (if querying a polyline) should be queried. Call the **ObjectInfo( ) function** to determine the number of polygons or sections in an object. The **ObjectNodeHasM( )** function supports Multipoint objects and returns TRUE if a specific node in a Multipoint object has an m-value.

The *node_num* parameter must have a value of one or more; this tells MapBasic which of the object's nodes should be queried. You can use the **ObjectInfo( ) function** to determine the number of nodes in an object.

If the object does not support m-values or an m-value for this node is not defined, it returns FALSE.

Example

The following example queries the first graphic object in the table Routes. If the first object is a polyline, the program queries if the first node in the object has z-coordinates or m-values and queries z-coordinates and m-values of the first node in the polyline.

```
Dim i_obj_type As SmallInt,
   z, m As Float
   hasZ, hasM as Logical
Open Table "routes"
Fetch First From routes
```

```
   ' at this point, the expression:
   ' routes.obj
   ' represents the graphical object that's attached
   ' to the first record of the  routes  table.
i_obj_type = ObjectInfo(routes.obj, OBJ_INFO_TYPE)
If  i_obj_type = OBJ_PLINE Then
   ' ... then the object is a polyline...
   If (ObjectNodeHasZ(routes.obj, 1, 1)) Then
      z = ObjectNodeZ(routes.obj, 1, 1) ' read z-coordinate
   End If
   If (ObjectNodeHasM(routes.obj, 1, 1)) Then
      m = ObjectNodeM(routes.obj, 1, 1) ' read m-value
   End If
End If
```

See Also:

Querying Map Objects, **ObjectInfo( ) function**

ObjectNodeHasZ( ) function

Purpose

Returns TRUE if a specific node in a region, polyline, or multipoint object has a z-coordinate. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

ObjectNodeHasZ(*object*, *polygon_num*, *node_num*)

object is an Object expression.

polygon_num is a positive integer value indicating which polygon or section to query. It is ignored for Multipoint objects (it used for regions and polylines).

node_num is a positive integer value indicating which node to read.

Return Value

Logical

Description

The **ObjectNodeHasZ( )** function returns TRUE if a specific node from a region, polyline, or multipoint object has a z-coordinate. The *polygon_num* parameter must have a value of one or more. This specifies which polygon (if querying a region) or which section (if querying a polyline) should be queried. Call the **ObjectInfo( ) function** to determine the number of polygons or sections in an object. The **ObjectNodeHasZ( )** function supports Multipoint objects and returns TRUE if a specific node in a Multipoint object has a z-coordinate.

The *node_num* parameter must have a value of one or more; this tells MapBasic which of the object's nodes should be queried. You can use the **ObjectInfo( ) function** to determine the number of nodes in an object.

If *object* does not support z-coordinates or a z-coordinate for this node is not defined, it returns FALSE.

Example

The following example queries the first graphic object in the table Routes. If the first object is a polyline, the program queries if the first node in the object has z-coordinates or m-values and queries z-coordinates and m-values of the first node in the polyline.

```
Dim i_obj_type As SmallInt,
   z, m As Float
   hasZ, hasM as Logical
Open Table "routes"
Fetch First From routes
   ' at this point, the expression:
   ' routes.obj
   ' represents the graphical object that's attached
   ' to the first record of the routes table.
i_obj_type = ObjectInfo(routes.obj, OBJ_INFO_TYPE)
If  i_obj_type = OBJ_PLINE Then
   ' ... then the object is a polyline...
   If (ObjectNodeHasZ(routes.obj, 1, 1)) Then
      z = ObjectNodeZ(routes.obj, 1, 1) ' read z-coordinate
   End If
   If (ObjectNodeHasM(routes.obj, 1, 1)) Then
      m = ObjectNodeM(routes.obj, 1, 1) ' read m-value
   End If
End If
```

See Also:

Querying Map Objects, **ObjectInfo( ) function**

ObjectNodeM( ) function

Purpose

Returns the m-value of a specific node in a region, polyline, or multipoint object. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

ObjectNodeM(*object*, *polygon_num*, *node_num*)

object is an Object expression.

polygon_num is a positive integer value indicating which polygon or section to query. It is ignored for Multipoint objects (it used for regions and polylines).

node_num is a positive integer value indicating which node to read.

Return Value

Float

Description

The **ObjectNodeM( )** function returns the m-value of a specific node from a region, polyline, or multipoint object.

The *polygon_num* parameter must have a value of one or more. This specifies which polygon (if querying a region), or which section (if querying a polyline), should be queried. Call the **ObjectInfo( ) function** to determine the number of polygons or sections in an object. The **ObjectNodeM( )** function supports Multipoint objects and returns the m-value of a specific node in a Multipoint object.

The *node_num* parameter must have a value of one or more; this tells MapBasic which of the object's nodes should be queried. You can use the **ObjectInfo( ) function** to determine the number of nodes in an object.

If an object does not support m-values, or an m-value for this node is not defined, then an error is thrown.

Example

The following example queries the first graphic object in the table Routes. If the first object is a polyline, the program queries z-coordinates and m-values of the first node in the polyline.

```
Dim i_obj_type As SmallInt,
   z, m As Float
Open Table "routes"
Fetch First From routes
   ' at this point, the expression:
   ' routes.obj
   ' represents the graphical object that's attached
   ' to the first record of the routes table.
i_obj_type = ObjectInfo(routes.obj, OBJ_INFO_TYPE)
If i_obj_type = OBJ_PLINE Then
   ' ... then the object is a polyline...
   z = ObjectNodeZ(routes.obj, 1, 1) ' read z-coordinate
   m = ObjectNodeM(routes.obj, 1, 1) ' read m-value
End If
```

See Also:

Querying Map Objects, **ObjectInfo( ) function**

ObjectNodeX( ) function

Purpose

Returns the x-coordinate of a specific node in a region or polyline object. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

ObjectNodeX(*object*, *polygon_num*, *node_num*)

object is an Object expression.

polygon_num is a positive integer value indicating which polygon or section to query. It is ignored for Multipoint objects (it used for regions and polylines).

node_num is a positive integer value indicating which node to read.

Return Value

Float

Description

The **ObjectNodeX( )** function returns the x-value of a specific node from a region or polyline object. The corresponding **ObjectNodeY( ) function** returns the y-coordinate value.

The *polygon_num* parameter must have a value of one or more. This specifies which polygon (if querying a region) or which section (if querying a polyline) should be queried. Call the **ObjectInfo( ) function** to determine the number of polygons or sections in an object. The **ObjectNodeX( )** function supports Multipoint objects and returns the x-coordinate of a specific node in a Multipoint object.

The *node_num* parameter must have a value of one or more; this tells MapBasic which of the object's nodes should be queried. You can use the **ObjectInfo( ) function** to determine the number of nodes in an object. The **ObjectNodeX( )** function returns the value in the coordinate system currently in use by MapBasic; by default, MapBasic uses a Longitude/Latitude coordinate system. See **Set CoordSys statement** for more information about coordinate systems.

Example

The following example queries the first graphic object in the table Routes. If the first object is a polyline, the program queries the x- and y-coordinates of the first node in the polyline, then creates a new Point object at the location of the polyline's starting node.

```
Dim i_obj_type As SmallInt, x, y As Float, new_pnt As Object
Open Table "routes"
Fetch First From routes
' at this point, the expression:
' routes.obj
' represents the graphical object that's attached
' to the first record of the routes table.
```

```
i_obj_type = ObjectInfo(routes.obj, OBJ_INFO_TYPE)
If i_obj_type = OBJ_PLINE Then
' ... then the object is a polyline...
   x = ObjectNodeX(routes.obj, 1, 1) ' read longitude
   y = ObjectNodeY(routes.obj, 1, 1) ' read latitude
   Create Point Into Variable new_pnt (x, y)
   Insert Into routes (obj) Values (new_pnt)
End If
```

See Also:

Alter Object statement, **ObjectGeography( ) function**, **ObjectInfo( ) function**, **ObjectNodeY( ) function**, **Set CoordSys statement**

ObjectNodeY( ) function

Purpose

Returns the y-coordinate of a specific node in a region or polyline object. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

ObjectNodeY(*object*, *polygon_num*, *node_num*)

object is an Object expression.

polygon_num is a positive integer value indicating which polygon or section to query. It is ignored for Multipoint objects (it used for regions and polylines).

node_num is a positive integer value indicating which node to read.

Return Value

Float

Description

The **ObjectNodeY( )** function returns the y-value of a specific node from a region or polyline object. See **ObjectNodeX( ) function** for more information.

Example

See **ObjectNodeX( ) function**.

See Also:

Alter Object statement, **ObjectGeography( ) function**, **ObjectInfo( ) function**, **Set CoordSys statement**

ObjectNodeZ( ) function

Purpose

Returns the z-value of a specific node in a region, polyline, or multipoint object.

Syntax

ObjectNodeZ(*object*, *polygon_num*, *node_num*)

object is an Object expression.

polygon_num is a positive integer value indicating which polygon or section to query. It is ignored for Multipoint objects (it used for regions and polylines).

node_num is a positive integer value indicating which node to read

Return Value

Float

Description

The **ObjectNodeZ( )** function returns the z-value of a specific node from a region, polyline, or multipoint object.

The *polygon_num* parameter must have a value of one or more. This specifies which polygon (if querying a region) or which section (if querying a polyline) should be queried. Call the **ObjectInfo( ) function** to determine the number of polygons or sections in an object. The **ObjectNodeZ( )** function supports Multipoint objects and returns the z-coordinate of a specific node in a Multipoint object.

The *node_num* parameter must have a value of one or more; this tells MapBasic which of the object's nodes should be queried. You can use the **ObjectInfo( ) function** to determine the number of nodes in an object.

If object does not support Z-values, or Z-value for this node is not defined, then an error is thrown.

Example

The following example queries the first graphic object in the table Routes. If the first object is a polyline, the program queries z-coordinates and m-values of the first node in the polyline.

```
Dim i_obj_type As SmallInt,
   z, m As Float
Open Table "routes"
Fetch First From routes
   ' at this point, the expression:
   ' routes.obj
   ' represents the graphical object that's attached
   ' to the first record of the routes table.
i_obj_type = ObjectInfo(routes.obj, OBJ_INFO_TYPE)
If i_obj_type = OBJ_PLINE Then
```

```
   ' ... then the object is a polyline...
   z = ObjectNodeZ(routes.obj, 1, 1) ' read z-coordinate
   m = ObjectNodeM(routes.obj, 1, 1) ' read m-value
End If
```

See Also:

Querying Map Objects, **ObjectInfo( ) function**

Objects Check statement

Purpose

Checks a given table for various aspects of incorrect data, or possible incorrect data, which may cause problems and/or incorrect results in various operations. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Objects Check From tablename Into Table tablename
   [ SelfInt [ Symbol Clause] ]
   [ Overlap [ Pen Clause ] [ Brush Clause ] ]
   [ Gap Area [ Unit Units ] [ Pen Clause ] [ Brush Clause ] ] ]
```

tablename is a string representing the name of a table.

Clause is an expression.

Units is a value of an area.

Description

Objects Check will check the table designated in the **From** clause for various aspects of bad data which may cause problems or incorrect results with various operations. Only region objects will be checked. The region objects will be optionally checked for self-intersections, and areas of overlap and gaps.

Self-intersections may cause problems with various calculations, including the calculation for the area of a region. They may also cause incorrect results from various object-processing operations, such as combine, buffer, erase, erase outside, and split.

For any of these problems, a point object is created and placed into the output table. The output table can be supplied through the **Into Table** clause. If no **Into Table** clause exists, the output data is placed into the same table as the input table.

If the **SelfInt** option is included, then the table will be checked for self-intersections. Where found, point objects are created using the style provided by the **Symbol clause**. By default, this is a 28-point red pushpin.

Many region tables are designed to be boundary tables. The STATES.TAB and WORLD.TAB files provided with the sample data are examples of boundary tables. In tables such as these, boundaries should not overlap (for example, the state of Utah should not overlap with the state of Wyoming). The **Overlap** option will check the table for places where regions overlap with other regions.

Regions will be created in the output table representing any areas of overlap. These regions will be created using the **Brush clause** to represent the interior of the regions, and the **Pen clause** to represent the boundary of the regions. By default, these regions are drawn with solid yellow interiors and thin black boundaries.

Gaps are enclosed areas where no region object currently exists. In a boundary table, most regions abut other regions and share a common boundary. Just as there should be no overlaps between the regions, there should also be no gaps between the regions. In some cases, these boundary gaps are legitimate for the data. An example of this would be the Great Lakes in the World map, which separate parts of Canada from the USA. Most gaps that are data problems occur because adjacent boundaries do not have common boundaries that completely align. These gap areas are generally small.

To help weed out the legitimate gap areas, such as the Great Lakes, from problem gap areas, a **Gap Area** is used. Any potential gap that is larger than this gap area is discarded and not reported. The units that the **Gap Area** is in is presented by the **Units** clause. If the **Units** sub-clause is not present, then the **Gap Area** value will be interpreted in MapBasic's current area unit.

Gaps will be presented using the **Pen clause** and **Brush clause** that follow the **Gap** keyword. By default, these regions are drawn with blue interiors and a thin black boundary.

Example

This example will run **Objects Check** on the table called TestFile and store the results in the table called DumpFile. It will also use the **Overlap** keyword and change the default Point and Polygon styles.

```
objects check from TestFile into table Dumpfile Overlap
Selfint Symbol (67,16711680,28)
Overlap Pen (1,2,0) Brush (2,16776960,0)
Gap 100000 Units "sq mi" Pen (1,2,0) Brush (2,255,0)
```

See Also:

Objects Enclose statement

Objects Clean statement

Purpose

Cleans the objects from the given table, and optionally removes overlaps and gaps between regions. The table may be the Selection table. All objects to be cleaned must be closed object types (for example, regions, rectangles, rounded rectangles, or ellipses). You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Objects Clean From tablename
   [ Overlap ]
   [ Gap Area [ Unit Units ]]
```

tablename is a string representing the name of a table.

Units is a value of an area.

Description

The objects in the input *tablename* are first checked for various data problems and inconsistencies, such as self-intersections, overlaps, and gaps. Self-intersecting regions in the form of a figure 8 will be changed into a region containing two polygons that touch each other at a single point. Regions containing spikes will have the spike portion removed. The resulting cleaned object will replace the original input object.

If the **Overlap** keyword is included, then overlapping areas will be removed from regions. The portion of the overlap will be removed from all overlapping regions except the one with the largest area.

Objects Clean removes the overlap when one object is completely inside another. This is an exception to the rule of “biggest object wins”. If one object is completely inside another object, then the object that is inside remains, and a hole is punched in the containing object. The result does not contain any overlaps.

Gaps are enclosed areas where no region object currently exists. In a boundary table, most regions abut other regions and share a common boundary. Just as there should be no overlaps between the regions, there should also be no gaps between the regions. In some cases, both these boundary gaps and holes are legitimate for the data. An example of this would be the Great Lakes in the World map, which separate parts of Canada from the USA. Most gaps that are data problems occur because adjacent boundaries do not have common boundaries that completely align. These gap areas are generally small.

To help weed out the legitimate gap areas, such as the Great Lakes, from problem gap areas, a **Gap Area** is used. Any potential gap that is larger than this gap area is discarded and not reported. The units of the **Gap Area** are indicated by the **Units** sub-clause. If the **Units** sub-clause is not present, then the **Gap Area** value is interpreted in MapBasic's current area unit. Gaps that are found will be removed by combining the area defining the gap to the region with the largest area that touches the gap. To help determine a reasonable Gap Area, use the **Objects Check statement**. Any gaps that the **Objects Check statement** flags will be removed with the **Objects Clean** statement.

Example

```
Open Table "STATES.TAB" Interactive
Map From STATES
Set Map Layer 1 Editable On
select * from STATES
Objects Clean From Selection Overlap Gap 10 Units "sq m"
```

See Also:

Create Object statement, **Objects Disaggregate statement**, **Objects Check statement**

Objects Combine statement

Purpose

Combines objects in a table; corresponds to MapInfo Professional's **Objects > Combine** command. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Objects Combine
   [ Into Target ]
   [ Data column = expression [ , column = expression ... ] ]
```

column is a string representing the name of a column in the table being modified.

expression is an expression used to populate the *column*.

Description

Objects Combine creates an object representing the geographic union of the currently selected objects. Optionally, **Objects Combine** can also perform data aggregation, calculating sums or averages of the data values that are associated with the objects being combined.

The **Objects Combine** statement corresponds to MapInfo Professional's **Objects > Combine** menu item. For an introduction to this operation, see the discussion of the **Objects > Combine** menu item in the *MapInfo Professional User Guide*. To see a demonstration of the **Objects Combine** statement, run MapInfo Professional, open the MapBasic Window, and use the **Objects > Combine** command. Objects involved in the combine operation must either be all closed objects (for example, regions, rectangles, rounded rectangles, or ellipses) or all linear objects (for example, lines, polylines, or arcs). Mixed closed and linear objects as well as point and text objects are not allowed.

The optional **Into Target** clause is only valid if an editing target has been specified (either by the user or through the **Set Target statement**), and only if the target consists of one object. If you include the **Into Target** clause, MapInfo Professional combines the currently-selected objects with the current target object. The object produced by the combine operation then replaces the object that had been the editing target.

If you include the **Into Target** clause, and if the selected objects are from the same table as the target object, MapInfo Professional deletes the rows corresponding to the selected objects.

If you include the **Into Target** clause, and if the selected objects are from a different table than the target object, MapInfo Professional does not delete the selected objects. If you omit the **Into Target** clause, MapInfo Professional combines the currently-selected objects without involving the current editing target (if there is an editing target). The rows corresponding to the selected objects are deleted, and a new row is added to the table, containing the object produced by the combine operation.

The **Data** clause controls data aggregation. (For an introduction to data aggregation, see the description of the **Objects > Combine** operation in the *MapInfo Professional User Guide*.) The **Data** clause includes a comma-separated list of assignments. You can assign any expression to a column, assuming the expression is of the correct data type (numeric, string, etc.).

The following table lists the more common types of column assignments:

Expression	Description
col_name = *col_name*	The column contents are not altered.
col_name = *value*	MapBasic stores the hard-coded value in the column of the result object.
col_name = Sum(*col_name*)	Used only for numeric columns. The column in the result object contains the sum of the column values of all objects being combined.
col_name = Avg(*col_name*)	Used only for numeric columns. The column in the result object contains the average of column values of all objects in the group.
col_name = WtAvg(*colname*, *wtcolname*)	Used only for numeric columns. MapInfo Professional performs weighted averaging, averaging all of the *col_name* column values, and weighting the average calculation based on the contents of the *wt_colname* column.

The **Data** clause can contain an assignment for every column in the table. If the **Data** clause only includes assignments for some of the columns, MapBasic assigns blank values to those columns that are not listed in the **Data** clause. If you omit the **Data** clause entirely, but you include the **Into Target** clause, then MapInfo Professional retains the target object's original column values.

If you omit both the **Data** clause and the **Into Target** clause, then the object produced by the combine operation is stored in a new row, and MapInfo Professional assigns blank values to all of the columns of the new row.

See Also:

Combine( ) function, **Set Target statement**

Objects Disaggregate statement

Purpose

Breaks an object into its component parts. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Objects Disaggregate [ Into Table name ]
   [ All | Collection ]
   [ Data column_name = expression [ , column_name = expression ... ]
```

name is a string representing the name of a table to store the disaggregated objects.

column_name is a string representing the name of a column in the table being modified.

expression is an expression used to determine what is placed into the *column_name* columns.

Description

If an object contains multiple entities, then a new object is created in the output table for each entity.

By default, any multi-part object will be divided into its atomic parts. A Region object will be broken down into some number of region objects, depending on the **All** flag. If the **All** flag is present, then the Region will produce a series of single polygon Region objects, one object for each polygon contained in the original object. Holes (interior boundaries) will produce solid single polygon Region objects. If the **All** flag is not present, then Holes will be retained in the output objects. For example, if an input Region contains three polygons, and one of those polygons is a Hole in another polygon, then the output will be two Region objects, one of which will contain the hole.

Multiple-section Polyline objects will produce new single-section Polyline objects. Multipoint objects will produce new Point objects, one Point object per node from the input Multipoint.

Collections will be treated recursively. If a Collection contains a Region, then new Region objects will be produced as described above, depending on the **All** switch. If the Collection contains a Polyline object, the new Polyline objects will be produced for each section that exists in the input object. If a Collection contains a Multipoint, then new Point objects will be produced, one Point object for each node in the Mutlipoint. All other object types, including Points, Lines, Arcs, Rectangles, Rounded Rectangles, and Ellipses, which are already single component objects, will be moved to the output unchanged.

If a Region contains a single polygon, it will be passed unchanged to the output. If a Polyline object contains a single section, it will be passed unchanged to the output. If a Multipoint object contains a single node, the output object will be changed into a Point object containing that node. Arcs, Rectangles, Rounded Rectangles, and Ellipses will be passed unchanged to the output. Other object types, such as Text, will not be accepted by the **Objects Disaggregate** statement, and will produce an error.

The **Collection** keyword will only break up Collection objects. If a Collection object contains a Region, then that Region will be a new object on output. If a Collection object contains a Polyline, then that Polyline will be a new object in the output. If a Collection object contains a Multipoint, then that Multipoint will be a new object in the output. This differs from the above functionality since the output Region may contain multiple polygons, the output Polyline may contain multiple segments. The functionality above will never produce a Multipoint object.

With the **Collection** keyword, all other object types, including Points, Multipoints, Lines, Polylines, Arcs, Regions, Rectangles, Rounded Rectangles, and Ellipses, will be passed to the output unchanged.

If no **Into Table** is provided, the currently editable table is used as the output table. The input objects are taken from the current selection.

The optional **Data** clause controls what values are stored in the columns of the target objects. The **Data** clause can contain a comma-separated list of column assignments. Each column assignment can take one of the forms listed in the following table:

Assignment	Effect
col_name = *col_name*	Does not alter the value stored in the column.
col_name = *value*	Stores a specific value in the column. If the column is a character column, the value can be a string. If the column is a numeric column, the value can be a number.
col_name = Proportion(*col_name*)	Used only for numeric columns; reduces the number stored in the column in proportion to how much of the object's area was erased.

The **Data** clause can contain an assignment for every column in the table. If the **Data** clause only specifies assignments for some of the columns, blank values are assigned to those columns that are not listed in the **Data** clause. If you omit the **Data** clause entirely, all columns are blanked out of the target objects, storing zero values in numeric columns and blank values in character columns.

Example

```
Open Table "STATES.TAB" Interactive
Map From STATES
Set Map Layer 1 Editable On
select * from STATES
Objects Disaggregate Into Table STATES
```

See Also:

Create Object statement

Objects Enclose statement

Purpose

Creates regions that are formed from collections of polylines; corresponds to MapInfo Professional's **Objects > Enclose** menu item. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Objects Enclose
   [ Into Table tablename]
   [ Region ]
```

tablename is a string representing the name of the table you want to place objects in.

Description

Objects Enclose creates objects representing closures linear objects (lines, polylines, and arcs). A new region is created for each enclosed polygonal area. Input objects are obtained from the current selection. Unlike the **Objects Combine statement**, the **Objects Enclose** statement does not remove the original input objects. No data aggregation is done.

The optional **Region** clause allows closed objects (regions, rectangles, rounded rectangles, and ellipses) to be used as input to the **Objects Enclose** statement. The input regions will be converted to Polylines for the purpose of this operation. The effects are identical to first converting any closed objects to Polyline objects, and then performing the **Objects Enclose** operation. All input objects must be linear or closed, and any other objects (for example, points, multipoints, collections, and text) will cause the operation to produce an error. If closed objects exist in the selection, and the **Region** keyword is not present, then those objects will be ignored.

The **Objects Enclose** statement corresponds to MapInfo Professional's **Objects > Enclose** menu item. For an introduction to this operation, see the discussion of the **Objects > Enclose** menu item in the *MapInfo Professional User Guide*. To see a demonstration of the **Objects Enclose** statement, run MapInfo Professional, open the MapBasic Window, and use the **Objects > Combine** command.

The optional **Into Table** clause places the objects created by this command into the table. Otherwise, the output objects are placed in the same table that contains the input objects.

Example

This will select all the objects in a table called testfile, performs an **Objects Enclose** and stores the resulting objects in a table called dump_file.

```
select * from testfile
Objects Enclose Into Table dump_file
```

See Also:

Objects Check statement, **Objects Combine statement**

Objects Erase statement

Purpose

Erases any portions of the target object(s) that overlap the selection; corresponds to choosing **Objects > Erase**. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Objects Erase Into Target
   [ Data column_name = expression [ , column_name = expression ... ]
```

column_name is a string representing the name of a column in the table being modified.

expression is an expression used to determine what is erased from the *column_name* columns.

Description

The **Objects Erase** statement erases part of (or all of) the objects that are currently designated as the editing target. Using the **Objects Erase** statement is equivalent to choosing MapInfo Professional's **Objects > Erase** menu item. For an introduction to using **Objects > Erase**, see the *MapInfo Professional User Guide*.

Objects Erase erases any parts of the target objects that overlap the currently selected objects. To erase only the parts of the target objects that do not overlap the selection, use the **Objects Intersect statement**.

Before you call **Objects Erase**, one or more closed objects (regions, rectangles, rounded rectangles, or ellipses) must be selected, and an editing target must exist. The editing target may have been set by the user choosing **Objects > Set Target**, or it may have been set by the MapBasic **Set Target statement**.

For each Target object, one object will be produced for that portion of the target that lies outside all cutter objects. If the Target lies inside cutter objects, then no object is produced for output.

The optional **Data** clause controls what values are stored in the columns of the target objects. The **Data** clause can contain a comma-separated list of column assignments.

Each column assignment can take one of the forms listed in the following table:

Assignment	Effect
col_name = *col_name*	MapBasic does not alter the value stored in the column.
col_name = *value*	MapBasic stores a specific value in the column. If it is a character column, the value can be a string; if it is a numeric column, the value can be a number.
col_name = Proportion(*col_name*)	Used only for numeric columns; MapBasic reduces the number stored in the column in proportion to how much of the object's area was erased. So, if the operation erases half of an area's object, the object's column value is reduced by half.

The **Data** clause can contain an assignment for every column in the table. If the **Data** clause only specifies assignments for some of the columns, MapBasic assigns blank values to those columns that are not listed in the **Data** clause.

If you omit the **Data** clause entirely, MapBasic blanks out all columns of the target object, storing zero values in numeric columns and blank values in character columns.

Example

In the following example, the **Objects Erase** statement does not include a **Data** clause. As a result, MapBasic stores blank values in the columns of the target object(s). This example assumes that one or more target objects have been designated, and one or more objects have been selected.

```
Objects Erase Into Target
```

In the next example, the **Objects Erase** statement includes a **Data** clause, which specifies expressions for three columns (State_Name, Pop_1990, and Med_Inc_80). This operation assigns the string “area remaining” to the State_Name column and specifies that the Pop_1990 column should be reduced in proportion to the amount of the object that is erased. The Med_Inc_80 column retains the value it had before the **Objects Erase** statement. The target objects' other columns are blanked out.

```
Objects Erase Into Target
   Data
      State_Name = "area remaining",
      Pop_1990 = Proportion( Pop_1990 ),
      Med_Inc_80 = Med_Inc_80
```

See Also:

Erase( ) function, **Objects Intersect statement**

Objects Intersect statement

Purpose

Erases any portions of the target object(s) that do not overlap the selection; corresponds to choosing **Objects > Erase Outside**. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Objects Intersect Into Target
   [**Data** *column_name* = *expression* [, *column_name* = *expression* ...]]

column_name is a string representing the name of a column in the table being modified.

expression is an expression used to determine what is erased from the *column_name* columns.

Description

The **Objects Intersect** statement erases part or all of the object(s) currently designated as the editing target. Using the **Objects Intersect** statement is equivalent to choosing MapInfo Professional's **Objects > Erase Outside** menu item. For an introduction to using **Objects > Erase Outside**, see the *MapInfo Professional User Guide*.

The optional **Data** clause controls what values are stored in the columns of the target objects. The **Data** clause can contain a comma-separated list of column assignments. Each column assignment can take one of the forms listed in the following table:

Assignment	Effect
col_name = *col_name*	MapBasic does not alter the value stored in the column.
col_name = *value*	MapBasic stores a specific value in the column. If the column is a character column, the value can be a string; if the column is a numeric column, the value can be a number.
col_name = Proportion(*col_name*)	Used only for numeric columns; MapBasic reduces the number stored in the column in proportion to how much of the object's area was erased. Thus, if the operation erases half of the area of an object, the object's column value is reduced by half.

The **Data** clause can contain an assignment for every column in the table. If the **Data** clause only specifies assignments for some of the columns, MapBasic assigns blank values to those columns that are not listed in the **Data** clause. If you omit the **Data** clause entirely, MapBasic blanks out all columns of the target objects, storing zero values in numeric columns and blank values in character columns.

The **Objects Intersect** statement is very similar to the **Objects Erase statement**, with one important difference: **Objects Intersect** erases the parts of the target objects(s) that do not overlap the current selection, while the **Objects Erase statement** erases the parts of the target object. For each Target object, a new object is created for each area that intersects a cutter object. For example, if a target object is intersected by three cutter objects, then three new objects will be created. The parts of the target that lie outside all cutter objects will be discarded. For more information, see **Objects Erase statement**.

Example

```
Objects Intersect Into Target
   Data
      Field2=Proportion(Field2)
```

See Also:

Create Object statement, **Overlap( ) function**, **Objects Erase statement**

Objects Move statement

Purpose

Moves the objects obtained from the current selection within the input table. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Objects Move
   Angle angle
   Distance distance
   [ Units unit ]
   [ Type { Spherical | Cartesian }]
```

angle is a value representing the angle to move the selected object.

distance is a number representing the distance to move the selected object.

unit is the distance unit of *distance*.

Description

Objects Move moves the objects within the input table. The source objects are obtained from the current selection. The resulting objects replace the input objects. No data aggregation is performed or necessary, since the data associated with the original source objects is unchanged.

The object is moved in the direction represented by *angle*, measured from the positive X-axis (east) with positive angles being counterclockwise, and offset at a distance given by the *distance* parameter. The *distance* is in the units specified by *unit* parameter, if present. If the **Units** clause is not present, then the current distance unit is the default. By default, MapBasic uses miles as the distance unit; to change this unit, use the **Set Distance Units statement**.

The optional **Type** sub-clause lets you specify the type of distance calculation used to create the offset. If **Spherical** type is specified, then the calculation is done by mapping the data into a Latitude/Longitude On Earth projection and using *distance* measured using Spherical distance calculations. If Cartesian is specified, then the calculation is done by considering the data to be projected to a flat surface and distances are measured using Cartesian distance calculations. If the **Type** sub-clause is not present, then the Spherical distance calculation type is used. If the data is in a Latitude/Longitude Projection, then Spherical calculations are used regardless of the **Type** setting. If the data is in a NonEarth Projection, the Cartesian calculations are used regardless of the **Type** setting.

There are some considerations for Spherical measurements that do not hold for Cartesian measurements. If you move an object that is in Lat/Long, the shape of the object remains the same, but the area of the object will change. This is because you are picking one offset delta in degrees, and the actual measured distance for a degree is different at different locations.

For the Offset functions, the actual offset delta is calculated at some fixed point on the object (for example, the center of the bounding box), and then that value is converted from the input units into the coordinate system's units. If the coordinate system is Lat/Long, the conversion to degrees uses

the fixed point. The actual converted distance measurement could vary at different locations on the object. The distance from the input object and the new offset object is only guaranteed to be exact at the single fixed point used.

Example

```
Objects Move Angle 45 Distance 100 Units "mi" Type Spherical
```

See Also:

Objects Offset statement

Objects Offset statement

Purpose

Copies objects, obtained from the current selection, offset from the original objects. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Objects Offset
   [ Into Table intotable ]
   Angle angle
   Distance distance
   [ Units unit ]
   [ Type { Spherical | Cartesian }]
   [ Data column = expression [, column = expression ... ]]
```

intotable is a string representing the table that the new values are copied to.

angle is a value representing the angle which to offset the selected objects.

distance is a number representing the distance to offset the selected objects.

unit is the distance unit of *distance*.

column is a string representing the column on which to perform the offset.

expression is an expression to calculate the offset for the column.

Description

Objects Offset makes a new copy of objects offset from the original source objects. The source objects are obtained from the current selection. The resulting objects are placed in the *intotable*, if the **Into** clause is present. Otherwise, the objects are placed into the same table as the input objects are obtained from (for example, the base table of the selection).

The object is moved in the direction represented by *angle*, measured from the positive X-axis (east) with positive angles being counterclockwise, and offset at a distance given by the *distance* parameter. The *distance* is in the units specified by the *unit* parameter. If the **Units** clause is not present, then the current distance unit is the default. By default, MapBasic uses miles as the distance unit; to change this unit, use the **Set Distance Units statement**.

The optional **Type** sub-clause lets you specify the type of distance calculation used to create the offset. If **Spherical** type is specified, then the calculation is done by mapping the data into a Latitude/Longitude On Earth projection and using distance measured using Spherical distance calculations. If Cartesian is specified, then the calculation is done by considering the data to be projected to a flat surface and distances are measured using Cartesian distance calculations. If the **Type** sub-clause is not present, then the Spherical distance calculation type is used. If the data is in a Latitude/Longitude Projection, then Spherical calculations are used regardless of the **Type** setting. If the data is in a NonEarth Projection, the Cartesian calculations are used regardless of the **Type** setting.

If you specify a **Data** clause, the application performs data aggregation.

There are some considerations for Spherical measurements that do not hold for Cartesian measurements. If you move an object that is in Lat/Long, the shape of the object remains the same, but the area of the object will change. This is because you are picking one offset delta in degrees, and the actual measured distance for a degree is different at different locations.

For the Offset functions, the actual offset delta is calculated at some fixed point on the object (for example, the center of the bounding box), and then that value is converted from the input units into the coordinate system's units. If the coordinate system is Lat/Long, the conversion to degrees uses the fixed point. The actual converted distance measurement could vary at different locations on the object. The distance from the input object and the new offset object is only guaranteed to be exact at the single fixed point used.

Example

```
Objects Offset Into Table c:\temp\table1.tbl Angle 45 Distance 100 Units
"mi" Type Spherical
```

See Also:

Offset( ) function

Objects Overlay statement

Purpose

Adds nodes to the target objects at any places where the target objects intersect the currently selected objects; corresponds to choosing **Objects > Overlay Nodes**. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Objects Overlay Into Target
```

Description

Before you call **Objects Overlay**, one or more objects must be selected, and an editing target must exist. The editing target may have been set by the user choosing **Objects > Set Target**, or it may have been set by the MapBasic **Set Target statement**. For more information, see the discussion of Overlay Nodes in the *MapInfo Professional Reference*.

See Also:

OverlayNodes( ) function, **Set Target statement**

Objects Pline statement

Purpose

Splits a single section polyline into two polylines. You issue call this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Objects Pline Split At Node index
   [ Into Table name ]
   [ Data column_name = expression [ , column_name = expression ... ]]
```

index is an integer of the index number of the node to split.

name is a string representing the name of the table to hold the new objects.

column_name is a string representing the name of the column where the new values are stored.

expression is an expression which is used to assign values to *column_name*.

Description

If an object is a single section polyline, then two new single section polyline objects are created in the output table *name*. The **Node** *index* should be a valid MapBasic index for the polyline to be split. If **Node** is a start or end node for the polyline, the operation is cancelled and an error message is displayed.

The optional **Data** clause controls what values are stored in the columns of the output objects. The **Data** clause can contain a comma-delimited list of column assignments. Each column assignment can take one of the forms listed in the following table:

Assignment	Effect
col_name = col_name	Does not alter the value stored in the column.
col_name = value	Stores a specific value in the column. If the column is a character column the value can be a string; if the column is a numeric column, the value can be a number.

The **Data** clause can contain an assignment for every column in the table. If the **Data** clause specifies assignments for only some of the columns, blank values are assigned to those columns that are not listed in the **Data** clause.

If you omit the **Data** clause entirely, all columns are blanked out of the target objects, storing zero values in numeric columns and blank values in character columns.

Example

In the following partial example, the selected polyline is split at the specified node (node index of 12). The unchanged values from each record of the selected polyline are inserted into the new records for the split polyline.

```
Objects Pline Split At Node 12 Into Table WORLD Data
Country=Country,Capital=Capital,Continent=Continent,Numeric_code=Numeric_
code,FIPS=FIPS,ISO_2=ISO_2,ISO_3=ISO_3,Pop_1994=Pop_1994,Pop_Grw_Rt=Pop_G
rw_Rt,Pop_Male=Pop_Male,Pop_Fem=Pop_Fem...
```

See Also:

ObjectLen( ) function, **ObjectNodeX( ) function**, **ObjectNodeY( ) function**, **Objects Disaggregate statement**

Objects Snap statement

Purpose

Cleans the objects from the given table, and optionally performs various topology-related operations on the objects, including snapping nodes from different objects that are close to each other into the same location and generalization/thinning. The table may be the Selection table. All of the objects to be cleaned must either be all linear (for example, polylines and arcs) or all closed (for example, regions, rectangles, rounded rectangles, or ellipses). Mixed linear and closed objects cannot be cleaned in one operation, and an error will result. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Objects Snap From tablename
   [ Tolerance [ Node node_distance ][ Vector vector_distance ]
      [ Units unit_string] ]
   [ Thin [ Bend bend_distance ][ Distance spacing_distance ]
      [ Units unit_string ]]
   [ Cull Area cull_area [ Units unit_string ]]]
```

tablename is a string representing the name of the table of the objects to be checked.

node_distance is a number representing a radius around the end points nodes of a polyline.

vector_distance is a number representing a radius used for internal nodes of polylines.

bend_distance is a number representing the co-linear tolerance of a series of nodes.

spacing_distance is a number representing the minimum distance a series of nodes in the same object can be to each other without being removed.

unit_string is a string representing the distance units to be used.

cull_area is a number representing the threshold area within which polygons are culled.

unit_string is a string representing the area units to be used.

Description

The objects from the input *tablename* are checked for various data problems and inconsistencies, such as self-intersections. Self-intersecting regions in the form of a figure 8 will be changed into a region containing two polygons that touch each other at a single point. Regions containing spikes have the spike portion removed. The resulting cleaned object replaces the original input object. If any overlaps exist between the objects they are removed. Removal of overlaps generally consists of cutting the overlapping portion out of one of the objects, while leaving it in the other object. The region that contains the originally overlapping section consists of multiple polygons. One polygon represents the non-overlapping portion, and a separate polygon represents each overlapping section.

The **Node** and **Vector Tolerances** values snap nodes from different objects together, and can be used to eliminate small overlaps and gaps between objects. The **Units** sub-clause of **Tolerances** lets you specify a distance measurement name (such as “km” for kilometers) to apply to the **Node** and **Vector** values. If the **Units** sub-clause is not present, then the **Node** and **Vector** values are interpreted in MapBasic's current distance unit. By default, MapBasic uses miles as the distance units; to change this unit, use the **Set Distance Units statement**.

The **Node** tolerance is a radius around the end point nodes of a polyline. If there are nodes from other objects within this radius, then one or both of the nodes will be moved such that they will be in the same location (for example, they will be snapped together).

The **Vector** tolerance is a radius used for internal nodes of polylines. Its purpose is the same as the **Node** tolerance, except it is used only for internal (non-end point) nodes of a polyline. Note that for Region objects, there is no explicit concept of end point nodes, since the nodes form a closed loop. For Region objects, only the **Vector** tolerance is used, and it is applied to all nodes in the object. The **Node** tolerance is ignored for Region objects. For Polyline objects, the **Node** tolerance must be greater than or equal to the **Vector** tolerance.

The **Bend** and **Distance** values can be used to help thin or generalize the input objects. This reduces the number of nodes used in the object while maintaining the general shape of the object. The **Units** sub-clause of **Thin** lets you specify a distance measurement name (such as “km” for kilometers) to apply to the **Bend** and **Distance** values. If the **Units** sub-clause is not present, then the **Bend** and **Distance** values are interpreted in MapBasic's current distance unit.

The **Bend** tolerance is used to control how co-linear a series of nodes can be. Given three nodes, connect all of the nodes in a triangle. Measure the perpendicular distance from the second node to the line connecting the first and third nodes. If this distance is less than the **Bend** tolerance, then the three nodes are considered co-linear, and the second node is removed from the object.

The **Distance** tolerance is used to eliminate nodes within the same object that are close to each other. Measure the distance between two successive nodes in an object. If the distance between them is less than the **Distance** tolerance, then one of the nodes can be removed.

The **Cull Area** value is used to eliminate polygons from regions that are smaller than the threshold area. The **Units** sub-clause of **Cull** lets you specify an area measurement name (such as “sq km” for square kilometers) to apply to the **Area** value. If the **Units** sub-clause is not present, then the **Area** value is interpreted in MapBasic's current area unit. By default, MapBasic uses square miles as the area unit; to change this unit, use the **Set Area Units statement**.

For all of the distance and area values mentioned above, the type of measurement used is always Cartesian. Please keep in mind the coordinate system that your data is in. A length and area calculation in Longitude/Latitude calculated using the Cartesian method is not mathematically precise. Ensure that you are working in a suitable coordinate system (a Cartesian system) before applying the tolerance values.

Example

```
Open Table "STATES.TAB" Interactive
Map From STATES
Set Map Layer 1 Editable On
select * from STATES
Objects Snap From Selection Tolerance Node 3 Vector 3 Units "mi" Thin Bend
0.5 Distance 1 Units "mi" Cull Area 10 Units "sq mi"
```

See Also:

Create Object statement, **Overlap( ) function**

Objects Split statement

Purpose

Splits target objects, using the currently-selected objects as a “cookie cutter”; corresponds to choosing **Objects > Split**. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Objects Split Into Target
[**Data** *column_name = expression* [, *column_name = expression* ...]

column_name is a string representing the name of the column where the new values are stored.

expression is an expression which is used to assign values to *column_name*.

Description

Use the **Objects Split** statement to split each of the target objects into multiple objects. Using **Objects Split** is equivalent to choosing MapInfo Professional's **Objects > Split** menu item. For more information on split operations, see the *MapInfo Professional Reference*.

Before you call **Objects Split**, one or more closed objects (regions, rectangles, rounded rectangles, or ellipses) must be selected, and an editing target must exist. The editing target may have been set by the user choosing **Objects > Set Target**, or it may have been set by the MapBasic **Set Target statement**.

For each target object, a new object is created for each area that intersects a cutter object. For example, if a target object is intersected by three cutter objects, then three new objects will be created. In addition, a single object will be created for all parts of the target object that lie outside all cutter objects. This is equivalent to performing both an **Objects Erase statement** and an **Objects Intersect statement** (**Objects > Erase Outside**).

The optional **Data** clause controls what values are stored in the columns of the target objects. The **Data** clause can contain a comma-separated list of column assignments. Each column assignment can take one of the forms listed in the following table:

Assignment	Effect
col_name = *col_name*	MapBasic does not alter the value stored in the column; each object resulting from the split operation retains the original column value.
col_name = *value*	MapBasic stores a specific value in the column. If the column is a character column, the value can be a string; if the column is a numeric column, the value can be a number. Each object resulting from the split operation retains the specified value.
col_name = Proportion(*col_name*)	Used only for numeric columns; MapInfo Professional divides the original target object's column value among the graphical objects resulting from the split. Each object receives “part of” the original column value, with larger objects receiving larger portions of the numeric values.

The **Data** clause can contain an assignment for every column in the table. If the **Data** clause only specifies assignments for some of the columns, MapBasic assigns blank values to those columns that are not listed in the **Data** clause.

If you omit the **Data** clause entirely, MapBasic blanks out all columns of the target objects, storing zero values in numeric columns and blank values in character columns.

Example

In the following example, the **Objects Split** statement does not include a **Data** clause. As a result, MapBasic stores blank values in the columns of the target object(s).

```
Objects Split Into Target
```

In the next example, the statement includes a **Data** clause, which specifies expressions for three columns (State_Name, Pop_1990, and Med_Inc_80). This first part of the **Data** clause assigns the string “sub-division” to the State_Name column; as a result, “sub-division” will be stored in the State_Name column of each object produced by the split. The next part of the **Data** clause specifies that the target object's original Pop_1990 value should be divided among the objects produced by the split. The third part of the **Data** clause specifies that each of the new objects should retain the original value from the Med_Inc_80 column.

```
Objects Split Into Target
   Data
      State_Name = "sub-division",
      Pop_1990 = Proportion( Pop_1990 ),
      Med_Inc_80 = Med_Inc_80
```

See Also:

Alter Object statement

Offset( ) function

Purpose

Returns a copy of the input object offset by the specified distance and angle. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Offset(*object*, *angle*, *distance*, *units*)

object is the object being offset.

angle is the angle to offset the object.

distance is a number representing the distance to offset the object.

units is a string representing the unit in which to measure *distance*.

Return Value

Object

Description

Offset( ) produces a new object that is a copy of the input object offset by *distance* along *angle* (in degrees with horizontal in the positive X-axis being 0 and positive being counterclockwise). The *units* string, similar to that used for the **ObjectLen( ) function** or **Perimeter( ) function**, is the unit for the *distance* value. The distance type used is Spherical unless the Coordinate System is NonEarth. For NonEarth, Cartesian distance type is automatically used. The coordinate system used is the coordinate system of the input object.

There are some considerations for Spherical measurements that do not hold for Cartesian measurements. If you move an object that is in Lat/Long, the shape of the object remains the same, but the area of the object will change. This is because you are picking one offset delta in degrees, and the actual measured distance for a degree is different at different locations.

For the Offset functions, the actual offset delta is calculated at some fixed point on the object (for example, the center of the bounding box), and then that value is converted from the input units into the coordinate system's units. If the coordinate system is Lat/Long, the conversion to degrees uses the fixed point. The actual converted distance measurement could vary at different locations on the object. The distance from the input object and the new offset object is only guaranteed to be exact at the single fixed point used.

Example

```
Offset(Rect, 45, 100, "mi")
```

See Also:

Objects Offset statement, **OffsetXY( ) function**

OffsetXY( ) function

Purpose

Returns a copy of the input object offset by the specified X and Y offset values. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

OffsetXY(*object*, *xoffset*, *yoffset*, *units*)

object is the object being offset.

xoffset and *yoffset* are numbers representing the distance along the x and y axes to offset the object.

units is a string representing the unit in which to measure distance.

Return Value

Object

Description

OffsetXY( ) produces a new object that is a copy of the input object offset by *xoffset* along the X-axis and *yoffset* along the Y-axis. The *units* string, similar to that used for the **ObjectLen( ) function** or **Perimeter( ) function**, is the unit for the distance values. The distance type used is Spherical unless the coordinate system is NonEarth. For NonEarth, the Cartesian distance type is automatically used. The coordinate system used is the coordinate system of the input object.

There are some considerations for Spherical measurements that do not hold for Cartesian measurements. If you move an object that is in Lat/Long, the shape of the object remains the same, but the area of the object will change. This is because you are picking one offset delta in degrees, and the actual measured distance for a degree is different at different locations.

For the Offset functions, the actual offset delta is calculated at some fixed point on the object (for example, the center of the bounding box), and then that value is converted from the input units into the coordinate system's units. If the coordinate system is Lat/Long, the conversion to degrees uses the fixed point. The actual converted distance measurement could vary at different locations on the object. The distance from the input object and the new offset object is only guaranteed to be exact at the single fixed point used.

Example

```
OffsetXY(Rect, 92, -22, "mi")
```

See Also:

Offset( ) function

OnError statement

Purpose

Enables an error-handling routine.

Syntax

OnError Goto{ *label* | 0 }

label is a string representing a label within the same procedure or function.

Restrictions

You cannot issue an **OnError** statement through the MapBasic window.

Description

The **OnError** statement either enables an error-handling routine, or disables a previously enabled error-handler. (An error-handler is a group of statements executed in the event of an error).

BASIC programmers should note that in the MapBasic syntax, **OnError** is a single word.

An **OnError Goto** *label* statement enables an error-handling routine. Following such an **OnError** statement, if the application generates an error, MapBasic jumps to the label line specified. The statements following the *label* presumably correct the error condition, warn the user about the error condition, or both. Within the error-handling routine, use a **Resume statement** to resume program execution.

Once you have inserted error-handling statements in your program, you may need to place a flow-control statement (for example, **Exit Sub statement** or **End Program statement**) immediately before the error handler's label. This prevents the program from unintentionally "falling through" to the error handling statements, but it does not prevent MapBasic from calling the error handler in the event of an error. See the example below.

An **OnError Goto 0** statement disables the current error-handling routine. If an error occurs while there is no error-handling routine, MapBasic displays an error dialog box, then halts the application.

Each error handler is local to a particular function or procedure. Thus, a sub procedure can define an error handler by issuing a statement such as:

```
OnError Goto recover
```

(assuming that the same procedure contains a label called "recover"). If, after executing the above **OnError** statement, the procedure issues a **Call statement** to call another sub procedure, the "recover" error handler is suspended until the program returns from the Call statement. This is because each label (for example, "recover") is local to a specific procedure or function. With this arrangement, each function and each sub procedure can have its own error handling.

If an error occurs within an error-handling routine, your MapBasic program halts.

Example

```
OnError GoTo no_states
Open Table "states"

OnError GoTo no_cities
Open Table "cities"

Map From cities, states

after_mapfrom:
   OnError GoTo 0
   '
   ' ...
   '
End Program

no_states:
   Note "Could not open table States... no Map used."
   Resume after_mapfrom

no_cities:
   Note "City data not available..."
   Map From states
   Resume after_mapfrom
```

See Also:

Err( ) function, **Error statement**, **Error$( ) function**, **Resume statement**

Open Connection statement

Purpose

Creates a connection to an external geocode or Isogram service provided by a MapMarker or Envinsa server. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Open Connection Service
   Geocode [ MapMarker | Envinsa ] | Isogram URL URLstring
   [ User name_string [ Password pwd_string ] ]
   [ Interactive [ On | Off ] ]
    into variable var_name
```

URLString is a string representing a valid URL. *URLString* must be a valid URL to a routing service if you are specifying **Isogram**, or to a geocoding service if you are specifying **Geocode**.

name_string is a string representing the user name for an Envinsa or MapMarker installation.

pwd_string is a string representing the password corresponding to *user_name*.

var_name is a integer representing the variable which will hold the returned connection number.

Description

The **Open Connection** statement creates a connection to a Geocode or Isogram service. Each statement must specify a service and provider to which the connection is being established. Since the Isogram service is only provided by Envinsa no provider can be specified. If the service is Geocode and no service provider is specified, **Envinsa** is assumed.

The **Into variable** keywords are required as *var_name* is the variable that holds the returned connection number that is then passed to other statements, such as the **Set Connection Geocode statement**, the **Geocode statement**, the **Set CoordSys statement**, and the **Create Object Isogram statement**.

Interactive determines whether a username/password dialog box is shown if and only if the credentials passed in for authentication are not adequate. With **Interactive** specified to **Off**, no dialog box is displayed and the command fails if the authentication fails. With **Interactive** specified as **On**, the dialog box appears if the authentication fails.

The default for the command is **Interactive Off**. That is, if the **Interactive** keyword is not used at all, it is the same as **Interactive Off**. However, if **Interactive** is specified, it is equivalent to **Interactive On**.

Examples

All examples without the keyword **MapMarker** assume Envinsa.

The following example opens a geocoding connection without **Interactive** specified. **Interactive** is set to **Off** by default.

```
Open Connection Into Variable CnctNum Service Geocode URL
"http://EnvinsaServices/LocationUtility/services/LocationUtility"
```

This example opens a geocode connection and specifies **Interactive** as **On**.

```
Open Connection Into Variable CnctNum Service Geocode URL
"http://EnvinsaServices/LocationUtility/services/LocationUtility"
Interactive On
```

This example opens a geocode connection with a server that requires authentication.

```
dim baseURLVariable as String
baseURLVariable = "http://EnvinsaServices/”
Open Connection Service Geocode URL baseURLVariable +
"LocationUtility/services/LocationUtility" User "geocodeuser" Password
"GeoMe" Into Variable CnctNum
```

This example opens an Isogram connection with a server that requires authentication.

```
dim baseURLVariable as String
baseURLVariable = "http://EnvinsaServices/”
Open Connection Service IsoGram URL baseURLVariable +
"Route/services/Route" User "isogramuser" Password "ISOMe" Into Variable
CnctNum
```

See Also:

Close Connection statement, **Set Connection Geocode statement**, **Set Connection Isogram statement**

Open File statement

Purpose

Opens a file for input/output.

Syntax

```
Open File filespec
   [ For { Input | Output | Append | Random | Binary } ]
   [ Access { Read | Write | Read Write } ]
   As [ # ] filenum
   [ Len = recordlength ]
   [ ByteOrder { LOWHIGH | HIGHLOW } ]
   [ CharSet char_set ]
```

filespec is a string representing the name of the file to be opened.

filenum is an integer number to associate with the open file; this number is used in subsequent operations (for example, **Get statement** or **Put statement**).

recordlength identifies the number of characters per record, including any end-of-line markers used; applies only to Random access.

char_set is the name of a character set; see **CharSet clause**.

Restrictions

You cannot issue an **Open File** statement through the MapBasic window.

Description

The **Open File** statement opens a file, so that MapBasic can read information from and/or write information to the file.

In MapBasic, there is an important distinction between files and tables. MapBasic provides one set of statements for using tables (for example, **Open Table statement**, **Fetch statement**, and **Select statement**) and another set of statements for using other files in general (for example, **Open File**, **Get statement**, **Put statement**, **Input # statement**, **Print # statement**).

The **For** clause specifies what type of file i/o to perform: Sequential, Random, or Binary. Each type of i/o is described below. If you omit the **For** clause, the file is opened in **Random** mode.

Sequential File I/O

If you are going to read a text file that is variable-length (for example, one line is 55 characters long, and the next is 72 characters long, etc.), you should specify a Sequential mode: **Input**, **Output**, or **Append**.

If you specify the **For Input** clause, you can read from the file by issuing an **Input # statement** and a **Line Input # statement**.

If you specify the **For Output** clause or the **For Append** clause, you can write to the file by issuing a **Print # statement** and a **Write # statement**.

If you specify **For Input**, the **Access** clause may only specify **Read**; conversely, if you specify **For Output**, the **Access** clause may only specify **Write**.

Do not specify a **Len** clause for files opened in any of the Sequential modes.

Random File I/O

If the text file you are going to read is fixed-length (for example, every line is 80 characters long), you can access the file in **Random** mode, by specifying the clause: **For Random**.

When you open a file in **Random** mode, you must provide a **Len =** *recordlength* clause to specify the record length. The *recordlength* value should include any end-of-line designator, such as a carriage-return line-feed sequence.

When using **Random** mode, you can use the **Access** clause to specify whether you intend to **Read** from the file, **Write** to the file, or do both (**Read Write**). After opening a file in **Random** mode, use the **Get statement** and the **Put statement** to read from, and write to, the file.

Binary File I/O

In **Binary** access, MapBasic converts MapBasic variables to binary values when writing, and converts from binary values when reading. Storing numerical data in a Binary file is more compact than storing Binary data in a text file; however, Binary files cannot be displayed or printed directly, as can text files.

To open a file in Binary mode, specify the clause: **For Binary**.

When using **Binary** mode, you can use the **Access** clause to specify whether you intend to **Read** from the file, **Write** to the file, or do both (**Read Write**). After opening a file in Binary mode, use the **Get statement** and the **Put statement** to read from, and write to, the file.

Do not specify a **Len** clause or a **CharSet clause** for files opened in Binary mode.

Controlling How the File Is Interpreted

The **CharSet** clause specifies a character set. The *char_set* parameter should be a string constant, such as “WindowsLatin1”. If you omit the CharSet clause, MapInfo Professional uses the default character set for the hardware platform that is in use at run-time. Note that the CharSet clause only applies to files opened in **Input**, **Output**, or **Random** modes. See **CharSet clause** for more information.

If you open a file for **Random** or **Binary** access, the **ByteOrder** clause specifies how numbers are stored within the file.

If your application only runs on one hardware platform, you do not need to be concerned with byte order; MapBasic simply uses the byte-order scheme that is “native” to that platform. However, if you intend to read and write binary files, and you need to transport the files across multiple hardware platforms, you may need to use the **ByteOrder** clause.

Examples

```
Open File "cxdata.txt" For INPUT As #1
Open File "cydata.txt" For RANDOM As #2 Len=42
Open File "czdata.bin" For BINARY As #3
```

See Also:

Close File statement, **EOF( ) function**, **Get statement**, **Input # statement**, **Open Table statement**, **Print # statement**, **Put statement**, **Write # statement**, **CharSet clause**

Open Report statement

Purpose

Loads a report into the Crystal Report Designer module. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Open Report *reportfilespec*

reportfilespec is a string representing a full path and file name for an existing report file.

See Also:

Create Report From Table statement

Open Table statement

Purpose

Opens a MapInfo Professional table for input/output. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Open Table filename [ As tablename ]
   [ Hide ] [ ReadOnly ] [ Interactive ] [ Password pwd ]
   [ NoIndex ] [ View Automatic ] [ DenyWrite ]
   [ VMGrid | VMRaster | VMDefault ]
```

filename is a string which specifies which MapInfo table to open.

tablename is a string representing an “alias” name by which the table should be identified.

pwd is a string representing the database-level password for the database, to be specified when database security is turned on. Applies to Access tables only.

VMGrid treats all VM GRD files as Grid Layers when opened.

VMRaster treats all VM GRD files as Raster Layers when opened.

VMDefault treats GRD as Raster or Grid depending on existence of RasterStyle 6 1 tag in TAB file.

Description

The **Open Table** statement opens an existing table. The effect is comparable to the effect of an end-user choosing **File > Open** and selecting a table to open. A table must be opened before MapInfo Professional can process that table in any way.

The name of the file to be opened (specified by the *filespec* parameter) must correspond to a table which already exists; to create a new table from scratch, use the **Create Table statement**. The **Open Table** statement only applies to MapInfo tables; to use files that are in other formats, use the **Register Table statement** and the **Open File statement**.

If the statement includes an **As** clause, MapInfo Professional opens the table under the “alias” table name indicated by the *tablename* parameter, rather than by the actual table name. This affects the way the table name appears in lists, such as the list that appears when a user chooses **File > Close**. Furthermore, when an **Open Table** statement specifies an alias table name, subsequent MapBasic table operations (for example, a **Close Table statement**) must refer to the alias table name, rather than the permanent table name. An alias table name remains in effect until the table is closed. Opening a table under an alias does not have the effect of permanently renaming the table.

If the statement includes the **Hide** clause, the table will not appear in any dialog boxes that display lists of open tables (for example, the **File > Close** dialog box). Use the **Hide** clause if you need to open a table that should remain hidden to the user. If the statement includes the **ReadOnly** clause, the user is not allowed to edit the table.

The optional **Interactive** keyword tells MapBasic to prompt the user to locate the table if it is not found at the specified path. The **Interactive** keyword is useful in situations where you do not know the location of the user's files. If the statement includes the **NoIndex** keyword, the MapInfo index will not be re-built for an MS Access table when opened.

View Automatic is an optional clause to the **Open Table** statement that allows the MapInfo table, workspace or application file associated with a hotlink object to launch in the currently running instance of MapInfo Professional or start a new instance if none is running. If **View Automatic** is present, after opening the table, MapInfo Professional will either add it to an existing mapper, open a new mapper, or open a browser. This is especially useful with the HotLinks feature.

DenyWrite is an optional clause for MS Access tables only, If it is specified, other users will not be able to edit the table. If another user already has read-write access to the table, the **Open Table** command will fail.

Attempting to open two tables that have the same name

MapInfo Professional can open two separate tables that have the same name. In such cases, MapInfo Professional needs to open the second table under a special name, to avoid conflicts. Depending on whether the **Open Table** statement includes the **Interactive** keyword, MapBasic either assigns the special table name automatically, or displays a dialog box to let the user select a special table name.

For example, a user might keep two copies of a table called “Sites”, one copy in a directory called 2006 (for example, “C:\2006\SITES.TAB”) and another, perhaps newer copy of the table in a different directory (for example, “C:\2005\SITES.TAB”). When the user (or an application) opens the first Sites table, MapInfo Professional opens the table under its default name (“Sites”). If an application issues an **Open Table** statement to open the second Sites table, MapInfo Professional automatically opens the second table under a modified name (for example, “Sites_2”) to distinguish it from the first table. Alternately, if the **Open Table** statement includes the **Interactive** clause, MapInfo Professional displays a dialog box to let the user select the alternate name.

Regardless of whether the **Open Table** statement specifies the **Interactive** keyword, the result is that a table may be opened under a non-default name. Following an **Open Table** statement, issue the function call `TableInfo(0, TAB_INFO_NAME)` to determine the name with which MapInfo Professional opened the table.

Attempting to open a table that is already open

If a table is already open, and an **Open Table As** statement tries to re-open the same table under a new name, MapBasic generates an error code. A single table may not be open under two different names simultaneously.

However, if a table is already open, and then an **Open Table** statement tries to re-open that table without specifying a new name, MapBasic does not generate an error code. The table simply remains open under its current name.

Example

The following example opens the table STATES.TAB, then displays the table in a Map window. Because the **Open Table** statement uses an **As** clause to open the table under an alias (USA), the Map window's title bar will say “USA Map” rather than “States Map.”

```
Open Table "States" As USA
Map From USA
```

The next example follows an **Open Table** statement with a **TableInfo( ) function** call. In the unlikely event that a separate table by the same name (States) is already open when you run the program below, MapBasic will open “C:STATES.TAB” under a special alias (for example, “STATES_2”). The **TableInfo( ) function** call returns the alias under which the “C:STATES.TAB” table was opened.

```
Include "MAPBASIC.DEF"
Dim s_tab As String
Open Table "C:states"
s_tab = TableInfo(0, TAB_INFO_NAME)
Browse * From s_tab
Map From s_tab
```

See Also:

Close Table statement, **Create Table statement**, **Delete statement**, **Fetch statement**, **Insert statement**, **TableInfo( ) function**, **Update statement**

Open Window statement

Purpose

Opens or displays a window. You can issue this statement in the MapBasic Window in MapInfo Professional.

Syntax

Open Window *window_name*

window_name is a string representing a window name (for example, Ruler) or window code (for example, WIN_RULER).

Description

The **Open Window** statement displays an MapInfo Professional window. For example, the following statement displays the statistics window, as if the user had chosen **Options > Show Statistics Window**.

```
Open Window Statistics
```

The *window_name* parameter should be one of the window names from the table below.

Window Name	Window Description
MapBasic	The MapBasic window. You also can refer to this window by its define code from MAPBASIC.DEF (WIN_MAPBASIC).
Statistics	The Statistics window (WIN_STATISTICS).
Legend	The Theme Legend window (WIN_LEGEND).
Info	The Info tool window (WIN_INFO).
Ruler	The Ruler tool window (WIN_RULER).
Help	The Help window (WIN_HELP).
Message	The Message window used by the **Print statement** (WIN_MESSAGE).
TableList	A dialog box that displays a list of currently opened tables. ⓘ This dialog box is not strictly a window so does not have an entry in MAPBASIC.DEF

You cannot open a document window (Map, Graph, Browse, Layout) through the **Open Window** statement. There is a separate statement for opening each type of document window (see the **Map statement**, **Graph statement**, **Browse statement**, **Layout statement**, and **Create Redistricter statement**).

See Also:

Close Window statement, **Print statement**, **Set Window statement**

Overlap( ) function

Purpose

Returns an object representing the geographic intersection of two objects; produces results similar to MapInfo Professional's **Objects > Erase Outside** command. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Overlap(*object1*, *object2*)

object1 is an object; it cannot be a point or text object.

object2 is an object; it cannot be a point or text object.

Return Value

An object that is the geographic intersection of *object1* and *object2*.

Description

The **Overlap( )** function calculates the geographic intersection of two objects (the area covered by both objects), and returns an object representing that intersection.

MapBasic retains all styles (color, etc.) of the original *object1* parameter; then, if necessary, MapBasic applies the current drawing styles.

If one of the objects is linear (for example, a polyline) and the other object is closed (for example, a region), **Overlap( )** returns the portion of the linear object that is covered by the closed object.

See Also:

AreaOverlap( ) function, **Erase( ) function**, **Objects Intersect statement**

OverlayNodes( ) function

Purpose

Returns an object based on an existing object, with new nodes added at points where the object intersects a second object. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

OverlayNodes(*input_object*, *overlay_object*)

input_object is an object whose nodes will be included in the output object; it may not be a point or text object.

overlay_object is an object that will be intersected with input_object; it may not be a point or text object.

Return Value

A region object or a polyline object.

Description

The **OverlayNodes( )** function returns an object that contains all the nodes in *input_object* plus nodes at all locations where the *input_object* intersects with the *overlay_object*.

If the *input_object* is a closed object (region, rectangle, rounded rectangle, or ellipse), **OverlayNodes( )** returns a region object. If *input_object* is a linear object (line, polyline, or arc), **OverlayNodes( )** returns a polyline.

The object returned retains all styles (color, etc.) of the original *input_object*.

To determine whether the *OverlayNodes( )* function added any nodes to the input_object, use the **ObjectInfo( ) function** to count the number of nodes (OBJ_INFO_NPNTS). Even if two objects do intersect, the **OverlayNodes( )** function does not add any nodes if *input_object* already has nodes at the points of intersection.

See Also:

Objects Overlay statement

Pack Table statement

Purpose

Provides the functionality of MapInfo Professional's **Table > Maintenance > Pack Table** command. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Pack Table table { Graphic | Data | Graphic Data } [ Interactive ]
```

table is a string representing the name of an open table that does not have unsaved changes.

Description

To pack a table's data, include the optional **Data** keyword. When you pack a table's data, MapInfo Professional physically deletes any rows that had been flagged as “deleted.”

To pack a table's graphical objects, include the optional **Graphic** keyword. Packing the graphical objects removes empty space from the map file, resulting in a smaller table. However, packing a table's graphical objects may cause editing operations to be slower.

The **Pack Table** statement can include both the **Graphic** keyword and the **Data** keyword, and it must include at least one of the keywords.

A **Pack Table** statement may cause map layers to be removed from a Map window, possibly causing the loss of themes or cosmetic objects.

If you include the **Interactive** keyword, MapInfo Professional prompts the user to save themes and/or cosmetic objects (if themes or cosmetic objects are about to be lost). This statement cannot pack linked tables. Also, this statement cannot pack a table that has unsaved edits. To save edits, use the **Commit Table statement**.

Packing a table can invalidate custom labels that are stored in workspaces. Suppose you create custom labels and save them in a workspace. If you delete rows from your table and pack the table, you may get incorrect labels the next time you load the workspace. (Within a workspace, custom labels are stored with respect to row ID numbers; when you pack a table, you change the table's row ID numbers, possibly invalidating custom labels stored in workspaces.) If you only delete rows from the end of the table (for example, from the bottom of the Browser window), packing will not invalidate the custom labels.

Packing Access Tables

The **Pack Table** statement saves a copy of the original Microsoft Access table without the column types that MapInfo Professional does not support. If a Microsoft Access table has MEMO, OLE, or LONG BINARY type columns, those columns are lost during a pack.

Example

```
Pack Table parcels Data
```

See Also:

Open Table statement

PathToDirectory$( ) function

Purpose

Returns only the specified file's directory. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

PathToDirectory$(*filespec*)

filespec is a string expression representing a full file specification.

Return Value

String

Description

The **PathToDirectory$( )** function returns just the “directory” component from a full file specification.

A full file specification can include a directory and a filename. The file specification `C:\MAPINFO\DATA\WORLD.TAB` includes the directory “C:\MAPINFO\DATA\”.

Example

```
Dim s_filespec, s_filedir As String
s_filespec = "C:\MAPINFO\DATA\STATES.TAB"
s_filedir = PathToDirectory$(s_filespec)

' s_filedir now contains the string "C:\MAPINFO\DATA\"
```

See Also:

PathToFileName$( ) function, **PathToTableName$( ) function**

PathToFileName$( ) function

Purpose

Returns just the file name from a specified file. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

PathToFileName$(*filespec*)

filespec is a string expression representing a full file specification.

Return Value

String

Description

The **PathToFileName$( )** function returns just the “filename” component from a full file specification.

A full file specification can include a directory and a filename. The **PathToFileName$( )** function returns the file's name, including the file extension if there is one.

The file specification `C:\MAPINFO\DATA\WORLD.TAB` includes a directory (“C:\MAPINFO\DATA\”) and a filename (“WORLD.TAB”).

Example

```
Dim s_filespec, s_filename As String
s_filespec = "C:\MAPINFO\DATA\STATES.TAB"
s_filename = PathToFileName$(s_filespec)

' filename now contains the string "STATES.TAB"
```

See Also:

PathToDirectory$( ) function, **PathToTableName$( ) function**

PathToTableName$( ) function

Purpose

Returns a string representing a table alias (such as “_1995_Data”) from a complete file specification (such as “C:\MapInfo\Data\1995 Data.tab”). You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

PathToTableName$(*filespec*)

filespec is a string expression representing a full file specification.

Return Value

String, up to 31 characters long.

Description

Given a full file name that identifies a table's .TAB file, this function returns a string that represents the table's alias. The alias is the name by which a table appears in the MapInfo Professional user interface (for example, on the title bar of a Browser window).

To convert a file name to a table alias, MapInfo Professional removes the directory path from the beginning of the string and removes ".TAB" from the end of the string. Any special characters (for example, spaces or punctuation marks) are replaced with the underscore character (_). If the table name starts with a number, MapInfo Professional inserts an underscore at the beginning of the alias. If the resulting string is longer than 31 characters, MapInfo Professional trims characters from the end; aliases cannot be longer than 31 characters.

Note that a table may sometimes be open under an alias that differs from its default alias. For example, the following **Open Table statement** uses the optional **As** clause to force the World table to use the alias "Earth":

```
Open Table "C:\MapInfo\Data\World.tab" As Earth
```

Furthermore, if the user opens two tables that have identical names but different directory locations, MapInfo Professional assigns the second table a different alias, so that both tables can be open at once. In either of these situations, the "default alias" returned by **PathToTableName$( )** might not match the alias under which the table is currently open. To determine the alias under which a table was actually opened, call the **TableInfo( ) function** with the TAB_INFO_NAME code.

Example

```
Dim s_filespec, s_tablename As String
s_filespec = "C:\MAPINFO\DATA\STATES.TAB"
s_tablename = PathToTableName$(s_filespec)
' s_tablename now contains the string "STATES"
```

See Also:

PathToDirectory$( ) function, **PathToFileName$( ) function**, **TableInfo( ) function**

Pen clause

Purpose

Specifies a line style for graphic objects. You can use this clause in the MapBasic Window in MapInfo Professional.

Syntax

Pen *pen_expr*

pen_expr is a Pen expression, for example, **MakePen**(*width*, *pattern*, *color*)

Description

The **Pen** clause specifies a line style—in other words, a set of thickness, pattern, and color settings that dictate the appearance of a line or polyline object.

The **Pen** clause is not a complete MapBasic statement. Various object-related statements, such as the **Create Line statement**, let you include a **Pen** clause to specify an object's line style. The keyword **Pen** may be followed by an expression which evaluates to a **Pen** value. This expression can be a **Pen** variable:

```
Pen pen_var
```

or a call to a function (for example, the **CurrentPen( ) function** or the **MakePen( ) function**) which returns a Pen value:

```
Pen MakePen(1, 2, BLUE)
```

You can create an interleaved line style by adding 128 to the pattern value. The following example draws a two (2) pixel cyan colored line using pattern 101 in an interleaved style (101+128=229):

```
Pen MakePen(2, 229, CYAN)
```

With some MapBasic statements (for example, the **Set Map statement**), the keyword **Pen** can be followed immediately by the three parameters that define a Pen style (width, pattern, and color) within parentheses:

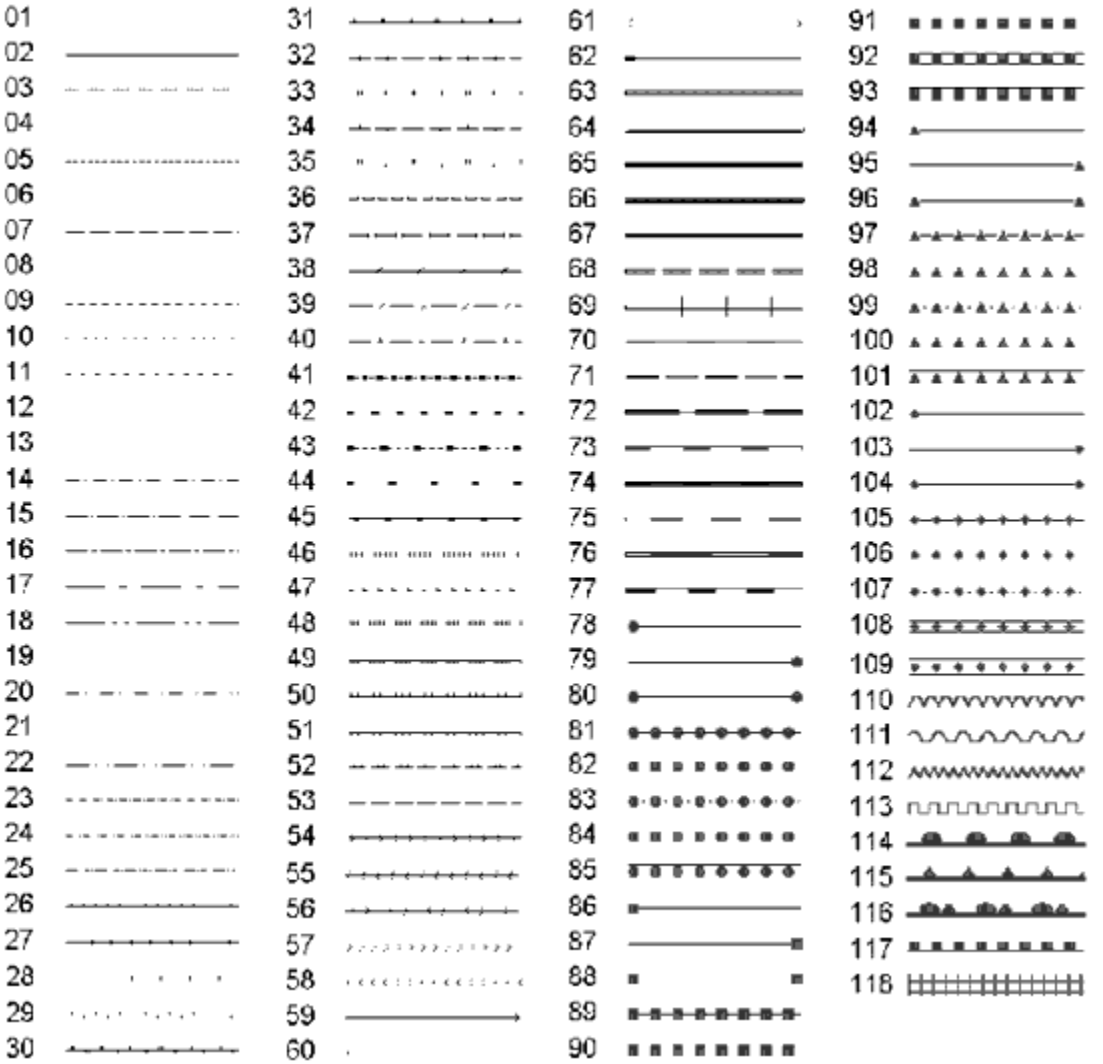
```
Pen(1, 2, BLUE)
```

Some MapBasic statements take a **Pen** expression as a parameter (for example, the name of a Pen variable), rather than a full **Pen** clause (the keyword **Pen** followed by the name of a Pen variable). The **Alter Object statement** is one example.

The following table summarizes the components that define a Pen:

Component	Description
width	Integer value, usually from 1 to 7, representing the thickness of the line (in pixels). To create an invisible line style, specify a width of zero, and use a pattern value of 1 (one). To specify a width using points, calculate the pen width from a point size using the **PointsToPenWidth( ) function**. This calculation multiplies the point size by 10 and then adds 10 to the result, so the pen width is always larger than 10.
pattern	Integer value from 1 to 118; see table below. Pattern 1 is invisible. To specify an interleaved line style, add 128 to the pattern. However, not all patterns benefit from an interleaved line style.
color	Integer RGB color value; see **RGB( ) function**.

The available pen patterns appear in the figure below.



Examples

```
Include "MAPBASIC.DEF"
Dim cable As Object
Create Line
   Into Variable cable
   (73.5, 42.6) (73.67, 42.9)
   Pen MakePen(1, 2, BLACK)
```

Apply line styles to a layer in a map as a layer style override: Pen width = 5 points; Line style B17 in line style picker; penpattern = 66.

```
Set Map Window <windowid>
   Layer 1 Display Global
```

```
   Global Line (60,194,16711680)
   ' Interleave: 194 = 66 + 128

Set Map Window 234499920
   Layer 1 Display Global
   Global Line MakePen(PointsToPenWidth(5),66,16711680)
   ' 5 point line, non-interleaved
```

See Also:

Alter Object statement, **CreateLine( ) function**, **Create Pline statement**, **CurrentPen( ) function**, **IsPenWidthPixels( ) function**, **MakePen( ) function**, **PointsToPenWidth( ) function**, **PenWidthToPoints( ) function**, **RGB( ) function**, **Set Style statement**

PenWidthToPoints( ) function

Purpose

Returns the point size for a given pen width. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

PenWidthToPoints(*penwidth*)

penwidth is an integer greater than 10 representing the pen width.

Return Value

Float

Description

The **PenWidthToPoints( )** function takes a pen width and returns the point size for that pen. The pen width for a line style may be returned by the **StyleAttr( ) function**. The pen width returned by the **StyleAttr( ) function** may be in points or pixels. Pen widths of less than ten are in pixels. Any pen width of ten or greater is in points. **PenWidthToPoints( )** only returns values for pen widths that are in points. To determine if pen widths are in pixels or points, use the **IsPenWidthPixels( ) function**.

Example

```
Include "MAPBASIC.DEF"
Dim CurPen As Pen
Dim Width As Integer
Dim PointSize As Float
CurPen = CurrentPen( )
Width = StyleAttr(CurPen, PEN_WIDTH)
If Not IsPenWidthPixels(Width) Then
   PointSize = PenWidthToPoints(Width)
End If
```

See Also:

CurrentPen( ) function, **IsPenWidthPixels( ) function**, **MakePen( ) function**, **Pen clause**, **PointsToPenWidth( ) function**, **StyleAttr( ) function**

Perimeter( ) function

Purpose

Returns the perimeter of a graphical object. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Perimeter(*obj_expr*, *unit_name*)

obj_expr is an object expression.

unit_name is a string representing the name of a distance unit (for example, “km”).

Return Value

Float

Description

The **Perimeter( )** function calculates the perimeter of the *obj_expr* object. The **Perimeter( )** function is defined for the following object types: ellipses, rectangles, rounded rectangles, and polygons. Other types of objects have perimeter measurements of zero.

The **Perimeter( )** function returns a length measurement in the units specified by the *unit_name* parameter; for example, to obtain a length in miles, specify “mi” as the *unit_name* parameter. See **Set Distance Units statement** for the list of valid unit names.

The **Perimeter( )** function returns approximate results when used on rounded rectangles. MapBasic calculates the perimeter of a rounded rectangle as if the object were a conventional rectangle. For the most part, MapInfo Professional performs a Cartesian or Spherical operation. Generally, a spherical operation is performed unless the coordinate system is nonEarth, in which case, a Cartesian operation is performed.

Example

The following example shows how you can use the **Perimeter( )** function to determine the perimeter of a particular geographic object.

```
Dim perim As Float
Open Table "world"
Fetch First From world
perim = Perimeter(world.obj, "km")
' The variable perim now contains
' the perimeter of the polygon that's attached to
' the first record in the World table.
```

You can also use the **Perimeter( )** function within the **Select statement**. The following **Select statement** extracts information from the States table, and stores the results in a temporary table called Results.

Because the **Select statement** includes the **Perimeter( )** function, the Results table will include a column showing each state's perimeter.

```
Open Table "states"
Select state, Perimeter(obj, "mi")
   From states
   Into results
```

See Also:

Area( ) function, **ObjectLen( ) function**, **Set Distance Units statement**

PointsToPenWidth( ) function

Purpose

Returns a pen width for a given point size. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

PointsToPenWidth(*pointsize*)

pointsize is a float value in tenths of a point.

Return Value

SmallInt

Description

The **PointsToPenWidth( )** function takes a value in tenths of a point and converts that into a pen width.

Example

```
Include "MAPBASIC.DEF"
Dim Width As Integer
Dim p_bus_route As Pen
Width = PointsToPenWidth(1.7)
p_bus_route = MakePen(Width, 9, RED)
```

See Also:

CurrentPen( ) function, **IsPenWidthPixels( ) function**, **MakePen( ) function**, **Pen clause**, **PenWidthToPoints( ) function**, **StyleAttr( ) function**

PointToMGRS$( ) function

Purpose

Converts an object value representing a point into a string representing an MGRS (Military Grid Reference System) coordinate. Only point objects are supported. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

PointToMGRS$(*inputobject*)

inputobject is an object expression representing a point.

Description

MapInfo Professional automatically converts the input point from the current MapBasic coordinate system to a Long/Lat (WGS84) datum before performing the conversion to an MGRS string. However, by default, the MapBasic coordinate system is Long/Lat (no datum); using this as an intermediate coordinate system can cause a significant loss of precision in the final output, since datumless conversions are much less accurate. As a rule, the MapBasic coordinate system should be set to either Long/Lat (WGS84) or to the coordinate system of the source data table, so that no unnecessary intermediate conversions are performed. See Example 2 below.

Return Value

String

Examples

The following examples illustrate the use of both the MGRSToPoint( ) and PointToMGRS$( ) functions.

Example 1:

```
dim obj1 as Object
dim s_mgrs As String
dim obj2 as Object

obj1 = CreatePoint(-74.669, 43.263)
s_mgrs = PointToMGRS$(obj1)
obj2 = MGRSToPoint(s_mgrs)
```

Example 2:

```
Open Table "C:\Temp\MyTable.TAB" as MGRSfile

' When using the PointToMGRS$( ) or MGRSToPoint( ) functions,
' it is very important to make sure that the current MapBasic
' coordsys matches the coordsys of the table where the
' point object is being stored.

'Set the MapBasic coordsys to that of the table used
```

```
Set CoordSys Table MGRSfile

'Update a Character column (e.g. COL2) with MGRS strings from
'a table of points

Update MGRSfile
   Set Col2 = PointToMGRS$(obj)

'Update two float columns (Col3 & Col4) with
'CentroidX & CentroidY information
'from a character column (Col2) that contains MGRS strings.

Update MGRSfile
   Set Col3 = CentroidX(MGRSToPoint(Col2))

Update mgrstestfile ' MGRSfile
   Set Col4 = CentroidY(MGRSToPoint(Col2))

 Table MGRSfile
Close Table MGRSfile
```

See Also:

MGRSToPoint( ) function

PointToUSNG$(obj, datumid)

Purpose

Converts an object value representing a point into a string representing an USNG (United States National Grid) coordinate. Only point objects are supported. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

PointToUSNG$(*obj, datumid*)

obj is an object expression representing the point to be converted. It must evaluate to a point object.

datumid is a numeric expression representing the datum id. It must evaluate to one of the following values.

```
DATUMID_NAD27  (62)
DATUMID_NAD83  (74)
DATUMID_WGS84 (104)
```

DATUMID_* are defines in MapBasic.def. WGS84 and NAD83 are treated as equivalent.

Description

MapInfo Professional automatically converts the input point from the current MapBasic coordinate system to a Long/Lat (WGS84 and NAD27) datum before performing the conversion to an USNG string. However, by default, the MapBasic coordinate system is Long/Lat (no datum); using this as an intermediate coordinate system can cause a significant loss of precision in the final output, since datumless conversions are much less accurate. As a rule, the MapBasic coordinate system should be set to either Long/Lat (WGS84 and NAD27) or to the coordinate system of the source data table, so that no unnecessary intermediate conversions are performed.

Return Value

String

Example 1

The following example illustrates the use of USNGToPoint( ) and PointToUSNG$( ) functions.

```
dim obj1 as Object
dim s_USNG As String
dim obj2 as Object

obj1 = CreatePoint(-74.669, 43.263)
s_USNG = PointToUSNG$(obj1)
obj2 = USNGToPoint(s_USNG)
```

Example 2

```
Open Table "C:\Temp\MyTable.TAB" as USNGfile

' When using the PointToUSNG$( ) or USNGToPoint( ) functions,
' it is very important to make sure that the current MapBasic
' coordsys matches the coordsys of the table where the
' point object is being stored.

'Set the MapBasic coordsys to that of the table used
Set CoordSys Table USNGfile

'Update a Character column (e.g. COL2) with USNG strings from
'a table of points

Update USNGfile
   Set Col2 = PointToUSNG$(obj)

'Update two float columns (Col3 & Col4) with
'CentroidX & CentroidY information
'from a character column (Col2) that contains USNG strings.

Update USNGfile
   Set Col3 = CentroidX(USNGToPoint(Col2))

Update USNGtestfile ' USNGfile
   Set Col4 = CentroidY(USNGToPoint(Col2))
```

```
 Table USNGfile
Close Table USNGfile
```

See Also:

USNGToPoint(string)

Print statement

Purpose

Prints a prompt or a status message in the Message window. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Print *message*

message is a string expression.

Description

The **Print** statement prints a message to the Message window. The Message window is a special window which does not appear in MapInfo's standard user interface. The Message window lets you display custom messages that relate to a MapBasic program. You could use the Message window to display status messages (“Record deleted”) or prompts for the user (“Select the territory to analyze.”). To set the font for the Message window, use the **Set Window statement**. A MapBasic program can explicitly open the Message window through the **Open Window statement**.

If a **Print** statement occurs while the Message window is closed, MapBasic opens the Message window automatically. The **Print** statement is similar to the **Note statement**, in that you can use either statement to display status messages or debugging messages. However, the **Note statement** displays a dialog box, pausing program execution until the user clicks **OK**. The **Print** statement simply prints text to a window, without pausing the program. Each **Print** statement is printed to a new line in the Message window. After you have printed enough messages to fill the Message window, scroll buttons appear at the right edge of the window, to allow the user to scroll through the messages.

To clear the Message window, print a string which includes the form-feed character (code 12):

```
Print Chr$(12) 'This statement clears the Message window
```

By embedding the line-feed character (code 10) in a message, you can force a single message to be split onto two or more lines. The following **Print** statement produces a two-line message:

```
Print "Map Layers:" + Chr$(10) + " World, Capitals"
```

The **Print** statement converts each Tab character (code 09) to a space (code 32).

Example

The next example displays the Message window, sets the window's size (three inches wide by one inch high), sets the window's font (Arial, bold, 10-point), and prints a message to the window.

```
Include "MAPBASIC.DEF" ' needed for color name 'BLUE'
Open Window Message ' open Message window
Set Window Message
   Font ("Arial", 1, 10, BLUE) ' Arial bold...
   Position (0.25, 0.25) ' place in upper left
   Width 3.0 ' make window 3" wide
   Height 1.0 ' make window 1" high
Print "MapBasic Dispatcher now on line"
```

The buffer size for message window text has been doubled to 8191 characters.

See Also:

Ask( ) function, **Close Window statement**, **Note statement**, **Open Window statement**, **Set Window statement**

Print # statement

Purpose

Writes data to a file opened in a Sequential mode (**Output** or **Append**).

Syntax

Print # *file_num* [, *expr*]

file_num is the number of a file opened through the **Open File statement**.

expr is an expression to write to the file.

Description

The **Print #** statement writes data to an open file. The file must be open and in a sequential mode which allows output (**Output** or **Append**).

The *file_num* parameter corresponds to the number specified in the **As** clause of the **Open File statement**.

MapInfo Professional writes the expression *expr* to a line of the file. To store a comma-separated list of expressions in each line of the file, use the **Write # statement** instead of **Print #**.

See Also:

Line Input statement, **Open File statement**, **Write # statement**

PrintWin statement

Purpose

Prints an existing window. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
PrintWin [ Window window_id ][ Interactive ][ File output_filename ]
   [ Overwrite ]
```

window_id is a window identifier.

output_filename is a string representing the name of an output file. If the output file already exists, an error will occur, unless the **Overwrite** keyword is specified.

Description

The **PrintWin** statement prints a window.

If the statement includes the optional **Window** clause, MapBasic prints the specified window; otherwise, MapBasic prints the active window.

The *window_id* parameter represents a window identifier; see the **FrontWindow( ) function** and the **WindowInfo( ) function** for more information about obtaining window identifiers.

If you include the **Interactive** keyword, MapInfo Professional displays the Print dialog box. If you omit the **Interactive** keyword, MapInfo Professional prints the window automatically, without displaying the dialog box.

Examples

Example 1

```
Dim win_id As Integer
Open Table "world"
Map From world
win_id = FrontWindow( )
'
' knowing the ID of the Map window,
' the program could now print the map by
' issuing the statement:
'
PrintWin Window win_id Interactive
```

Example 2

```
PrintWin Window FrontWindow( ) File "c:\output\file.plt"
```

See Also:

FrontWindow( ) function, **Run Menu Command statement**, **WindowInfo( ) function**

PrismMapInfo( ) function

Purpose

Returns properties of a Prism Map window. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

PrismMapInfo(*window_id*, *attribute*)

window_id is an integer window identifier.

attribute is an integer code, indicating which type of information should be returned.

Return Value

Float, logical, or string, depending on the attribute parameter.

Description

The **PrismMapInfo( )** function returns information about a Prism Map window.

The *window_id* parameter specifies which Prism Map window to query. To obtain a window identifier, call the **FrontWindow( ) function** immediately after opening a window, or call the **WindowID( ) function** at any time after the window's creation.

There are several numeric attributes that **PrismMapInfo( )** can return about any given Prism Map window. The *attribute* parameter tells the **PrismMapInfo( )** function which Map window statistic to return. The *attribute* parameter should be one of the codes from the following table; codes are defined in MAPBASIC.DEF.

Attribute	ID	Return Value
PRISMMAP_INFO_SCALE	1	Float result representing the PrismMaps scale factor.
PRISMMAP_INFO_BACKGROUND	4	Integer result representing the background color, see **RGB( ) function**.
PRISMMAP_INFO_LIGHT_X	6	Float result representing the x-coordinate of the light in the scene.
PRISMMAP_INFO_LIGHT_Y	7	Float result representing the y-coordinate of the Light in the scene.
PRISMMAP_INFO_LIGHT_Z	8	Float result representing the z-coordinate of the Light in the scene.
PRISMMAP_INFO_LIGHT_COLOR	9	Integer result representing the Light color, see **RGB( ) function**.

Attribute	ID	Return Value
PRISMMAP_INFO_CAMERA_X	10	Float result representing the x-coordinate of the Camera in the scene.
PRISMMAP_INFO_CAMERA_Y	11	Float result representing the y-coordinate of the Camera in the scene.
PRISMMAP_INFO_CAMERA_Z	12	Float result representing the z-coordinate of the Camera in the scene.
PRISMMAP_INFO_CAMERA_FOCAL_X	13	Float result representing the x-coordinate of the Cameras FocalPoint in the scene.
PRISMMAP_INFO_CAMERA_FOCAL_Y	14	Float result representing the y-coordinate of the Cameras FocalPoint in the scene.
PRISMMAP_INFO_CAMERA_FOCAL_Z	15	Float result representing the z-coordinate of the Camera's FocalPoint in the scene.
PRISMMAP_INFO_CAMERA_VU_1	16	Float result representing the first value of the ViewUp Unit Normal Vector.
PRISMMAP_INFO_CAMERA_VU_2	17	Float result representing the second value of the ViewUp Unit Normal Vector.
PRISMMAP_INFO_CAMERA_VU_3	18	Float result representing the third value of the ViewUp Unit Normal Vector.
PRISMMAP_INFO_CAMERA_VPN_1	19	Float result representing the first value of the View Plane Unit Normal Vector.
PRISMMAP_INFO_CAMERA_VPN_2	20	Float result representing the second value of the ViewPlane Unit Normal Vector.
PRISMMAP_INFO_CAMERA_VPN_3	21	Float result representing the third value of the ViewPlane Unit Normal Vector.
PRISMMAP_INFO_CAMERA_CLIP_NEAR	22	Float result representing the cameras near clipping plane.
PRISMMAP_INFO_CAMERA_CLIP_FAR	23	Float result representing the cameras far clipping plane.
PRISMMAP_INFO_INFOTIP_EXPR	24	String for Infotip. Not previously documented.

Example

This example prints out all the state variables specific to the PrismMap window:

```
include "Mapbasic.def"
```

```
Print "PRISMMAP_INFO_SCALE: " + PrismMapInfo(FrontWindow( ),
PRISMMAP_INFO_SCALE)
Print "PRISMMAP_INFO_BACKGROUND: " + PrismMapInfo(FrontWindow( ),
PRISMMAP_INFO_BACKGROUND)
Print "PRISMMAP_INFO_UNITS: " + PrismMapInfo(FrontWindow( ),
PRISMMAP_INFO_UNITS)
Print "PRISMMAP_INFO_LIGHT_X : " + PrismMapInfo(FrontWindow( ),
PRISMMAP_INFO_LIGHT_X )
Print "PRISMMAP_INFO_LIGHT_Y : " + PrismMapInfo(FrontWindow( ),
PRISMMAP_INFO_LIGHT_Y )
Print "PRISMMAP_INFO_LIGHT_Z: " + PrismMapInfo(FrontWindow( ),
PRISMMAP_INFO_LIGHT_Z)
Print "PRISMMAP_INFO_LIGHT_COLOR: " + PrismMapInfo(FrontWindow( ),
PRISMMAP_INFO_LIGHT_COLOR)
Print "PRISMMAP_INFO_CAMERA_X: " + PrismMapInfo(FrontWindow( ),
PRISMMAP_INFO_CAMERA_X)
Print "PRISMMAP_INFO_CAMERA_Y : " + PrismMapInfo(FrontWindow( ),
PRISMMAP_INFO_CAMERA_Y )
Print "PRISMMAP_INFO_CAMERA_Z : " + PrismMapInfo(FrontWindow( ),
PRISMMAP_INFO_CAMERA_Z )
Print "PRISMMAP_INFO_CAMERA_FOCAL_X: " + PrismMapInfo(FrontWindow( ),
PRISMMAP_INFO_CAMERA_FOCAL_X)
Print "PRISMMAP_INFO_CAMERA_FOCAL_Y: " + PrismMapInfo(FrontWindow( ),
PRISMMAP_INFO_CAMERA_FOCAL_Y)
Print "PRISMMAP_INFO_CAMERA_FOCAL_Z: " + PrismMapInfo(FrontWindow( ),
PRISMMAP_INFO_CAMERA_FOCAL_Z)
Print "PRISMMAP_INFO_CAMERA_VU_1: " + PrismMapInfo(FrontWindow( ),
PRISMMAP_INFO_CAMERA_VU_1)
Print "PRISMMAP_INFO_CAMERA_VU_2: " + PrismMapInfo(FrontWindow( ),
PRISMMAP_INFO_CAMERA_VU_2)
Print "PRISMMAP_INFO_CAMERA_VU_3: " + PrismMapInfo(FrontWindow( ),
PRISMMAP_INFO_CAMERA_VU_3)
Print "PRISMMAP_INFO_CAMERA_VPN_1: " + PrismMapInfo(FrontWindow( ),
PRISMMAP_INFO_CAMERA_VPN_1)
Print "PRISMMAP_INFO_CAMERA_VPN_2: " + PrismMapInfo(FrontWindow( ),
PRISMMAP_INFO_CAMERA_VPN_2)
Print "PRISMMAP_INFO_CAMERA_VPN_3: " + PrismMapInfo(FrontWindow( ),
PRISMMAP_INFO_CAMERA_VPN_3)
Print "PRISMMAP_INFO_CAMERA_CLIP_NEAR: " + PrismMapInfo(FrontWindow( ),
PRISMMAP_INFO_CAMERA_CLIP_NEAR)
Print "PRISMMAP_INFO_CAMERA_CLIP_FAR: " + PrismMapInfo(FrontWindow( ),
PRISMMAP_INFO_CAMERA_CLIP_FAR)
```

See Also:

Create PrismMap statement, **Set PrismMap statement**

ProgramDirectory$( ) function

Purpose

Returns the directory path to where the MapInfo Professional software is installed. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

```
ProgramDirectory$( )
```

Return Value

String

Description

The **ProgramDirectory$( )** function returns a string representing the directory path where the MapInfo Professional software is installed.

Example

```
Dim s_prog_dir As String
s_prog_dir = ProgramDirectory$( )
```

See Also:

HomeDirectory$( ) function, **SystemInfo( ) function**

ProgressBar statement

Purpose

Displays a dialog box with a Cancel button and a horizontal progress bar.

Syntax

```
ProgressBar status_message
   Calling handler
   [ Range n ]
```

status_message is a string value displayed as a message in the dialog box.

handler is the name of a Sub procedure.

n is a number at which the job is finished.

Restrictions

You cannot issue the **ProgressBar** statement through the MapBasic window.

Description

The **ProgressBar** statement displays a dialog box with a horizontal progress bar and a **Cancel** button. The bar indicates the percentage of completion of a lengthy operation. The user can halt the operation by clicking the **Cancel** button. Following the **ProgressBar** statement, a MapBasic program can call `CommandInfo(CMD_INFO_DLG_OK)` to determine whether the operation finished or whether the user cancelled first (see below). Where CMD_INFO_DLG_OK (1).

The *status_message* parameter is a string value, such as “Processing data…”, which is displayed in the dialog box.

The *handler* parameter is the name of a sub procedure in the same MapBasic program. As described below, the sub procedure must perform certain actions in order for it to interact with the **ProgressBar** statement.

The *n* parameter is a number, representing the count value at which the operation will be finished. For example, if an operation needs to process 7,000 rows of a table, the **ProgressBar** statement might specify 7000 as the *n* parameter. If no Range *n* clause is specified, the *n* parameter has a default value of 100.

When a program issues a **ProgressBar** statement, MapBasic calls the specified *handler* sub procedure. The sub procedure should perform a small amount of processing, specifically a few seconds' worth of processing at most, and then it should end. At that time, MapBasic checks to see if the user clicked the **Cancel** button. If the user did click **Cancel**, MapBasic removes the dialog box, and proceeds with the statements which follow the **ProgressBar** statement (and thus, the lengthy operation is never completed). Alternately, if the user did not click **Cancel**, MapBasic automatically calls the *handler* sub procedure again. If the user never clicks **Cancel**, the **ProgressBar** statement repeatedly calls the procedure until the operation is finished.

The *handler* procedure must be written in such a way that each call to the procedure performs only a small percent of the total job. Once a **ProgressBar** statement has been issued, MapBasic will repeatedly call the *handler* procedure until the user clicks **Cancel** or until the *handler* procedure indicates that the procedure is finished. The *handler* indicates the job status by assigning a value to the special MapBasic variable, also named ProgressBar.

If the *handler* assigns a value of negative one to the ProgressBar variable (`ProgressBar = -1`) then MapBasic detects that the operation is finished, and accordingly halts the ProgressBar loop and removes the dialog box. Alternately, if the *handler* procedure assigns a value other than negative one to the ProgressBar variable (`ProgressBar = 50`) then MapBasic re-displays the dialog box's “percent complete” horizontal bar, to reflect the latest figure of percent completion. MapBasic calculates the current percent of completion by dividing the current value of the ProgressBar variable by the Range setting, *n*. For example, if the **ProgressBar** statement specified the **Range** clause `Range 400` and if the current value of the ProgressBar variable is 100, then the current percent of completion is 25%, and MapBasic will display the horizontal bar as being 25% filled.

The statements following the **ProgressBar** statement often must determine whether the ProgressBar loop halted because the operation was finished, or because the user clicked the **Cancel** button. Immediately following the **ProgressBar** statement, the function call `CommandInfo(CMD_INFO_DLG_OK)` returns TRUE if the operation was complete, or FALSE if the operation halted because the user clicked cancel. Where CMD_INFO_DLG_OK (1).

Example

The following example demonstrates how a procedure can be written to work in conjunction with the **ProgressBar** statement. In this example, we have an operation involving 600 iterations; perhaps we have a table with 600 rows, and each row must be processed in some fashion. The main procedure issues the **ProgressBar** statement, which then automatically calls the sub procedure, write_out. The write_out procedure processes records until two seconds have elapsed, and then returns (so that MapBasic can check to see if the user pressed **Cancel**). If the user does not press **Cancel**, MapBasic will repeatedly call the write_out procedure until the entire task is done.

```
Include "mapbasic.def"
Declare Sub Main
Declare Sub write_out

Global next_row As Integer

Sub Main
   next_row = 1
   ProgressBar "Writing data..." Calling write_out Range 600
   If CommandInfo(CMD_INFO_STATUS) Then
      Note "Operation complete! Thanks for waiting."
   Else
      Note "Operation interrupted!"
   End If
End Sub
Sub write_out
   Dim start_time As Float
   start_time = Timer( )
   ' process records until either (a) the job is done,
   ' or (b) more than 2 seconds elapse within this call
   Do While next_row <= 600 And Timer( ) - start_time < 2
      '''''''''''''''''''''''''''''''''''''''''
      ''' Here, we would do the actual work  '''
      ''' of processing the file.            '''
      '''''''''''''''''''''''''''''''''''''''''
      next_row = next_row + 1
   Loop

   ' Now figure out why the Do loop terminated: was it
   ' because the job is done, or because more than 2
   ' seconds have elapsed within this iteration?
   If next_row > 600 Then
      ProgressBar = -1 'tell caller "All Done!"
   Else
      ProgressBar = next_row 'tell caller "Partly done"
   End If
End Sub
```

See Also:

CommandInfo( ) function, **Note statement**, **Print statement**

Proper$( ) function

Purpose

Returns a mixed-case string, where only the first letter of each word is capitalized. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Proper$(*string_expr*)

string_expr is a string expression.

Return Value

String

Description

The **Proper$( )** function first converts the entire *string_expr* string to lower case, and then capitalizes only the first letter of each word in the string, thus producing a result string with “proper” capitalization. This style of capitalization is appropriate for proper names.

Example

```
Dim name, propername As String

name = "ed bergen"
propername = Proper$(name)
' propername now contains the string "Ed Bergen"

name = "ABC 123"
propername = Proper$(name)
' propername now contains the string "Abc 123"

name = "a b c d"
propername = Proper$(name)
' propername now contains the string "A B C D"
```

See Also:

LCase$( ) function, **UCase$( ) function**

ProportionOverlap( ) function

Purpose

Returns a number that indicates what percentage of one object is covered by another object. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

ProportionOverlap(*object1*, *object2*)

object1 is the bottom object, and it is a closed object.

object2 is the top object, and it is a closed object.

Return Value

A float value equal to **AreaOverlap**(*object1*, *object2*) / **Area**(*object1*).

Restrictions

ProporationOverlap( ) only works on closed objects. If both objects are not closed (such as points and lines), then you may see an error message. Closed objects are objects that can produce an area, such as regions (polygons).

See Also:

AreaOverlap( ) function

Put statement

Purpose

Writes the contents of a MapBasic variable to an open file.

Syntax

Put [**#**] *filenum*, [*position*,] *var_name*

filenum is the number of a file opened through an **Open File statement**.

position is the file position to write to (does not apply to sequential file access).

var_name is the name of a variable which contains the data to be written.

Description

The **Put** statement writes to an open file.

If the **Open File statement** specified a sequential access mode (**Output** or **Append**), use the **Print # statement** or the **Write # statement** instead of **Put**.

If the **Open File statement** specified **Random** file access, the **Put** statement's **Position** clause can be used to indicate which record in the file to overwrite. When the file is opened, the file position points to the first record of the file (record 1). If the **Open File statement** specified Binary file access, one variable can be written at a time. The byte sequence written to the file depends on whether the hardware platform's byte ordering; see the **ByteOrder** clause of the **Open File statement**. The number of bytes written depends on the variable type, as summarized below:

Variable Type	Storage In File
Logical	One byte, either 0 or non-zero.
SmallInt	Two byte integer
Integer	Four byte integer
Float	Eight byte IEEE format
String	Length of string plus a byte for a 0 string terminator
Date	Four bytes: Small integer year, byte month, byte day
Other Variable types	Cannot be written.

The **Position** parameter sets the file pointer to a specific offset in the file. When the file is opened, the position is initialized to 1 (the start of the file). As a **Put** is done, the position is incremented by the number of bytes written. If the **Position** clause is not used, the **Put** simply writes to the current file position. If the file was opened in Binary mode, the **Put** statement cannot specify a variable-length string variable; any string variable used in a **Put** statement must be fixed-length. If the file was opened in Random mode, the **Put** statement cannot specify a fixed-length string variable which is longer than the record length of the file.

See Also:

EOF( ) function, **Get statement**, **Open File statement**, **Print # statement**, **Write # statement**

Randomize statement

Purpose

Initializes MapBasic's random number function. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Randomize [ With seed ]
```

seed is an integer expression.

Description

The **Randomize** statement “seeds” the random number generator so that later calls to the **Rnd( ) function** produce random results. Without this statement before the first call to the **Rnd( ) function**, the actual series of random numbers will follow a standard list. In other words, unless the program includes a **Randomize** statement, the sequence of values returned by the **Rnd( ) function** will follow the same pattern each time the application is run.

The **Randomize** statement is only needed once in a program and should occur prior to the first call to the **Rnd( ) function**.

If you include the **With** clause, the *seed* parameter is used as the seed value for the pseudo-random number generator. If you omit the **With** clause, MapBasic automatically seeds the pseudo-random number generator using the current system clock. Use the **With** clause if you need to create repeatable test scenarios, where your program generates repeatable sequences of "random" numbers.

Example

```
Randomize
```

See Also:

Rnd( ) function

RasterTableInfo( ) function

Purpose:

Returns information about a Raster or Grid Table. (WMS, Tile Server, and Seamless Raster tables not supported).

Syntax:

RasterTableInfo(table_id, attribute)

table_id is a string representing a table name, a positive integer table number, or 0 (zero). The table must be a raster or grid table.

attribute is an integer code indicating which aspect of the raster table to return.

Return Value

String, SmallInt, Integer or Logical, depending on the attribute parameter specified.

The attribute parameter can be any value from the table below. Codes in the left column (for example, RASTER_TAB_INFO_IMAGE_NAME) are defined in MAPBASIC.DEF.

attribute code	ID	RasterTableInfo() returns
RASTER_TAB_INFO_IMAGE_NAME	1	String result, representing the image file name associated with this raster table.
RASTER_TAB_INFO_WIDTH	2	Integer result, representing the width of the image, in pixels
RASTER_TAB_INFO_HEIGHT	3	Integer result, representing the height of the image, in pixels

attribute code	ID	RasterTableInfo() returns
RASTER_TAB_INFO_IMAGE_TYPE	4	SmallInt result, representing the type of image: • IMAGE_TYPE_RASTER (0) for raster images • IMAGE_TYPE_GRID (1) for grid images
RASTER_TAB_INFO_BITS_PER_PIXEL	5	SmallInt result, representing the number of bits/pixel for the raster data
RASTER_TAB_INFO_IMAGE_CLASS	6	SmallInt result, representing the image class: • IMAGE_CLASS_PALETTE (2) for palette images • IMAGE_CLASS_GREYSCALE (1) for greyscale images • IMAGE_CLASS_RGB (3) for RGB images • IMAGE_CLASS_BILEVEL (0) for 2 color bilevel images
RASTER_TAB_INFO_NUM_CONTROL_POINTS	7	SmallInt result, representing the number of control points. Use RasterControlPointInfo() and GeoControlPointInfo() to get specific control points.
RASTER_TAB_INFO_BRIGHTNESS	8	SmallInt result, representing the brightness as a percentage (0-100%)
RASTER_TAB_INFO_CONTRAST	9	SmallInt result, representing the contrast of the image as a percentage (0-100%)
RASTER_TAB_INFO_GREYSCALE	10	Logical result, representing if the image display should display as greyscale instead of the default image mode
RASTER_TAB_INFO_DISPLAY_TRANSPARENT	11	Logical result, representing if the image should display with a transparent color. If TRUE, RASTER_TAB_INFO_TRANSPARENT_COLOR represents the color that will be made transparent.
RASTER_TAB_INFO_TRANSPARENT_COLOR	12	Integer result, represent the color of the transparent pixels, as BGR.
RASTER_TAB_INFO_ALPHA	13	SmallInt result, representing the alpha factor for the translucency of the image (0-255)

RegionInfo( ) function

Purpose:

This function was created to determine the orientation of points in polygons -- whether they are ordered clockwise, or counter-clockwise. The only attribute the function reports on is the 'direction' of the points in a specified polygon. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax:

RegionInfo(*object*, *REGION_INFO_IS_CLOCKWISE*, *polygon_num*)

Where:

```
REGION_INFO_IS_CLOCKWISE    1
```

object refers to the object that is the subject of the function

REGION_INFO_IS_CLOCKWISE indicates whether the object is oriented in a clockwise or counterclockwise direction. A parameter of 1 indicates that the object is oriented in a clockwise order.

polygon_num indicates the polygon that is the subject of the function when an object contains more than one polygon.

Example:

If you were to select the state of Utah from States mapper and issued the following command in the MapBasic window, you would get a result of `F` or `False`, since the nodes in the single region of Utah are drawn in counter-clockwise order. Colorado's nodes are drawn in clockwise order and return `T` or `True`.

```
print RegionInfo(selection.obj,1,1)
```

ReadControlValue( ) function

Purpose

Reads the current status of a control in the active dialog box.

Syntax

ReadControlValue(*id_num*)

id_num is an integer value indicating which control to read.

Return Value

Integer, logical, string, Pen, Brush, Symbol, or Font, depending on the type of control

Description

The **ReadControlValue( )** function returns the current value of one of the controls in an active dialog box. A **ReadControlValue( )** function call is only valid while there is an active dialog box; thus, you may only call the **ReadControlValue( )** function from within a dialog box control's handler procedure.

The integer *id_num* parameter specifies which control MapBasic should read. If the *id_num* parameter has a value of -1 (negative one), the **ReadControlValue( )** function returns the value of the last control which was operated by the user. To explicitly specify which control you want to read, pass **ReadControlValue( )** an integer ID that identifies the appropriate control.

A dialog box control does not have a unique ID unless you include an **ID** clause in the **Dialog statement**'s **Control** clause. Some types of dialog box controls have no readable values (for example, static text labels).

The table below summarizes what types of values will be returned by various controls. Note that special processing is required for handling MultiListBox controls: since the user can select more than one item from a MultiListBox control, a program may need to call **ReadControlValue( )** multiple times to obtain a complete list of the selected items.

Control Type	ReadControlValue( ) Return Value
EditText	String, up to 32,767 bytes long, representing the current contents of the text box; if the EditText is tall enough to accommodate multiple lines of text, the string may include `Chr$(10)` values, indicating that the user entered line-feeds (for example, in Windows, by pressing Ctrl-Enter).
CheckBox	TRUE if the check box is currently selected, FALSE otherwise.
DocumentWindow	Integer that represents the HWND for the window control. This HWND should be passed as the parent window handle in the **Set Next Document statement**.
RadioGroup	SmallInt value identifying which button is selected (1 for the first button).
PopupMenu	SmallInt value identifying which item is selected (1 for the first item).
ListBox	SmallInt value identifying the selected list item (1 for the first, 0 if none).
BrushPicker	Brush value.
FontPicker	Font value.
PenPicker	Pen value.

Control Type	ReadControlValue( ) Return Value
SymbolPicker	Symbol value.
MultiListBox	Integer identifying one of the selected items. The user can select one or more of the items in a MultiListBox control. Since **ReadControlValue( )** can only return one piece of information at a time, your program may need to call **ReadControlValue( )** multiple times in order to determine how many items are selected. The first call to **ReadControlValue( )** returns the number of the first selected list item (1 if the first list item is selected); the second call will return the number of the second selected list item, etc. When **ReadControlValue( )** returns zero, the list of selected items has been exhausted. Subsequent calls to **ReadControlValue( )** then begin back at the top of the list of selected items. If **ReadControlValue( )** returns zero on the first call, none of the list items are selected.

Error Conditions

ERR_FCN_ARG_RANGE (644) error is generated if an argument is outside of the valid range.

ERR_INVALID_READ_CONTROL (842) error is generated if the **ReadControlValue( )** function is called when no dialog box is active.

Example

The following example creates a dialog box that asks the user to type a name in a text edit box. If the user clicks **OK**, the application calls **ReadControlValue( )** to read in the name that was typed.

```
Declare Sub Main
Declare Sub okhandler
Sub Main
   Dialog
      Title "Sign in, Please"
   Control OKButton
      Position 135, 120 Width 50
      Title "OK"
      Calling okhandler
   Control CancelButton
      Position 135, 100 Width 50
      Title "Cancel"
   Control StaticText
      Position 5, 10
      Title "Please enter your name:"
   Control EditText
      Position 55, 10 Width 160
      Value "(your name here)"
      Id 23 'arbitrary ID number
End Sub
Sub okhandler
   ' this sub is called when/if the user
```

```
   ' clicks the OK control
   Note "Welcome aboard, " + ReadControlValue(23) + "!"
End Sub
```

See Also:

Alter Control statement, **Dialog statement**, **Dialog Preserve statement**, **Dialog Remove statement**

ReDim statement

Purpose

Re-sizes an array variable.

Syntax

ReDim *var_name* (*newsize*) [, ...]

var_name is a string representing the name of an existing local or global array variable.

newsize is an integer value dictating the new array size. The maximum value is 32,767.

Description

The **ReDim** statement re-sizes (or “re-dimensions”) one or more existing array variables. The variable identified by *var_name* must have already been defined as an array variable through a **Dim statement** or a **Global statement**.

The **ReDim** statement can increase or decrease the size of an existing array. If your program no longer needs a given array variable, the **ReDim** statement can re-size that array to have zero elements (this minimizes the amount of memory required to store variables).

Unlike some BASIC languages, MapBasic does not allow custom subscript settings for arrays; a MapBasic array's first element always has a subscript of one.

If you store values in an array, and then enlarge the array through the **ReDim** statement, the values you stored in the array remain intact.

Example

```
Dim names_list(10) As String, cur_size As Integer
' The following statements determine the current
' size of the array, and then ReDim the array to
' a size 10 elements larger

cur_size = UBound(names_list)
ReDim names_list(cur_size + 10)

' The following statement ReDims the array to a
' size of zero elements. Presumably, this array
' is no longer needed, and it is resized to zero
' for the sake of saving memory.
```

```
ReDim names_list(0)
```

As shown below, the **ReDim** statement can operate on arrays of custom Type variables, and also on arrays that are Type elements.

```
Type customer
   name As String
   serial_nums(0) As Integer
End Type

Dim new_customers(1) As customer

' First, redimension the "new_customers" array,
' making it five items deep:

ReDim new_customers(5)

' Now, redimension the "serial_nums" array element
' of the first item in the "new_customers" array:

ReDim new_customers(1).serial_nums(10)
```

See Also:

Dim statement, **Global statement**, **UBound( ) function**

Register Table statement

Purpose

The Register Table statement builds a MapInfo Professional table from a spreadsheet, database, text file, raster, or grid image. Issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Register Table source_file {
   Type NATIVE |
   Type DBF [ Charset char_set ] |
   Type ASCII [ Delimiter delim_char ][ Titles ][ CharSet char_set ] |
   Type WKS [ Titles ] [ Range range_name ] |
   Type WMS Coordsys...
   Type WFS [ Charset char_set ] Coordsys... [ Symbol... ]
      [ Linestyle Pen(...) ] [ Regionstyle Pen(...) Brush(...) ]
      [Editable]
   Type XLS [ Titles ] [ Range range_name ] [ Interactive ] |
   Type ACCESS Table table_name [ Password pwd ] [ CharSet char_set ]}
   Type ODBC
      Connection { Handle connection_number | connection_string }
      Toolkit toolkit_name
      Cache { ON | OFF }
```

```
      [ Autokey { ON | OFF }]
      Table SQLQuery
      [ Versioned { ON | OFF }]
      [ Workspace Workspace_name ]
      [ ParentWorkspace ParentWorkspace_name ]
   Type GRID | Type RASTER
      [ ControlPoints ( MapX1, MapY1 ) ( RasterX1, RasterY1 ),
        ( MapX2, MapY2 ) ( RasterX2, RasterY2 ),
        ( MapX3, MapY3 ) ( RasterX3, RasterY3 )
        [, ... ]
      ]
      [ CoordSys ... ]
   Type FME [ Charset char_set ]
       CoordSys...
       Format format_type
       Schema feature_type
      [ Use Color ]
      [ Database ]
      [ SingleFile ]
      [ Symbol...]
      [ Linestyle Pen(...) ]
      [ Regionstyle Pen(...) Brush(...) ]
      [ Font ... ]
       Settings string1 [, string2 .. ]
  Type SHAPEFILE [ Charset char_set ] CoordSys auto
      [ PersistentCache { ON | OFF } ]
      [ Symbol...] [ Linestyle Pen(...) ]
      [ Regionstyle Pen(...) Brush(...) ]
      [ into destination_file ]
}
```

source_file is a string that represents the name of an existing database, spreadsheet, text file, raster, or grid image. If you are registering an Access table, this argument must identify a valid Access database.

char_set is the name of a character set; see **CharSet clause**. If not specified, then the system character set is used.

delim_char specifies the character used as a column delimiter. If the file uses Tab as the delimiter, specify 9. If the file uses commas, specify 44.

range_name is a string indicating a named range (for example, “MyTable”) or a cell range (for example, an Excel range can be specified as “Sheet1!R1C1:R9C6” or as “Sheet1!A1:F9”).

Interactive is optional for XLS, Grid, or Raster types. Specifying this for Grid or Raster types prompts the user for any missing control point or projection information. Not specifying this generates a .TAB file without user input, as when the user selects “Display” when opening a raster image from the **File > Open** dialog box. **Interactive** is not a valid parameter for registering shape (SHP) files.

Specifying the **Interactive** keyword for the XLS type, instructs the interface to display the Set Field Properties window when importing Excel files.

CoordSys is required for WMS and Shapefiles, the compiler indicates an error if it is missing. For other types, the **CoordSys** clause is optional. If **CoordSys** is specified, it overrides and replaces any coordinate system associated with the image. This is useful when registering a raster image that has an associated World file. For details, see **CoordSys clause**.

Symbol the symbol style to use for a point object created from a shapefile, see **Symbol clause**.

Pen the line style to use for a line object type created from a shapefile, see **Pen clause**.

Regionstyle Pen Brush the line style and fill style to be used for a region object type created from a shapefile, see the **Pen clause** and **Brush clause**.

table_name is a string that identifies an Access table.

pwd is the database-level password for the database, to be specified when database security is turned on.

connection_number is an integer value that identifies an existing connection to an ODBC database.

connection_string is a string used to connect to a database server. See **Server_ConnectInfo( ) function**.

toolkit_name is “ODBC” or “ORAINET.”

If **Autokey** is set **ON**, the table is registered with key auto-increment option. If **Autokey** is set **OFF** or this option is ignored, the table is registered without key auto-increment.

SQLQuery is the SQL query used to define the MapInfo table.

Versioned indicates if the table to be opened is a version-enabled (ON) table or not (OFF).

Workspace_name is the name of the current workspace in which the table will be operated. The name is case sensitive.

ParentWorkspace_name is the name of parent workspace of the current workspace.

ControlPoints are optional, but can be specified if the type is Grid or Raster. If the **ControlPoints** keyword is specified, it must be followed by at least 3 pairs of Map and Raster coordinates which are used to georegister an image. If the **ControlPoints** are specified, they will override and replace any control points associated with the image or an associated World file.

format_type formattype is a string that is used by FME to identify format that is opened.

feature_type specifies a featuretype (essentially schema name).

Use Color specifies if color information from dataset is used.

Database specifies if referenced datasource is from a database.

SingleFile specifies if referenced datasource consist of a single file.

string1 [, *string2* ..] These are Safe Software FME-specific settings that vary depending upon the format and settings options the user selects.

auto use this option if the Shapefile dataset has a .PRJ file, rather than specifying the coordinate system in the statement. If the .PRJ file does not exist or the coordinate system is not converted to a MapInfo coordinate system, the command will fail and the application will post an error message.

PersistentCache ON specifies if .MAP and .ID files generated during the opening of Shapefiles are saved on hard disk after closing a table. If **PersistentCache** is set to **OFF**, then these .MAP and .ID files are deleted after closing a table and are be generated each time the table is opened.

destination_file specifies the name to give to the MapInfo table (.TAB file). This string may include a path; if it does not include a path, the file is built in the same directory as the source file.

Description

Before you can use a non-native file (for example, a dBASE file) in MapInfo, you must register the file. The **Register Table** statement tells MapInfo Professional to examine a non-native file (for example, FILENAME.DBF) and build a corresponding table file (*filename*.TAB). Once the **Register Table** operation has built a table file, you can access the file as an MapInfo table.

The **Register Table** statement does not copy or alter the original data file. Instead, it scans the data, determines the datatypes of the columns, and creates a separate table file. The table is not opened automatically. To open the table, use an **Open Table statement**.

Each data file need only be registered once. Once the **Register Table** operation has built the appropriate table file, subsequent MapInfo Professional sessions simply Open the table, rather than repeat the **Register Table** operation.

The **Type** clause specifies where the file came from originally. This consists of the keyword **Type**, followed by one of the following character constants: **NATIVE**, **DBF**, **ASCII**, **WKS**, **WMS**, **WFS**, **XLS**, **ACCESS**, **ODBC**, **GRID**, **RASTER**, **FME**, or **SHAPEFILE**. The other information is necessary for preparing certain types of tables. If the type of file being registered is a grid, the coordsys string is read from the grid file and a MapInfo .TAB file is created. If a raster file is being registered, the .TAB file that is generated is the same as if the user selected “Display” when opening a raster image from the **File > Open** dialog box.

If the type of file being registered is a **GRID**, the coordsys string is read from the grid file and a MapInfo .TAB file is created. If a raster file is being registered, the .TAB file that is generated depends upon if georegistration information can be found in the image file or associated World file.

The **CharSet** clause specifies a character set. The *char_set* parameter should be a string such as “WindowsLatin1”. If you omit the CharSet clause, MapInfo Professional uses the default character set for the hardware platform that is in use at run-time. See **CharSet clause** for more information.

The **Delimiter** clause is followed by a string containing the delimiter character. The default delimiter is a TAB. The **Titles** clause indicates that the row before the range of data in the worksheet should be used as column titles. The **Range** clause allows the specification of a named range to use. The **into** clause is used to override the table name or location of the .TAB file. By default, it will be named the same as the data file, and stored in the same directory. However, when reading a read-only device such as a CD-ROM, you need to store the .TAB file on a volume that is not read-only.

Registering Access Tables

When you register an Access table, MapInfo Professional checks for a counter column with a unique index. If there is already a counter column, MapInfo Professional registers that column in the .TAB file. The column is read-only.

If the Access table does not have a counter column, MapInfo Professional modifies the Access table by adding a column called MAPINFO_ID with the counter datatype. In this case, the counter column does not display in MapInfo.

Do not alter the counter column in any way. It must be exclusively maintained automatically by MapInfo Professional.

Access datatypes are translated into the closest MapInfo datatypes. Special Access datatypes, such as OLE objects and binary fields, are not editable in MapInfo Professional.

Registering ODBC Tables

Before accessing a table live from a remote database, it is highly recommended that you first open a map table (for example, CANADA.TAB) for the database table. If you don't open a map table, the entire database table will be downloaded all at once, which could take a long time.

Open a map table and zoom in to an area that corresponds to a subset of rows you wish to see from the database table. For example, if you want to download rows pertaining to Ontario, zoom in to Ontario on the map. As a result, when you open the database table, only rows within the map window's MBR (minimum bounding rectangle), in this case Ontario, will be downloaded.

The following is a list of known problems/issues with live access:

- Every table must have a single unique key column.
- FastEdit is not supported.
- With MS ACCESS if the key is character, it does not display rows where the key value is less than the full column width for example, if the key is `char(5)` the value 'aaaa' will look like a deleted row.
- For Live Access, the **ReadOnly** checkbox on the save table dialog box is grayed out.
- Changes made by another user are not visible until a browser is scrolled or somehow refreshed. Inserts by another user are not seen until either: 1). An MBR search returns the row or 2). PACK command is issued in addition if cache is on another users updates may not appear until the cache is invalidated by a pan or zooming out.
- There will be a problem if a client-side join (through the **SQL Select** menu item or MapBasic) is done against two or more SPATIALWARE tables that are stored in different coordinate systems. This is not an efficient thing to do (it is better to do the join in the SQL statement that defines the table) but it is a problem in the current build.
- Oracle 7 tables that are indexed on a decimal field larger than 8 bytes will cause MapInfo Professional to crash when editing.
- If the server is Oracle, **Autokey** is the indicator to tell if the new feature, key auto-increment, will be used or not.
- If the **Cache OFF** statement is before the connection string an error will be generated at compile time.

Registering Shapefiles

When you register shapefiles, they can be opened in MapInfo Professional with read-only access. Since a shapefile itself does not contain projection information, you must specify a **CoordSys** clause. It is also possible to set styles that will be used when shapefile objects are displayed in MapInfo Professional. Projection and style information is stored as metadata in the TAB file.

Interactive is not a valid parameter to use when registering SHP files.

Example: DBF

```
Register Table "c:\mapinfo\data\rpt23.dbf"
   Type DBF
   Into "Report23"

Open Table "c:\mapinfo\data\Report23"
```

Example: ODBC

```
Open Table "C:\Data\CANADA\Canada.tab" Interactive
Map From Canada
set map redraw off
Set Map Zoom 1000 Units "mi"
set map redraw on
Register Table "odbc_cancaps"
   TYPE ODBC
   TABLE "Select * From schemaname.can_caps"
   CONNECTION
      DSN=dsnname;UID=username;PWD=password;DATABASE=dbname
      SERVER=servername
   Into
      "D:\MI\odbc_cancaps.TAB"
Open Table "D:\MI\odbc_cancaps.TAB" Interactive
Map From odbc_cancaps
```

Example: RASTER

Registering a completely georeferenced raster image (the raster handler can return at least three control points and a projection).

```
Register Table "GeoRef.tif" type RASTER into "GeoRef.TAB"
```

Registering a raster image that has an associated World file containing control point information, but no projection.

```
Register Table "RasterWithWorld.tif" type RASTER coordsys earth projection
9, 62, "m", -96, 23, 29.5, 45.5, 0, 0 into "RasterWithWorld.TAB"
```

Registering a raster image that has no control point or projection information.

```
Register Table "NoRegistration.BMP" type RASTER controlpoints (1000,2000)
(1,2), (2000,3000) (2, 3), (5000,6000) (5,6) coordsys earth projection 9,
62, "m", -96, 23, 29.5, 45.5, 0, 0 into "NoRegistration.tab"
```

Example: SHAPEFILE

The following example registers a shapefile.

```
Register Table "C:\Shapefiles\CNTYLN.SHP" TYPE SHAPEFILE Charset
"WindowsLatin1" CoordSys Earth Projection 1, 33 PersistentCache Off
linestyle Pen (2,26,16711935) Into "C:\Temp\CNTYLN.TAB"
Open Table "C:\Temp\CNTYLN.TAB" Interactive
Map From CNTYLN
```

Example: ODBC

The following example creates a tab file and then opens the tab file.

```
Register Table "SMALLINTEGER"  TYPE ODBC
   TABLE "Select *  From ""MIPRO"".""SMALLINTEGER"""
   CONNECTION "SRVR=scout;UID=mipro;PWD=mipro "
   toolkit  "ORAINET"
   Autokey ON
   Into
   "C:\projects\data\testscripts\english\remote\SmallIntEGER.TAB"
Open Table "C:\Projects\Data\TestScripts\English\remote\SmallIntEGER.TAB"
Interactive
Map From SMALLINTEGER
```

The following example creates a tab file and then opens the tab file. This example uses a workspace.

```
Register Table "Gwmusa" TYPE ODBC
    TABLE "Select * From ""MIUSER"".""GWMUSA"""
    CONNECTION "SRVR=troyny;UID=miuser;PWD=miuser"
    toolkit "ORAINET"
    Versioned On
    Workspace "MIUSER"
    ParentWorkspace "LIVE"
    Into "C:\projects\data\testscripts\english\remote\Gwmusa.tab"
Open Table "C:\Projects\Data\TestScripts\English\remote\Gwmusa.TAB"
Interactive Map From Gwmusa
```

Example: FME (Universal Data)

```
Register Table "D:\MUT\DWG\Data\africa_miller.DWG" Type FME
CoordSys Earth Projection 11, 104, "m", 0 Format "ACAD" Schema
"africa_miller" Use Color SingleFile  Symbol (35,0,16)  Linestyle Pen
(1,2,0)  RegionStyle Pen (1,2,0)  Brush (2,16777215,16777215)   Font
("Arial",0,9,0) Settings
"RUNTIME_MACROS","METAFILE,acad,_EXPAND_BLOCKS,yes,ACAD_IN_USE_BLOCK_HEAD
ER_LAYER,yes,ACAD_IN_RESOLVE_ENTITY_COLOR,yes,_EXPAND_VISIBLE,yes,_BULGES
_AS_ARCS,no,_STORE_BULGE_INFO,no,_READ_PAPER_SPACE,no,ACAD_IN_READ_GROUPS
,no,_IGNORE_UCS,no,_ACADPreserveComplexHatches,no,_MERGE_SCHEMAS,YES",
"META_MACROS","Source_EXPAND_BLOCKS,yes,SourceACAD_IN_USE_BLOCK_HEADER_LA
YER,yes,SourceACAD_IN_RESOLVE_ENTITY_COLOR,yes,Source_EXPAND_VISIBLE,yes,
Source_BULGES_AS_ARCS,no,Source_STORE_BULGE_INFO,no,Source_READ_PAPER_SPA
CE,no,SourceACAD_IN_READ_GROUPS,no,Source_IGNORE_UCS,no,Source_ACADPreser
```

```
veComplexHatches,no", "METAFILE","acad", "COORDSYS","", "IDLIST","" Into
"C:\Temp\africa_miller.tab"
Open table  "C:\Temp\africa_miller.tab"
Map From "africa_miller"
```

Supporting Transaction Capabilities for WFS Layers

Syntax

```
Register Table source_file
   { Type NATIVE |
   Type DBF [ Charset char_set ] |
   Type ASCII [ Delimiter delim_char ][ Titles ][ CharSet char_set ] |
   Type WKS [ Titles ] [ Range range_name ] |
   Type WMS Coordsys...
   Type WFS [ Charset char_set ] Coordsys... [ Symbol... ]
      [ Linestyle Pen(...) ] [ Regionstyle Pen(...) Brush(...) ]
      [Editable]
```

where:

Editable reflects the Allow Edits choice.

See Also:

Open Table statement, **Create Table statement**, **Server Create Workspace statement**, **Server Link Table statement**

Relief Shade statement

Purpose

Adds relief shade information to an open grid table. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Relief Shade
   Grid tablename
   Horizontal xy_plane_angle
   Vertical incident_angle
   Scale z_scale_factor
```

tablename is the alias name of the grid to which relief shade information is being calculated.

xy_plane_angle is the direction angle, in degrees, of the light source in the horizontal or xy plane. An *xy_plane_angle* of zero represents a light source shining from due East. A positive angle places the light source counterclockwise, so to place the light source in the NorthWest, set *xy_plane_angle* to 135.

incident_angle is the angle of the light source above the horizon or xy plane. An *incident_angle* of zero represents a light source right at the horizon. An *incident_angle* of 90 places the light source directly overhead.

z_scale_factor is the scale factor applied to the z-component of each grid cell. Increasing the *z_scale_factor* enhances the shading effect by exaggerating the vertical component. This can be used to bring out more detail in relatively flat grids.

Example

```
Relief Shade
   Grid Lumens
   Horizontal 135
   Vertical 45
   Scale 30
```

Reload Symbols statement

Purpose

Opens and reloads the MapInfo symbol file; this can change the set of symbols displayed in the Options > Symbol Style dialog box. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax 1 (MapInfo 3.0 Symbols)

`Reload Symbols`

Syntax 2 (Bitmap File Symbols)

`Reload Custom Symbols From` *directory*

directory is a string representing a directory path.

Description

This statement is used by the SYMBOL.MBX utility, which allows users to create custom symbols.

MapInfo 3.0 Symbols refers to the symbol set that came with MapInfo Professional for Windows 3.0 and has been maintained in subsequent versions of MapInfo Professional.

See Also:

Alter Object statement

RemoteMapGenHandler procedure

Purpose

A reserved procedure name, called when an OLE Automation client calls the MapGenHandler Automation method.

Syntax

```
Declare Sub RemoteMapGenHandler

Sub RemoteMapGenHandler
    statement_list
End Sub
```

statement_list is a list of MapBasic statements to execute when the OLE Automation client calls the MapGenHandler method.

Description

RemoteMapGenHandler is a special-purpose MapBasic procedure name, which is invoked through OLE Automation. If you are using OLE Automation to control MapInfo Professional, and you call the MapGenHandler method, MapInfo Professional calls the **RemoteMapGenHandler** procedures of any MapBasic applications that are running. The MapGenHandler method is part of the MapGen Automation model introduced in MapInfo Professional 4.1.

The MapGenHandler Automation method takes one argument: a string. Within the RemoteMapGenHandler procedure, you can retrieve the string argument by issuing the function call `CommandInfo(CMD_INFO_MSG)` and assigning the results to a string variable.

Example

For an example of using **RemoteMapGenHandler**, see the sample program MAPSRVR.MB.

RemoteMsgHandler procedure

Purpose

A reserved procedure name, called when a remote application sends an execute message.

Syntax

```
Declare Sub RemoteMsgHandler

Sub RemoteMsgHandler
    statement_list
End Sub
```

statement_list is a list of statements to execute upon receiving an execute message.

Description

RemoteMsgHandler is a special-purpose MapBasic procedure name that handles inter-application communication. If you run a MapBasic application that includes a procedure named **RemoteMsgHandler**, MapInfo Professional automatically calls the **RemoteMsgHandler** procedure every time another application (for example, a spreadsheet or database package) issues an "execute" command. The MapBasic procedure then can call the **CommandInfo( ) function** to retrieve the string corresponding to the execute command.

You can use the **End Program statement** to terminate a **RemoteMsgHandler** procedure once it is no longer wanted. Conversely, you should be careful not to issue an **End Program statement** while the **RemoteMsgHandler** procedure is still needed.

Inter-Application Communication Using Windows DDE

If a Windows application is capable of conducting a DDE (Dynamic Data Exchange) conversation, that application can initiate a conversation with MapInfo Professional. In the conversation, the external application is the client (active party), and a specific MapBasic application is the server (passive party).

Each time the DDE client sends an execute command, MapInfo Professional calls the server's **RemoteMsgHandler** procedure. Within the **RemoteMsgHandler** procedure, you can use the function call:

```
CommandInfo(CMD_INFO_MSG)
```

where:

```
CMD_INFO_MSG (1000)
```

to retrieve the string sent by the remote application. The DDE conversation must use the name of the sleeping application (for example, “C:\MAPBASIC\DISPATCH.MBX”) as the topic in order to facilitate **RemoteMsgHandler** functionality.

See Also:

DDEExecute statement, **DDEInitiate( ) function**, **SelChangedHandler procedure**, **ToolHandler procedure**, **WinChangedHandler procedure**, **WinClosedHandler procedure**

RemoteQueryHandler( ) function

Purpose

A special function, called when a MapBasic program acts as a DDE server, and the DDE client performs a “peek” request. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

```
Declare Function RemoteQueryHandler( ) As String

Function RemoteQueryHandler( ) As String
    statement_list
End Function
```

statement_list is a list of statements to execute upon receiving a peek request.

Description

The **RemoteQueryHandler( )** function works in conjunction with DDE (Dynamic Data Exchange). For an introduction to DDE, see the *MapBasic User Guide*. An external application can initiate a DDE conversation with your MapBasic program. To initiate the conversation, the external application

uses “MapInfo” as the DDE application name, and it uses the name of your MapBasic application as the DDE topic. Once the conversation is initiated, the external application (the client) can issue peek requests to request data from your MapBasic application (the server).

To handle peek requests, include a function called **RemoteQueryHandler( )** in your MapBasic application. When the client application issues a peek request, MapInfo Professional automatically calls the **RemoteQueryHandler( )** function. The client's peek request is handled synchronously; the client waits until **RemoteQueryHandler( )** returns a value.

The DDE client can peek at the global variables in your MapBasic program, even if you do not define a **RemoteQueryHandler( )** function. If the client issues a peek request using the name of a MapBasic global variable, MapInfo Professional automatically returns the global's value to the client instead of calling **RemoteQueryHandler( )**. In other words, if the data you want to expose is already stored in global variables, you do not need **RemoteQueryHandler( )**.

Example

The following example calls the **CommandInfo( ) function** to determine the item name specified by the DDE client. The item name is used as a flag; in other words, this program decides which value to return based on whether the client specified “code1” as the item name.

```
Function RemoteQueryHandler( ) As String
   Dim s_item_name As String

   s_item_name = CommandInfo(CMD_INFO_MSG)

   If s_item_name = "code1" Then
      RemoteQueryHandler = custom_function_1( )
   Else
      RemoteQueryHandler = custom_function_2( )
   End If

End Function
```

See Also:

DDEInitiate( ) function, **RemoteMsgHandler procedure**

Remove Cartographic Frame statement

Purpose

Allows you to remove cartographic frames from an existing cartographic legend created with the **Create Cartographic Legend statement**. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Cartographic Frame
   [ Window legend_window_id ]
   Id frame_id, frame_id, frame_id, ...
```

legend_window_id is an integer window identifier which you can obtain by calling the **FrontWindow( ) function** and the **WindowID( ) function**.

frame_id is the ID of the frame on the legend. You cannot use a layer name. For example, three frames on a legend would have the successive IDs, 1, 2, and 3.

See Also:

Add Cartographic Frame statement, **Alter Cartographic Frame statement**, **Create Cartographic Legend statement**, **Set Cartographic Legend statement**

Remove Map statement

Purpose

Removes one or more layers from a Map window. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Remove Map [ Window window_id ]
   Layer map_layer [ , map_layer ... ] | GroupLayer group_id [ , group_id
... ][ Interactive ]
```

window_id is the integer window identifier of a Map window; to obtain a window identifier, call the **FrontWindow( ) function** or the **WindowID( ) function**.

map_layer specifies which map layer(s) to remove; see examples below.

Description

The **Remove Map** statement removes one or more layers or group layers from a Map window. If no *window_id* is provided, the statement affects the topmost Map window.

The group_id can be an integer greater than zero to denote a specific group in the map, or the name of a group layer. If it is the name of a group layer, the first group layer in the list from the top down with the same name will be removed. Since the map_layer also refers to a unique identifier it can refer to a map layer in any group. But to remove an entire group, and all of its nested groups, use the **GroupLayer** clause.The *map_layer* parameter can be an integer greater than zero, a string containing the name of a table, or the keyword **Animate**, as summarized in the following table.

Examples	Descriptions of Examples
`Remove Map Layer 1`	If you specify “1” (one) as the *map_layer* parameter, the top map layer (other than the Cosmetic layer) is removed. Specify “1, 2" to remove the top two layers. Example: `Remove Map GroupLayer 1` This removes the first group layer in the list.
`Remove Map Layer "Zones"`	The Zones layer is removed (assuming that one of the layers in the map is named “Zones”).
`Remove Map Layer "Zones(1)"`	The first thematic layer based on the Zones layer is removed.
`Remove Map Layer Animate`	The animation layer is removed. To learn how to add an animation layer, see **Add Map statement**.

If you include the **Interactive** keyword, and if the layer removal will cause the loss of labels or themes, MapInfo Professional displays a dialog box that allows the user to save (a workspace), discard the labels and themes, or cancel the layer removal. If you omit the **Interactive** keyword, the user is not prompted.

A **Remove Map** statement does not close any tables; it only affects the number of layers displayed in the Map window. If a **Remove Map** statement removes the last non-cosmetic layer in a Map window, MapInfo Professional automatically closes the window.

See Also:

Create Map statement, **Map statement**, **Set Map statement**

Rename File statement

Purpose

Changes the name of a file. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

`Rename File` *`old_filespec`* **`As`** *`new_filespec`*

old_filespec is a string representing an existing file's name (and, optionally, path); the file must not be open.

new_filespec is a string representing the new name (and, optionally, path) for the file.

Description

The **Rename File** statement renames a file.

The *new_filespec* parameter specifies the file's new name. If *new_filespec* contains a directory path that differs from the file's original location, MapInfo Professional moves the file to the specified directory.

Example

```
Rename File "startup.wor" As "startup.bak"
```

See Also:

Rename File statement, **Save File statement**

Rename Table statement

Purpose

Changes the names (and, optionally, the location) of the files that make up a table. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Rename Table *table* **As** *newtablespec*

table is the name of an open table.

newtablespec is the new name (and, optionally, path) for the table.

Description

The **Rename Table** statement assigns a new name to an open table.

The *newtablespec* parameter specifies the table's new name. If *newtablespec* contains a directory name, MapBasic attempts to move the table to the specified directory in addition to renaming the table. The **Rename Table** statement renames the physical files which comprise a table. This effect is permanent (unless/until another **Rename Table** statement is issued).

This action can invalidate existing workspaces. Any workspaces created before the renaming operation will refer to the table by its previous, no-longer-applicable name.

Do not use the **Rename Table** statement to assign a temporary, working table name. If you need to assign a temporary name, use the **Open Table statement**'s optional **As** clause.

The **Rename Table** statement cannot rename a table that is actually a “view.” For example, a StreetInfo table (such as SF_STRTS) is actually a view, combining two other tables (SF_STRT1 and SF_STRT2). You could not rename the SF_STRTS table by calling **Rename Table**. You cannot

rename temporary query tables (for example, QUERY1). You cannot rename tables that have unsaved edits; if a table has unsaved edits, you must either save or discard the edits (or Rollback) before renaming.

Example

The following example renames the table casanfra as sf_hiway.

```
Open Table "C:\DATA\CASANFRA.TAB"
Rename Table CASANFRA As "SF_HIWAY.TAB"
```

The following example renames a table and moves it to a different directory path.

```
Open Table "C:\DATA\CASANFRA.TAB"
Rename Table CASANFRA As "c:\MAPINFO\SF_HIWAY"
```

See Also:

Close Table statement, **Drop Table statement**

Reproject statement

Purpose

Allows you to specify which columns should appear the next time a table is browsed. This statement has been deprecated.

Resume statement

Purpose

Returns from an **OnError** error handler.

Syntax

Resume { **0** | **Next** | *label* }

label is a label within the same procedure or function.

Restrictions

You cannot issue a **Resume** statement through the MapBasic window.

Description

The **Resume** statement tells MapBasic to return from an error-handling routine.

The **OnError statement** enables an error-handling routine, which is a group of statements MapBasic carries out in the event of a run-time error. Typically, each error-handling routine includes one or more **Resume** statements. The **Resume** statement causes MapBasic to exit the error-handling routine.

The various forms of the **Resume** statement let the application dictate which statement MapBasic is to execute after exiting the error-handling routine:

A **Resume 0** statement tells MapBasic to retry the statement which generated the error.

A **Resume Next** statement tells MapBasic to go to the first statement following the statement which generated the error.

A **Resume** *label* statement tells MapBasic to go to the line identified by the label. Note that the label must be in the same procedure.

Example

```
...
   OnError GoTo no_states
   Open Table "states"
   Map From states
after_mapfrom:
...
   End Program
no_states:
   Note "Could not open States; no Map used."
   Resume after_mapfrom
```

See Also:

Err( ) function, **Error statement**, **Error$( ) function**, **OnError statement**

RGB( ) function

Purpose

Returns an RGB color value calculated from Red, Green, Blue components. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

RGB(*red*, *green*, *blue*)

red is a numeric expression from 0 to 255, representing a concentration of red.

green is a numeric expression from 0 to 255, representing a concentration of green.

blue is a numeric expression from 0 to 255, representing a concentration of blue.

Return Value

Integer

Description

Some MapBasic statements allow you to specify a color as part of a pen or brush definition (for example, the **Create Point statement**). MapBasic pen and brush definitions require that each color be specified as a single integer value, known as an RGB value. The **RGB( )** function lets you calculate such an RGB value.

Colors are often defined in terms of the relative concentrations of three components—the red, green and blue components. Accordingly, the **RGB( )** function takes three parameters—red, green, and blue—each of which specifies the concentration of one of the three primary colors. Each color component should be an integer value from 0 to 255, inclusive.

The RGB value of a given color is calculated by the formula:

```
( red * 65536) + ( green * 256) + blue
```

The standard definitions file, MAPBASIC.DEF, includes **Define** statements for several common colors (BLACK, WHITE, RED, GREEN, BLUE, CYAN, MAGENTA, and YELLOW). If you want to specify red, you can simply use the identifier RED instead of calling **RGB( )**.

Example

```
Dim red,green,blue,color As Integer
red = 255
green = 0
blue = 0
color = RGB(red, green, blue)

' the RGB value stored in the variable: color
' will represent pure, saturated red.
```

See Also:

Brush clause, **Font clause**, **Pen clause**, **Symbol clause**

Right$( ) function

Purpose

Returns part or all of a string, beginning at the right end of the string. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Right$(*string_expr*, *num_expr*)

string_expr is a string expression.

num_expr is a numeric expression.

Return Value

String

Description

The **Right$( )** function returns a string which consists of the rightmost *num_expr* characters of the string expression *string_expr*.

The *num_expr* parameter should be an integer value, zero or larger. If *num_expr* has a fractional value, MapBasic rounds to the nearest integer. If *num_expr* is zero, **Right$( )** returns a null string. If *num_expr* is larger than the number of characters in the *string_expr* string, **Right$( )** returns a copy of the entire *string_expr* string.

Example

```
Dim whole, partial As String
whole = "Afghanistan"
partial = Right$(whole, 4)

' at this point, partial contains the string: "stan"
```

See Also:

Left$( ) function, **Mid$( ) function**

Rnd( ) function

Purpose

Returns a random number. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Rnd(*list_type*)

list_type selects the kind of random number list.

Return Value

A number of type float between 0 and 1 (exclusive).

Description

The **Rnd( )** function returns a random floating-point number, greater than zero and less than one.

The conventional use is of the form **Rnd**(1), in which the function returns a random number. The sequence of random numbers is always the same unless you insert a **Randomize statement** in the program. Any positive list_type parameter value produces this type of result.

A less common use is the form **Rnd**(0), which returns the previous random number generated by the **Rnd( )** function. This functionality is provided primarily for debugging purposes.

A very uncommon use is a call with a negative *list_type* value, such as **Rnd**(-1). For a given negative value, the **Rnd( )** function always returns the same number, regardless of whether you have issued a **Randomize statement**. This functionality is provided primarily for debugging purposes.

Example

```
Chknum = 10 * Rnd(1)
```

See Also:

Randomize statement

Rollback statement

Purpose

Discards a table's unsaved edits. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Rollback Table *tablename*

tablename is the name of an open table.

Description

If the specified table has been edited, but the edits have not been saved, the **Rollback** statement discards the unsaved edits. The user can obtain the same results by choosing **File > Revert**, except that command displays a dialog box.

When you Rollback a query table, MapInfo Professional discards any unsaved edits in the permanent table used for the query (except in cases where the query produces a join, or the query produces aggregated results, for example, using the **Select statement**'s **Group By** clause).

For example, if you edit a permanent table (such as WORLD), make a selection from WORLD, and browse the selection, MapInfo Professional will "snapshot" the Selection table, and call the snapshot (something like) QUERY1. If you then Rollback the QUERY1 table, MapInfo Professional discards any unsaved edits in the WORLD table, since the WORLD table is the table on which QUERY1 is based.

Using a **Rollback** statement on a linked table discards the unsaved edits and returns the table to the state it was in prior to the unsaved edits.

Example

```
If keep_changes Then
    Table towns
```

```
Else
   Rollback Table towns
End If
```

See Also:

Commit Table statement

Rotate( ) function

Purpose

Allows an object (not a text object) to be rotated about the rotation anchor point. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Rotate(*object*, *angle*)

object represents an object that can be rotated. It cannot be a text object.

angle is a float value that represents the angle (in degrees) to rotate the object.

Return Value

A rotated object.

Description

The **Rotate( )** function Rotates all object types except for text objects without altering the source object in any way.

To rotate text objects, use the **Alter Object OBJ_GEO_TEXTANGLE statement**.

If an arc, ellipse, rectangle, or rounded rectangle is rotated, the resultant object is converted to a polyline/polygon so that the nodes can be rotated.

Example

```
dim RotateObject as object
Open Table "C:\MapInfo_data\TUT_USA\USA\STATES.TAB"
map from states
select * from States where state = "IN"
RotateObject = rotate(selection.obj, 45)
insert into states (obj) values (RotateObject)
```

See Also:

RotateAtPoint( ) function

RotateAtPoint( ) function

Purpose

Allows an object (not a text object) to be rotated about a specified anchor point. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

RotateAtPoint(*object*, *angle*, *anchor_point_object*)

object represents an object that can be rotated. It cannot be a text object.

angle is a float value that represents the angle (in degrees) to rotate the object.

anchor_point_object is an object representing the anchor point which the object nodes are rotated about.

Return Value

A rotated object.

Description

The **RotateAtPoint( )** function rotates all object types except for text objects without altering the source object in any way.

To rotate text objects, use the **Alter Object OBJ_GEO_TEXTANGLE statement**.

If an arc, ellipse, rectangle, or rounded rectangle is rotated, the resultant object is converted to a polyline/polygon so that the nodes can be rotated.

Example

```
dim RotateAtPointObject as object
dim obj1 as object
dim obj2 as object
Open Table "C:\MapInfo_data\TUT_USA\USA\STATES.TAB" ]
map from states
select * from States where state = "CA"
obj1 = selection.obj
select * from States where state = "NV"
obj2 = selection.obj
oRotateAtPointObject = RotateAtPoint(obj1 , 65, centroid(obj2))
insert into states (obj) values (RotateAtPointObject )
```

See Also:

Rotate( ) function

Round( ) function

Purpose

Returns a number obtained by rounding off another number. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Round(*num_expr*, *round_to*)

num_expr is a numeric expression.

round_to is the number to which *num_expr* should be rounded off.

Return Value

Float

Description

The **Round( )** function returns a rounded-off version of the numeric *num_expr* expression.

The precision of the result depends on the *round_to* parameter. The **Round( )** function rounds the *num_expr* value to the nearest multiple of the *round_to* parameter. If *round_to* is 0.01, MapInfo Professional rounds to the nearest hundredth; if *round_to* is 5, MapInfo Professional rounds to the nearest multiple of 5; etc.

Example

```
Dim x, y As Float
x = 12345.6789

y = Round(x, 100)
' y now has the value 12300

y = Round(x, 1)
' y now has the value 12346

y = Round(x, 0.01)
' y now has the value 12345.68
```

See Also:

Fix( ) function, **Format$( ) function**, **Int( ) function**

RTrim$( ) function

Purpose

Trims space characters from the end of a string, and returns the results. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

RTrim$(*string_expr*)

string_expr is a string expression.

Return Value

String

Description

The **RTrim$( )** function removes any spaces from the end of the *string_expr* string, and returns the resultant string.

Example

```
Dim s_name As String
s_name = RTrim$("Mary Smith ")

' s_name now contains the string "Mary Smith"
' (no spaces at the end)
```

See Also:

LTrim$( ) function

Run Application statement

Purpose

Runs a MapBasic application or adds a MapInfo workspace. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Run Application [NoMRU] *file*

file is the name of an application file or a workspace file.

If the statement includes the **NoMRU** clause, the application or workspace name would not be added to the Most recently Used list of files.

Description

The **Run Application** statement runs a MapBasic application or loads an MapInfo workspace. By issuing a **Run Application** statement, one MapBasic application can run another application. To do so, the *file* parameter must represent the name of a compiled application file. The **Run Application** statement cannot run an uncompiled application. To halt an application launched by the **Run Application** statement, use the **Terminate Application statement**.

Example

The following statement runs the MapBasic application, REPORT.MBX:

```
Run Application "C:\MAPBASIC\APP\REPORT.MBX"
```

The following statement loads the workspace, PARCELS.WOR:

```
Run Application "Parcels.wor"
```

See Also:

Run Command statement, **Run Menu Command statement**, **Run Program statement**, **Terminate Application statement**

Run Command statement

Purpose

Executes a MapBasic command represented by a string. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Run Command *command*

command is a character string representing a MapBasic statement.

Description

The **Run Command** statement interprets a character string as a MapBasic statement, then executes the statement.

The **Run Command** statement has some restrictions, due to the fact that the *command* parameter is interpreted at run-time, rather than being compiled. You cannot use a **Run Command** statement to issue a **Dialog statement**. Also, variable names may not appear within the *command* string; that is, variable names may not appear enclosed in quotes. For example, the following group of statements would not work, because the variable names *x* and *y* appear inside the quotes that delimit the command string:

```
' this example WON'T work
Dim cmd_string As String
Dim x, y As Float
```

```
cmd_string = " x = Abs(y) "
Run Command cmd_string
```

However, variable names can be used in the construction of the *command* string.

In the following example, the *command* string is constructed from an expression that includes a character variable.

```
'this example WILL work
Dim cmd_string As String
Dim map_it, browse_it As Logical

Open Table "world"
If map_it Then
   cmd_string = "Map From "
   Run Command cmd_string + "world"
End If
If browse_it Then
   cmd_string = "Browse * From "
   Run Command cmd_string + "world"
End If
```

Example

The **Run Command** statement provides a flexible way of issuing commands that have variable-length argument lists. For example, the **Map From statement** can include a single table name, or a comma-separated list of two or more table names. An application may need to decide at run time (based on feedback from the user) how many table names should be included in the **Map From statement**. One way to do this is to construct a text string at run time, and execute the command through the **Run Command** statement.

```
Dim cmd_text As String
Dim cities_wanted, counties_wanted As Logical

Open Table "states"
Open Table "cities"
Open Table "counties"

cmd_text = "states" ' always include STATES layer

If counties_wanted Then
   cmd_text = "counties, " + cmd_text
End If

If cities_wanted Then
   cmd_text = "cities, " + cmd_text
End If

Run Command "Map From " + cmd_text
```

The following example shows how to duplicate a Map window, given the window ID of an existing map. The **WindowInfo( ) function** returns a string containing MapBasic statements; the **Run Command** statement executes the string.

```
Dim i_map_id As Integer

' First, get the ID of an existing Map window
' (assuming the Map window is the active window):
i_map_id = FrontWindow( )

' Now clone the active map window:
Run Command WindowInfo(i_map_id, WIN_INFO_CLONEWINDOW)
```

See Also:

Run Application statement, **Run Menu Command statement**, **Run Program statement**

Run Menu Command statement

Purpose

Runs a MapInfo Professional menu command, as if the user had selected the menu item. Can also be used to select a button on a ButtonPad. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Run Menu Command { *command_code* | **ID** *command_ID* }

command_code is an integer code from MENU.DEF (such as M_FILE_NEW), representing a standard menu item or button.

command_ID is a number representing a custom menu item or button.

Description

To execute a standard MapInfo Professional menu command, include the *command_code* parameter. The value of this parameter must match one of the menu codes listed in MENU.DEF. For example, the following MapBasic statement executes MapInfo Professional's **File > New** command:

```
Run Menu Command M_FILE_NEW
```

To select a standard button from MapInfo's ButtonPads, specify that button's code (from MENU.DEF). For example, the following statement selects the Radius Search button:

```
Run Menu Command M_TOOLS_SEARCH_RADIUS
```

To select a custom button or menu command (for example, a button or a menu command created through a MapBasic program), use the **ID** clause.

For example, if your program creates a custom tool button by issuing a statement such as…

```
Alter ButtonPad ID 1 Add
   ToolButton
```

```
Calling sub_procedure_name
ID 23
Icon MI_ICON_CROSSHAIR
```

…then the custom button has an ID of 23. The following statement selects the button.

```
Run Menu Command ID 23
```

Using MapBasic, the **Run Menu Command** statement can execute the MapInfo Professional **Help > MapInfo Professional Tutorial on the Web…** command.

```
Run Menu Command M_HELP_MAPINFO_WWW_TUTORIAL
```

You can access **Query > Invert Selection** using the following MapBasic command:

```
Run Menu Command M_QUERY_INVERTSELECT.
```

Access Page settings in **Options > Preferences > Printer** by using the following syntax:

```
RUN MENU COMMAND M_EDIT_PREFERENCES_PRINTER
```

Or

```
RUN MENU COMMAND 217
' if running from MapBasic window
```

See Also:

Run Application statement, **Run Program statement**

Integrated Mapping Applications

Integrated Mapping applications cannot display the Layer Control as a window. However, Integrated Mapping applications can display the Layer Control as a modal dialog box, by using the Run Menu Command statement with command M_MAP_LAYER_CONTROL_DIALOG (801). For example:

```
Run Menu Command 801
```

Preferences Dialog Box

MapInfo Professional's Preferences dialog box is a special case. The Preferences dialog box contains several buttons, each of which displays another dialog box. You can use **Run Menu Command** statement to invoke individual sub-dialog boxes. For example, the following statement displays the Map Window Preferences sub-dialog box:

```
Run Menu Command M_EDIT_PREFERENCES_MAP.
```

Layer Control Window and Dialog Box

MapInfo Professional 10.0 and higher display the Layer Control as a window and not as a dialog box. As of MapBasic 10.0, MapBasic applications can display the Layer Control as either a window or as a dialog box by executing a Run Menu command statement:

- To display Layer Control as a window, use M_MAP_LAYER_CONTROL (or its value, 822):
  ```
  Run Menu Command M_MAP_LAYER_CONTROL
  ```

- To display Layer Control as a dialog box (with OK and Cancel buttons), use M_MAP_LAYER_CONTROL_DIALOG (or its value, 801):

```
Run Menu Command M_MAP_LAYER_CONTROL_DIALOG
```

Releases before MapInfo Professional 10.0 display the Layer Control as a dialog box, so MapBasic applications (MBX) written with MapBasic 9.5 or earlier assume that the Layer Control is a dialog box. To be backwards compatible, MapInfo Professional 10.0 and higher executes older MapBasic applications that requests the Layer Control, using a Layer Control dialog box.

To control whether your MapBasic application (MBX) displays Layer Control as a window or a dialog box:

- If you recompile your MBX in MapBasic 10.0 (using the updated MENU.DEF from MapBasic 10.0), then the new MBX displays Layer Control as a window, not as a dialog box. This is ideal for most situations, because MapInfo Professional 10.0 users expect Layer Control to display as a window.
- If you have recompiled your MBX in MapBasic 10.0, but you want to continue displaying Layer Control as a dialog box, update your Run Menu Command as follows:

```
Run Menu Command M_MAP_LAYER_CONTROL_DIALOG
```

About the Layer Control Dialog Box

The Layer Control dialog box has fewer features compared to the Layer Control window. The following occur with the Layer Control dialog and not the Layer Control window:

- The Move Up and Move Down buttons are disabled if there are groups in the map.
- When you right-click on a layer, there is no context menu. As a result, most Group layer operations are not available.
- You are unable to move theme layers.

Run Program statement

Purpose

Runs an executable program. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

`Run Program` *`program_spec`*

program_spec is a command string that specifies the name of the program to run, and may also specify command-line arguments.

Description

If the specified *program_spec* does not represent a Windows application, MapBasic invokes a DOS shell, and runs the specified DOS program from there. If the *program_spec* is the character string “COMMAND.COM”, MapBasic invokes the DOS shell without any other program. In this case, the user is able to issue DOS commands, and then type `Exit` to return to MapInfo. When you spawn a program through a **Run Program** statement, Windows continues to control the computer. While the spawned program is running, Windows may continue to run other background tasks—including your

MapBasic program. This multitasking environment could potentially create conflicts. Thus, the MapBasic statements which follow the **Run Program** statement must not make any assumptions about the status of the spawned program.

When issuing the **Run Program** statement, you should take precautions to avoid multitasking conflicts. One way to avoid such conflicts is to place the **Run Program** statement at the end of a sequence of events. For example, you could create a custom menu item which calls a handler sub procedure, and you could make the **Run Program** statement the final statement in the handler procedure.

Example

The following **Run Program** statement runs the Windows text editor, “Notepad,” and instructs Notepad to open the text file THINGS.2DO.

```
Run Program "notepad.exe things.2do"
```

The following statement issues a DOS command.

```
Run Program "command.com /c dir c:\mapinfo\ > C:\temp\dirlist.txt"
```

See Also:

Run Application statement, **Run Command statement**, **Run Menu Command statement**

Save File statement

Purpose

Copies a file. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Save File old_filespec As new_filespec [ Append ]
```

old_filespec is a string representing the name (and, optionally, the path) of an existing file; the file must not be open.

new_filespec is a string representing the name (and, optionally, the path) to which the file will be copied; the file must not be open.

Description

The **Save File** statement copies a file. The file must not already be open for input/output.

If you include the optional **Append** keyword, and if the file *new_filespec* already exists, the contents of the file *old_filespec* are appended to the end of the file *new_filespec*.

Do not use **Save File** to copy a file that is a component of an open table (for example, *filename*.tab, *filename*.map, etc.). To copy a table, use the **Commit Table…As statement**.

The **Save File** statement cannot copy a file to itself.

Example

```
Save File "settings.txt" As "settings.bak"
```

See Also:

Kill statement, **Rename File statement**

Save MWS statement

Purpose

This statement allows you to save the current workspace as an XML-based MWS file for use with MapXtreme applications. These MWS files can be shared across platforms in ways that workspaces cannot. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Save MWS Window ( window_id [ , window_id ... ] )
   Default default_window_id As filespec
```

window_id is an integer window identifier for a Map window.

default_window_id is an integer window identifier for the Map window to be recorded in the MWS as the default map.

Description

MapInfo Professional enables you to save the maps in your workspace to an XML format for use with MapXtreme applications. When saving a workspace to MWS format, only the map windows and legends are saved. All other windows are discarded as MapXtreme applications cannot read that information. Once your workspace is saved in this format, it can be opened with the Workspace Manager utility that is included in the MapXtreme installation or with an application developed using MapXtreme. The file is valid XML so can also be viewed using any XML viewer or editor. MWS files created with MapInfo Professional 7.8 or later can be validated using schemas supplied with MapXtreme.

You will not be able to read files saved in MWS format in MapInfo Professional 7.8 or later.

In MapInfo Professional, you can set the visibility of a modifier theme without regard to its reference feature layer, so you can turn the visibility of the main reference layer off but still display the theme. In MapXtreme, the modifier themes (Dot Density, Ranges, Individual Value) are only drawn if the reference feature layer is visible. To ensure that modifiers marked as visible in MapInfo Professional display in tools like Workspace Manager, we force the visibility of the reference feature layer so that its modifier themes display.

It is important to note that many MapBasic statements and functions do not translate to MWS format. The sections below show what aspects of our maps can and cannot be saved into an MWS file. For detailed listing of the compatibilities between MapBasic and MISQL see the *MapInfo Professional User Guide*.

What is Saved in the MWS

The following information is included in the MWS workspace file:

- Tab files' name and alias;
- Coordinate system information;
- Map center and zoom settings;
- Layer list with implied order;
- Map size as pixel width and height;
- Map resize method;
- Style overrides;
- Raster layer overrides;
- Automatic labels;
- Custom labels;
- Queries referenced by map windows;
- Individual value themes;
- Dot density themes;
- Graduated symbol themes;
- Bar themes;
- Range themes;

- Pie themes;
- Grid themes as MapXtreme grid layers with a style override;
- Themes and label expressions based upon a single attribute column;
- Zoom-ranged overrides.

What is Not Saved to the MWS

The following information is not saved in the MWS workspace file:

- Any non-map windows (browsers, charts, redistricters, 3D map windows, Prism maps);
- Distance, area, or XY and military grid units;
- Snap mode, autoscroll, and smart pan settings;
- Printer setup information;
- Any table that is based on a query that is not referenced by a window;
- Any theme that is based upon computed columns, or based on an expression that cannot be translated from MapBasic syntax to MI SQL syntax;
- Labels based on expressions that cannot be translated from MapBasic syntax to MI SQL syntax;
- Queries with “sub-select” statements;
- Layers based on queries that includes “sub-select” statements;

> ⓘ A “sub-select” statement is any **Select** statement nested inside another **Select** statement.

- Export options;
- Hot links for labels and objects;
- Group layers;
- Whether object nodes, centroids or line direction is displayed.

See Also:

Save Workspace statement

Save Window statement

Purpose

Saves an image of a window to a file; corresponds to choosing **File > Save Window As**. This statement is used to save a Map window in raster and vector image formats. MapBasic supports raster image translucency. As of version 10.0 and later MapBasic supports translucency for vector images and the EMF+ and EMF+Dual image formats.

Supported vector formats are WMF, EMF, EMF+ and EMF+Dual. WMF and EMF are based on the same older technology used for non-enhanced windows. They display translucent vector maps, but they will appeared dithered, not as a true translucent image. EMF+, using enhanced rendering technology, will display translucent maps very well. EMF+Dual is a file that contains both an EMF and an EMF+ image.

Many older applications cannot read EMF+. The application tries to open it as an EMF (because the extension is EMF), and fails. EMF+Dual format is a compromise; older applications can open it as an EMF while newer applications can open it as an EMF+. For example, Office 2000 applications can read EMF, but not EMF+. Office 2007 reads EMF+.By saving the windows as EMF+Dual, both applications can read the same image.

MapInfo Professional reads all supported image formats with the exception of EMF+. All images display as raster images (including WMF and EMF).

Syntax

```
Save Window window_id
   As filespec
   Type filetype
   [ Width image_width [ Units paper_units ] ]
   [ Height image_height [ Units paper_units ] ]
   [ Resolution output_dpi ]
   [ Copyright notice [ Font... ] ]
```

window_id is an integer Window ID representing a Map, Layout, Graph, Legend, Statistics, Info, or Ruler window; to obtain a window ID, call a function such as the **FrontWindow( ) function** or the **WindowID( ) function**.

filespec is a string representing the name of the file to create.

filetype is a string representing a file format:

- "BMP" that specifies Bitmap format
- "WMF" that specifies Windows Metafile format
- "JPEG" that specifies JPEG format
- "JP2" that specifies JPEG 2000 format
- "PNG" that specifies Portable Network Graphics format
- "TIFF" that specifies TIFF format
- "TIFFCMYK" that specifies TIFF CMYK format
- "TIFFG4" that specifies TIFFG4 format
- "TIFFLZW" that specifies TIFFLZW format
- “GEOTIFF” that specifies georeferenced TIFF format
- "GIF" that specifies GIF format
- "PSD" that specifies Photoshop 3.0 format
- "EMF" that specifies Windows Enhanced Metafile format
- "EMF+" that specifies Windows EMF+ format
- "EMF+DUAL" that specifies a file format containing both EMF and EMF+ formats in a single file

image_width is a number that specifies the desired image width.

image_height is a number that specifies the desired image height.

paper_units is a string representing a paper unit name (for example, “cm” for centimeters).

output_dpi is a number that specifies the output resolution in DPI (dots per inch).

notice is a string that represents a copyright notice; it will appear at the bottom of the image.

The **Font** clause specifies a text style.

Description

The **Save Window** statement saves an image of a window to a file. The effect is comparable to the user choosing **File > Save Window As**, except that the **Save Window** statement does not display a dialog box. For Map, Layout, or Graph windows, the default image size is the size of the original window. For Legend, Statistics, Info, or Ruler windows, the default size is the size needed to represent all of the data in the window. Use the optional **Width** and **Height** clauses to specify a non-default image size. Resolution allows you to specify the dpi when exporting images to raster formats. The **Font clause** specifies a text style in the copyright notice.

To include a copyright notice on the bottom of the image, use the optional **Copyright** clause. See the example below. To eliminate the default notice, specify a **Copyright** clause with an empty string ("").

Error number 408 is generated if the export fails due to lack of memory or disk space. Note that specifying very large image sizes increases the likelihood of this error.

Examples

This example produces a Windows metafile:

```
Save Window i_mapper_ID As "riskmap.wmf" Type "WMF"
```

This example shows how to specify a copyright notice. The **Chr$( ) function** is used to insert the copyright symbol.

```
Save Window i_mapper_ID As "riskmap.bmp"
   Type "BMP"
   Copyright "Copyright " + Chr$(169) + " 1996, Pitney Bowes Software Inc.
Corp."
```

See Also:

Export statement

Save Workspace statement

Purpose

Creates a workspace file representing the current MapInfo Professional session. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Save Workspace As *filespec*

filespec is a string representing the name of the workspace file to create.

Description

The **Save Workspace** statement creates a workspace file that represents the current MapInfo Professional session. The effect is comparable to the user choosing **File > Save Workspace**, except that the **Save Workspace** statement does not display a dialog box.

To load an existing workspace file, use the **Run Application statement**.

Example

```
Save Workspace As "market.wor"
```

See Also:

Run Application statement

SearchInfo( ) function

Purpose

Returns information about the search results produced by SearchPoint( ) or SearchRect( ). You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

SearchInfo(*sequence_number, attribute*)

sequence_number is an integer number, from 1 to the number of objects located.

attribute is a small integer code from the table below.

Return Value

String or integer, depending on *attribute*.

Description

After you call SearchRect( ) or SearchPoint( ) to search for map objects, call SearchInfo( ) to process the search results.

The sequence_number argument is an integer number, 1 or larger. The number returned by SearchPoint( ) or SearchRect( ) is the maximum value for the sequence_number.

The attribute argument must be one of the codes (from MAPBASIC.DEF) in the following table:

attribute code	ID	SearchInfo( ) returns:
SEARCH_INFO_TABLE	1	String value: the name of the table containing this object. If an object is from a Cosmetic layer, this string has the form "Cosmetic*N*" (where *N* is a number, 1 or larger).
SEARCH_INFO_ROW	2	Integer value: this row's rowID number. You can use this rowID number in a **Fetch statement** or in a **Select statement**'s **Where** clause.

Search results remain in memory until the application halts or until you perform another search. Note that search results remain in memory even after the user closes the window or the tables associated with the search; therefore, you should process search results immediately. To manually free the memory used by search results, perform a search which you know will fail (for example, search at location 0, 0).

MapInfo Professional maintains a separate set of search results for each MapBasic application that is running, plus another set of search results for MapInfo Professional itself (for commands entered through the MapBasic window).

Error Conditions

ERR_FCN_ARG_RANGE (644) error is generated if *sequence_number* is larger than the number of objects located.

Example

The following program creates two custom tool buttons. If the user uses the point tool, this program calls the **SearchPoint( ) function**; if the user uses the rectangle tool, the program calls the **SearchRect( ) function**. In either case, this program calls **SearchInfo( )** to determine which object(s) the user chose.

```
Include "mapbasic.def"
Include "icons.def"
Declare Sub Main
Declare Sub tool_sub

Sub Main
   Create ButtonPad "Searcher" As
      ToolButton Calling tool_sub ID 1
         Icon MI_ICON_ARROW
         Cursor MI_CURSOR_ARROW
         DrawMode DM_CUSTOM_POINT
         HelpMsg "Click on a map location\nClick a location"
      Separator
         ToolButton Calling tool_sub ID 2
         Icon MI_ICON_SEARCH_RECT
         Cursor MI_CURSOR_FINGER_LEFT
         DrawMode DM_CUSTOM_RECT
         HelpMsg "Drag a rectangle in a map\nDrag a rectangle"
```

```
      Width 3

   Print "Searcher program now running."
   Print "Choose a tool from the Searcher toolbar"
   Print "and click on a map."
End Sub
Sub tool_sub
   ' This procedure is called whenever the user uses
   ' one of the custom buttons on the Searcher toolbar.
   Dim x, y, x2, y2 As Float,
      i, i_found, i_row_id, i_win_id As Integer,
      s_table As Alias
   i_win_id = FrontWindow( )
   If WindowInfo(i_win_id, WIN_INFO_TYPE) <> WIN_MAPPER Then
      Note "This tool only works on Map windows."
      Exit Sub
   End If
   ' Determine the starting point where the user clicked.
   x = CommandInfo(CMD_INFO_X)
   y = CommandInfo(CMD_INFO_Y)
   If CommandInfo(CMD_INFO_TOOLBTN) = 1 Then
      ' Then the user is using the point-mode tool.
      ' determine how many objects are at the chosen point.
      i_found = SearchPoint(i_win_id, x, y)
   Else
      ' The user is using the rectangle-mode tool.
      ' Determine what objects are within the rectangle.
      x2 = CommandInfo(CMD_INFO_X2)
      y2 = CommandInfo(CMD_INFO_y2)
      i_found = SearchRect(i_win_id, x, y, x2, y2)
   End If

   If i_found = 0 Then
      Beep ' No objects found where the user clicked.
   Else
      Print Chr$(12)
      If CommandInfo(CMD_INFO_TOOLBTN) = 2 Then
         Print "Rectangle: x1= " + x + ", y1= " + y
         Print "x2= " + x2 + ", y2= " + y2
   Else
      Print "Point: x=" + x + ", y= " + y
   End If

   ' Process the search results.
   For i = 1 to i_found
      ' Get the name of the table containing a "hit".
      s_table = SearchInfo(i, SEARCH_INFO_TABLE)

      ' Get the row ID number of the object that was a hit.
      i_row_id = SearchInfo(i, SEARCH_INFO_ROW)

      If Left$(s_table, 8) = "Cosmetic" Then
```

```
         Print "Object in Cosmetic layer"
      Else
         ' Fetch the row of the object the user clicked on.
         Fetch rec i_row_id From s_table
         s_table = s_table + ".col1"
         Print s_table
      End If
   Next
   End If
End Sub
```

See Also:

SearchPoint( ) function, **SearchRect( ) function**

SearchPoint( ) function

Purpose

Searches for map objects at a specific x/y location. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

SearchPoint(*map_window_id*, *x*, *y*)

map_window_id is a Map window's integer ID number

x is an x-coordinate (for example, longitude)

y is a y-coordinate (for example, latitude)

Return Value

Integer, representing the number of objects found.

Description

The **SearchPoint( )** function searches for map objects at a specific x/y location. The search applies to all selectable layers in the Map window, even the Cosmetic layer (if it is currently selectable). The return value indicates the number of objects found.

This function does not select any objects, nor does it affect the current selection. Instead, this function builds a list of objects in memory. After calling **SearchPoint( )**, call the **SearchInfo( ) function** to process the search results.

The search allows for a small tolerance, identical to the tolerance allowed by MapInfo Professional's Info tool. Points or linear objects that are very close to the location are included in the search results, even if the user did not click on the exact location of the object.

To allow the user to select an x/y location with the mouse, use the **Create ButtonPad statement** or the **Alter ButtonPad statement** to create a custom ToolButton. Use DM_CUSTOM_POINT as the button's draw mode. Within the button's handler procedure, call the **CommandInfo( ) function** to determine the x/y coordinates.

Example

For a code example, see the **SearchInfo( ) function**.

See Also:

SearchInfo( ) function, **SearchRect( ) function**

SearchRect( ) function

Purpose

Searches for map objects within a rectangular area. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

SearchRect(*map_window_id, x1, y1, x2, y2*)

map_window_id is a Map window's integer ID number.

x1, *y1* are coordinates that specify one corner of a rectangle.

x2, *y2* are coordinates that specify the opposite corner of a rectangle.

Return Value

Integer, representing the number of objects found.

Description

The **SearchRect( )** function searches for map objects within a rectangular area. The search applies to all selectable layers in the Map window, even the Cosmetic layer (if it is currently selectable). The return value indicates the number of objects found.

This function does not select any objects, nor does it affect the current selection. Instead, this function builds a list of objects in memory. After calling **SearchRect( )** you call **SearchInfo( ) function** to process the search results.

The search behavior matches the behavior of MapInfo Professional's Marquee Select button: If an object's centroid falls within the rectangle, the object is included in the search results.

To allow the user to select a rectangular area with the mouse, use the **Create ButtonPad statement**or the **Alter Button statement** to create a custom ToolButton. Use DM_CUSTOM_RECT as the button's draw mode. Within the button's handler procedure, call **CommandInfo( ) function** to determine the x/y coordinates.

Example

For a code example, see the **SearchInfo( ) function**.

See Also:

SearchInfo( ) function, **SearchPoint( ) function**

Second function

Purpose

Returns the second and millisecond component of a Time as a floating-point number. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

```
Second (Time)
```

Return Value

Number

Example

Copy this example into the MapBasic window for a demonstration of this function.

```
dim X as time
dim iSec as integer
X = CurDateTime()
Sec = Second(X)
Print iSec
```

Seek( ) function

Purpose

Returns the current file position.

Syntax

Seek(*filenum*)

filenum is the number of an open file.

Return Value

Integer

Description

The **Seek( )** function returns MapBasic's current position in an open file.

The *filenum* parameter represents the number of an open file; this is the same number specified in the **As** clause of the **Open File statement**.

The integer value returned by the **Seek( )** function represents a file position. If the file was opened in random-access mode, **Seek( )** returns a record number (the next record to be read or written). If the file was opened in binary mode, **Seek( )** returns the byte position of the next byte to be read from or written to the file.

Error Conditions

ERR_FILEMGR_NOTOPEN (366) error is generated if the specified file is not open.

See Also:

Get statement, **Open File statement**, **Put statement**, **Seek statement**

Seek statement

Purpose

Sets the current file position, to prepare for the next file input/output operation.

Syntax

Seek [**#**]*filenum*, *position*

filenum is an integer value, indicating the number of an open file.

position is an integer value, indicating the desired file position.

Description

The **Seek** statement resets the current file position of an open file. File input/output operations which follow a **Seek** statement will read from (or write to) the location specified by the **Seek**.

If the file was opened in Random access mode, the *position* parameter specifies a record number.

If the file was opened in a sequential access mode, the *position* parameter specifies a specific byte position; a position value of one represents the very beginning of the file.

See Also:

Get statement, **Input # statement**, **Open File statement**, **Print # statement**, **Put statement**, **Seek( ) function**, **Write # statement**

SelChangedHandler procedure

Purpose

A reserved procedure, called automatically when the set of selected rows changes.

Syntax

```
Declare Sub SelChangedHandler

Sub SelChangedHandler
    statement_list
End Sub
```

statement_list is a list of statements to execute when the set of selected rows changes.

Description

SelChangedHandler is a special MapBasic procedure name. If the user runs an application with a procedure named **SelChangedHandler**, the application "goes to sleep" when the Main procedure runs out of statements to execute. The sleeping application remains in memory until the application executes an **End Program statement**. As long as the application remains in memory, MapInfo Professional automatically calls the **SelChangedHandler** procedure whenever the set of selected rows changes.

Within the **SelChangedHandler** procedure, you can obtain information about recent changes made to the selection by calling **CommandInfo( ) function** with one of the following codes:

attribute code	ID	CommandInfo(attribute) returns:
CMD_INFO_SELTYPE	1	1 if one row was added to the selection; 2 if one row was removed from the selection; 3 if multiple rows were added to the selection; 4 if multiple rows were de-selected.
CMD_INFO_ROWID	2	Integer value: The number of the row which was selected or de-selected (only applies if a single row was selected or de-selected).
CMD_INFO_INTERRUPT	3	Logical value: TRUE if the user interrupted a selection process by pressing Esc; FALSE otherwise.

When any procedure in an application executes the **End Program statement**, the application is completely removed from memory. Thus, you can use the **End Program statement** to terminate a **SelChangedHandler** procedure once it is no longer wanted. Be careful not to issue an **End Program statement** while the **SelChangedHandler** procedure is still needed.

Multiple MapBasic applications can be "sleeping" at the same time. When the Selection table changes, MapBasic automatically calls all sleeping **SelChangedHandler** procedures, one after another.

A **SelChangedHandler** procedure should not take actions that affect the GUI "focus" or reset the current window. In other words, the **SelChangedHandler** procedure should not issue statements such as a **Note statement**, **Print statement**, or **Dialog statement**.

See Also:

CommandInfo( ) function, **SelectionInfo( ) function**

Select statement

Purpose

Selects particular rows and columns from one or more open tables, and treats the results as a separate, temporary table. Also provides the ability to sort and sub-total data. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Select expression_list
   From table_name [ , ... ] [ Where expression_group ]
      [ Into results_table [ Noselect ] ]
      [ Group By column_list ]
      [ Order By column_list ]
```

expression_list is a comma-separated list of expressions which will comprise the columns of the Selection results.

expression_group is a list of one or more expressions, separated by the keywords AND or OR.

table_name is the name of an open table.

results_table is the name of the table where query results should be stored.

column_list is a list of one or more names of columns, separated by commas.

Description

The **Select** statement provides MapBasic programmers with the capabilities of MapInfo Professional's **Query > SQL Select** dialog box.

The MapBasic **Select** statement is modeled after the Select statement in the Structured Query Language (SQL). Thus, if you have used SQL-oriented database software, you may already be familiar with the Select statement. Note, however, that MapBasic's **Select** statement includes geographic capabilities that you will not find in other packages.

Column expressions (for example, *tablename.columnname*) in a **Select** statement may only refer to tables that are listed in the **Select** statement's From clause. For example, a **Select** statement may only incorporate the column expression STATES.OBJ if the table STATES is included in the statement's **From** clause.

The **Select** statement serves a variety of different purposes. One **Select** statement might apply a test to a table, making it easy to browse only the records which met the criteria (this is sometimes referred to as filtering). Alternately, **Select** might be used to calculate totals or subtotals for an entire table. **Select** can also: sort the rows of a table; derive new column values from one or more existing columns; or combine columns from two or more tables into a single results table.

Generally speaking, a **Select** statement queries one or more open tables, and selects some or all of the rows from said table(s). The **Select** statement then treats the group of selected rows as a results table; Selection is the default name of this table (although the results table can be assigned another name through the Into clause). Following a **Select** statement, a MapBasic program—or, for that matter, a MapInfo Professional user—can treat the results table as any other MapInfo table.

After issuing a **Select** statement, a MapBasic program can use the **SelectionInfo( ) function** to examine the current selection.

The **Select** statement format includes several clauses, most of which are optional. The nature and function of a **Select** statement depend upon which clauses are included. For example: if you wish to use a **Select** statement to set up a filter, you should include a **Where** clause; if you wish to use a **Select** statement to subtotal the values in the table, you should include a **Group By** clause; if you want MapBasic to sort the results of the **Select** statement, you should include an **Order By** clause. Note that these clauses are not mutually exclusive; one **Select** statement may include all of the optional clauses.

Select clause

This clause dictates which columns MapBasic should include in the results table. The simplest type of *expression_list* is an asterisk character ("`*`"). The asterisk signifies that all columns should be included in the results. The statement:

```
Select * From world
```

tells MapBasic to include all of the columns from the "world" table in the results table. Alternately, the *expression_list* clause can consist of a list of expressions, separated by commas, each of which represents one column to include in the results table. Typically, each of these expressions involves the names of one or more columns from the table in question. Very often, MapBasic function calls and/or operators are used to derive some new value from one or more of the column names.

For example, the following **Select** statement specifies an *expression_list* clause with two expressions:

```
Select country, Round(population,1000000)
   From world
```

The *expression_list* above consists of two expressions, the first of which is a simple column name (`country`), and the second of which is a function call (**Round( )**) which operates on another column (`population`).

After MapBasic carries out the above **Select** statement, the first column in the results table will contain values from the world table's name column. The second column in the results table will contain values from the world table's population column, rounded off to the nearest million.

Each expression in the *expression_list* clause can be explicitly named by having an alias follow the expression; this alias would appear, for example, at the top of a Browser window displaying the appropriate table. The following statement would assign the field alias "Millions" to the second column of the results table:

```
Select country,Round(population,1000000) "Millions"
   From world
```

Any mappable table also has a special column, called object (or obj for short). If you include the column expression obj in the *expression_list*, the resultant table will include a column which indicates what type of object (if any) is attached to that row.

The *expression_list* may include either an asterisk or a list of column expressions, but not both. If an asterisk appears following the keyword **Select**, then that asterisk must be the only thing in the *expression_list*. In other words, the following statement would not be legitimate:

```
Select *, object From world ' this won't work!
```

From clause

The **From** clause specifies which table(s) to select data from. If you are doing a multiple-table join, the tables you are selecting from must be base tables, rather than the results of a previous query.

Where clause

One function of the **Where** clause is to specify which rows to select. Any expression can be used (see Expressions section below). Note, however, that groups of two or more expressions must be connected by the keywords And or Or, rather than being comma-separated. For example, a two-expression **Where** clause might read like this:

```
Where Income > 15000 And Income < 25000
```

Note that the And operator makes the clause more restrictive (both conditions must evaluate as TRUE for MapBasic to select a record), whereas the Or operator makes the clause less restrictive (MapBasic will select a record if either of the expressions evaluates to TRUE).

By referring to the special column name object, a **Where** clause can test geographic aspects of each row in a mappable table. Conversely, the expression “Not object” can be used to single out records which do not have graphical objects attached.

For example, the following **Where** clause would tell MapBasic to select only those records which are currently un-geocoded:

```
Where Not Object
```

If a **Select** statement is to use two or more tables, the statement must include a **Where** clause, and the **Where** clause must include an expression which tells MapBasic how to join the two tables. Such a join-related expression typically takes the form **Where** *tablename1.field = tablename2.field*, where the two fields have corresponding values. The following example shows how you might join the tables “States” and “City_1k.” The column City_1k.state contains two-letter state abbreviations which match the abbreviations in the column States.state.

```
Where States.state = City_1k.state
```

Alternately, you can specify a geographic operator to tell MapInfo Professional how to join the two tables.

```
Where states.obj Contains City_1k.obj
```

A **Where** clause can incorporate a subset of specific values by including the **Any** or **All** keyword. The **Any** keyword defines a subset, for the sake of allowing the **Where** clause to test if a given expression is TRUE for any of the values in the subset. Conversely, the **All** keyword defines a subset, for the sake of allowing the **Where** clause to test if a given condition is true for all of the values in the subset.

The following query selects any customer record whose state column contains “NY,” “MA,” or “PA.” The Any( ) function functions the same way as the SQL “IN” operator.

```
Select * From customers
   Where state = Any ("NY", "MA", "PA")
```

A **Where** clause can also include its own **Select** statement, to produce what is known as a subquery. In the next example, we use two tables: “products” is a table of the various products which our company sells, and “orders” is a table of the orders we have for our products. At any given time, some of the products may be sold out. The task here is to figure out which orders we can fill, based on which products are currently in stock. This query uses the logic, “select all orders which are not among the list of items that are currently sold out.”

```
Select * From orders
   Where partnum <>
   All(Select partnum from products
      where not instock)
```

On the second line of the query, the keyword **Select** appears a second time; this produces our sub-select. The sub-select builds a list of the parts that are currently not in stock. The **Where** clause of the main query then uses All( ) function to access the list of unavailable parts.

In the example above, the sub-select produces a set of values, and the main **Select** statement's **Where** clause tests for inclusion in that set of values. Alternately, a sub-select might use an aggregate operator to produce a single result.

The example below uses the Avg( ) aggregate operator to calculate the average value of the pop field within the table states.

Accordingly, the net result of the following **Select** statement is that all records having higher-than-average population are selected.

```
Select * From states
   Where population >
      (Select Avg(population) From states)
```

MapInfo Professional also supports the SQL keyword In. A **Select** statement can use the keyword In in place of the operator sequence = Any. In other words, the following **Where** clause, which uses the **Any** keyword:

```
Where state = Any ("NY", "MA", "PA")
```

is equivalent to the following **Where** clause, which uses the **In** keyword:

```
Where state In ("NY", "MA", "PA")
```

In a similar fashion, the keywords **Not** In may be used in place of the operator sequence: <> All.

A single **Select** statement may not include multiple, non-nested subqueries. Additionally, MapBasic's **Select** statement does not support “correlated subqueries.” A correlated subquery involves the inner query referencing a variable from the outer query. Thus, the inner query is reprocessed for each row in the outer table. Thus, the queries are correlated. An example:

```
' Note: the following statement, which illustrates
' correlated subqueries, will NOT work in MapBasic

Select * from leads
Where lead.name =
```

```
(Select var.name From vars
   Where lead.name = customer.name)
```

This limitation is primarily of interest to users who are already proficient in SQL queries, through the use of other SQL-compatible database packages.

Into clause

This optional clause lets you name the results table. If no **Into** clause is specified, the resulting table is named Selection. Note that when a subsequent operation references the Selection table, MapInfo Professional will take a "snapshot" of the Selection table, and call the snapshot QUERY*n* (for example, `QUERY1`).

If you include the **Noselect** keyword, the statement performs a query without changing the pre-existing Selection table. Use the **NoSelect** keyword if you need to perform a query, but you do not want to de-select whatever rows are already selected.

If you include the **Noselect** keyword, the query does not trigger the **SelChangedHandler procedure**.

Group By clause

This optional clause specifies how to group the rows when performing aggregate functions (sub-totalling). In a **Group By** clause, you typically specify a column name (or a list of column names); MapBasic then builds a results table containing subtotals. For example, if you want to subtotal your table on a state-by-state basis, your **Group By** clause should specify the name of a column which contains state names. The **Group By** clause may not reference a function with a variable return type, such as the **ObjectInfo( ) function**.

The aggregate functions **Sum( )**, **Min( )**, **Max( )**, **Count(*)**, **Avg( )**, and **WtAvg( )** allow you to calculate aggregated results.

These aggregate functions do not appear in the **Group By** clause. Typically, the **Select** *expression_list* clause includes one or more of the aggregate functions listed above, while the **Group By** clause indicates which column(s) to use in grouping the rows.

Suppose the Q4Sales table describes sales information for the fourth fiscal quarter. Each record in this table contains information about the dollar amount of a particular sale. Each record's Territory column indicates the name of the territory where the sale occurred. The following query counts how many sales occurred within each territory, and calculates the sum total of all of the sales within each territory.

```
Select territory, Count(*), Sum(amount)
   From q4sales
   Group By territory
```

The **Group By** clause tells MapBasic to group the table results according to the contents of the Territory column, and then create a subtotal for each unique territory name. The expression list following the keyword **Select** specifies that the results table should have three columns: the first

column will state the name of a territory; the second column will state the number of records in the q4sales table “belonging to” that territory; and the third column of the results table will contain the sum of the Amount columns of all records belonging to that territory.

The **Sum( )** function requires a parameter, to tell it which column to summarize. The **Count( )** function, however, simply takes an asterisk as its parameter; this tells MapBasic to simply count the number of records within that sub-totalled group. The **Count( )** function is the only aggregate function that does not require a column identifier as its parameter.

The following table describes MapInfo Professional's aggregate functions.

Function name	Description	Returns
Avg(*column*)	Returns the average value of the specified column.	float
Count(*)	Returns the number of rows in the group. Specify * (asterisk) instead of column name.	integer
Max(*column*)	Returns the largest value of the specified column for all rows in the group.	float
Min(*column*)	Returns the smallest value of the specified column for all rows in the group.	float
Sum(*column*)	Returns the sum of the column values for all rows in the group.	float
WtAvg(*column* , *weight_column*)	Returns the average of the column values, weighted. See below.	float

No MapBasic function, aggregate or otherwise, returns a decimal value. A decimal field is only a way of storing the data. The arithmetic is done with floating point numbers.

Calculating Weighted Averages

Use the **Wtavg( )** aggregate function to calculate weighted averages. For example, the following statement uses the **Wtavg( )** function to calculate a weighted average of the literacy rate in each continent:

```
Select continent, Sum(pop_1994), WtAvg(literacy, Pop_1994)
   From World
   Group By continent
   Into Lit_query
```

Because of the **Group By** clause, MapInfo Professional groups rows of the table together, according to the values in the Continent column. All rows having “North America” in the Continent column will be treated as one group; all rows having “Asia” in the Continent column will be treated as another group; etc. For each group of rows—in other words, for each continent—MapInfo Professional calculates a weighted average of the literacy rates.

A simple average (using the **Avg( )** function) calculates the sum divided by the count. A weighted average (using the **WtAvg( )** function) is more complicated, in that some rows affect the average more than other rows. In this example, the average calculation is weighted by the Pop_1994 (population) column; in other words, countries that have a large population will have more of an impact on the result than countries that have a small population.

Column Expressions in the Group By clause

In the preceding example, the **Group By** territory clause identifies the Territory column by name. Alternately, a **Group By** clause can identify a column by a number, using an expression of the form *col#*. In this type of expression, the # sign represents an integer number, having a value of one or more, which identifies one of the columns in the **Select** clause. Thus, the above **Select** statement could have read `Group By col1`, or even `Group By 1`, rather than **Group By** territory.

It is sometimes necessary to use one of these alternate syntaxes. If you wish to Group By a derived expression, which does not have a column name, then the **Group By** clause must use the *col#* syntax or the # syntax to refer to the proper column expression. In the following example, we Group By a column value derived through the **Month( )** function. Since this column expression does not have a conventional column name, our **Group By** clause refers to it using the *col#* format:

```
Select Month(sick_date), Count(*)
   From sickdays
   Group By 1
```

This example assumes that each row in the sickdays table represents a sick day claim. The results from this query would include twelve rows (one row for each month); the second column would indicate how many sick days were claimed for that month.

Grouping By Multiple Columns

Depending on your application, you may need to specify more than one column in the **Group By** clause; this happens when the contents of a column are not sufficiently unique. For example, you may have a table describing counties across the United States. County names are not unique; for example, many different states have a Franklin county. Therefore, if your **Group By** clause specifies a single county-name column, MapBasic will create one sub-total row in the results table for the county “Franklin”. That row would summarize all counties having the name “Franklin”, regardless of whether the records were in different states.

When this type of problem occurs, your **Group By** clause must specify two or more columns, separated by commas. For example, a group by clause might read:

```
   Group By county, state
```

With this arrangement, MapBasic would construct a separate group of rows (and, thus, a separate sub-total) for each unique expression of the form *countyname*, *statename*. The results table would have separate rows for Franklin County, MA versus Franklin County, FL.

Order By clause

This optional clause specifies which column or set of columns to order the results by. As with the **Group By** clause, the column is specified by name in the field list, or by a number representing the position in the field list. Multiple columns are separated by commas.

By default, results sorted by an **Order By** clause are in ascending order. An ascending character sort places “A” values before “Z” values; an ascending numeric sort places small numbers before large ones. If you want one of the columns to be sorted in descending order, you should follow that column name with the keyword **DESC**.

```
Select * From cities
   Order By state, population Desc
```

This query performs a two-level sort on the table Cities. First, MapBasic sorts the table, in ascending order, according to the contents of the state column. Then MapBasic sorts each state's group of records, using a descending order sort of the values in the population column. Note that there is a space, not a comma, between the column name and the keyword **DESC**.

The **Order By** clause may not reference a function with a variable return type, such as the **ObjectInfo( ) function**.

Geographic Operators

MapBasic supports several geographic operators: Contains, Contains Part, Contains Entire, Within, Partly Within, Entirely Within, and Intersects. These operators can be used in any expression, and are very useful within the **Select** statement's **Where** clause. All geographic operators are infix operators (operate on two objects and return a boolean). The operators are listed in the table below.

Usage	Evaluates TRUE if:
objectA Contains objectB	first object contains the centroid of second object
objectA Contains Part objectB	first object contains part of second object
objectA Contains Entire objectB	first object contains all of second object
objectA Within objectB	first object's centroid is within the second object
objectA Partly Within objectB	part of the first object is within the second object
objectA Entirely Within objectB	the first object is entirely inside the second object
objectA Intersects objectB	the two objects intersect at some point

Selection Performance

Some **Select** statements are considerably faster than others, depending in part on the contents of the **Where** clause.

If the **Where** clause contains one expression of the form:

columnname = constant_expression

or if the *Where* clause contains two or more expressions of that form, joined by the And operator, then the **Select** statement will be able to take maximum advantage of indexing, allowing the operation to proceed quickly. However, if multiple *Where* clause expressions are joined by the Or operator instead of by the And operator, the statement will take more time, because MapInfo Professional will not be able to take maximum advantage of indexing.

Similarly, MapInfo Professional provides optimized performance for *Where* clause expressions of the form:

```
[ tablename. ] obj geographic_operator object_expression
```

and for **Where** clause expressions of the form:

```
RowID = constant_expression
```

RowID is a special column name. Each row's RowID value represents the corresponding row number within the appropriate table; in other words, the first row in a table has a RowID value of one.

Examples

This example selects all customers that are in New York, Connecticut, or Massachusetts. Each customer record does not need to include a state name; rather, the query relies on the geographic position of each customer object to determine whether that customer is “in” a given state.

```
Select * From customers
   Where obj Within Any(Select obj From states
      Where state = "NY" or state = "CT" or state = "MA")
```

The next example demonstrates a sub-select. Here, we want to select all sales territories which contain customers that have been designated as “Federal.” The subselect selects all customer records flagged as Federal, and then the main select works from the list of Federal customers to select certain territories.

```
Select * From territories
   Where obj Contains Any (Select obj From customers
      Where customers.source = "Federal")
```

The following query selects all parcels that touch parcel 120059.

```
Select * From parcels
   Where obj Intersects (Select obj From parcels
      Where parcel_id = 120059)
```

See Also:

Open Table statement

SelectionInfo( ) function

Purpose

Returns information about the current selection. You can call this function from the MapBasic Window in MapInfo Professional.

> ⓘ Selected labels do not count as a “selection,” because labels are not complete objects, they are attributes of other objects.

Syntax

SelectionInfo(*attribute*)

attribute is an integer code from the table below.

Return Value

String or integer; see table below.

Description

The table below summarizes the codes (from MAPBASIC.DEF) that you can use as the attribute parameter.

attribute setting	ID	SelectionInfo( ) Return Value
SEL_INFO_TABLENAME	1	String: The name of the table the selection was based on. Returns an empty string if no data currently selected.
SEL_INFO_SELNAME	2	String: The name of the temporary table (for example, “Query1”) representing the query. Returns an empty string if no data currently selected.
SEL_INFO_NROWS	3	Integer: The number of selected rows. Returns zero if no data currently selected.

> ⓘ If the current selection is the result of a join of two or more tables, **SelectionInfo**(SEL_INFO_NROWS) returns the number of rows selected in the base table, which might not equal the number of rows in the Selection table. See example below.

Error Conditions

ERR_FCN_ARG_RANGE (644) error is generated if an argument is outside of the valid range.

Example

The following example uses a **Select statement** to perform a join. Afterwards, the variable *i* contains 40 (the number of rows currently selected in the base table, States) and the variable *j* contains 125 (the number of rows in the query results table).

```
Dim i, j As Integer
Select * From States, City_125
   Where States.obj Contains City_125.obj Into QResults
i = SelectionInfo(SEL_INFO_NROWS)
j = TableInfo(QResults, TAB_INFO_NROWS)
```

See Also:

Select statement, **TableInfo( ) function**

Server Begin Transaction statement

Purpose

Requests a remote data server to begin a new unit of work. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Server *ConnectionNumber* **Begin Transaction**

ConnectionNumber is an integer value that identifies the specific connection.

Description

The **Server Begin Transaction** statement is used to mark a beginning point for transaction processing. The database does not save the results of subsequent SQL Insert, Delete, and Update statements issued via the **Server_Execute( ) function** until a **Server Commit statement** is issued. Use the **Server Rollback statement** to discard changes.

Example

```
Dim hdbc As Integer
hdbc = Server_Connect("ODBC", "DLG=1")
Server hdbc Begin Transaction
' ... other server statements ...
Server hdbc Commit
```

See Also:

Server Commit statement, **Server Rollback statement**

Server Bind Column statement

Purpose

Assigns local storage that can be used by the remote data server. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Server *StatementNumber* **Bind Column** *n* **To** *Variable*, *StatusVariable*

StatementNumber is an integer value that identifies information about a SQL statement.

n is a column number in the result set to bind.

Variable is a MapBasic variable to contain a column value following a fetch.

StatusVariable is an integer code indicating the status of the value as either null, truncated, or a positive integer value.

Description

The **Server Bind Column** statement sets up an application variable as storage for the result data of a column specified in a remote **Select statement**. When the subsequent **Server Fetch statement** retrieves a row of data from the server, the value for the column is stored in the variable specified by the **Server Bind Column statement**. The status of the column result is stored in the status variable.

StatusVariable value	ID	Condition
SRV_NULL_DATA	-1	Returned when the column has no data for that row.
SRV_TRUNCATED_DATA	-2	Returned when there is more data in the column than can be stored in the MapBasic variable.
Positive integer value		Number of bytes returned by the server.

Example

```
' Application to "print" address labels
' Assumes that a relational table ADDR exists with 6 columns...
Dim hdbc, hstmt As Integer
Dim first_name, last_name, street, city, state, zip As String
Dim fn_stat, ln_stat, str_stat, ct_stat, st_stat, zip_stat As Integer
hdbc = Server_Connect("ODBC", "DLG=1")
hstmt = Server_Execute( hdbc, "select * from ADDR")
Server hstmt Bind Column 1 To first_name,fn_stat
Server hstmt Bind Column 2 To last_name, ln_stat
Server hstmt Bind Column 3 To street, str_stat
Server hstmt Bind Column 4 To city, ct_stat
Server hstmt Bind Column 5 To state, st_stat
Server hstmt Bind Column 6 To zip, zip_stat
```

```
Server hstmt Fetch NEXT
While Not Server_Eot(hstmt)
   Print first_name + " " + last_name
   Print street
   Print city + ", " + state + " " + zip
   Server hstmt Fetch NEXT
Wend
Server hstmt Close
Server hdbc Disconnect
```

See Also:

Server_ColumnInfo( ) function

Server Close statement

Purpose

Frees resources associated with running a remote data access statement. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Server *StatementNumber* **Close**

StatementNumber is an integer value that identifies information about a SQL statement.

Description

The **Server Close** statement is used to inform the server that processing on the current remote statement is finished. All resources associated with the statement are returned. Remember to call the **Server Close statement** immediately after a **Server_Execute( ) function** for any non-query SQL statement you are finished processing.

Example

```
' Fetch the 5th record then close the statement
hstmt = Server_Execute(hdbc, "Select * from Massive_Database")
Server hstmt Fetch Rec 5
Server hstmt Close
```

See Also:

Server_Execute( ) function

Server_ColumnInfo( ) function

Purpose

Retrieves information about columns in a result set. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Server_ColumnInfo(*StatementNumber*, *ColumnNo*, *Attr*)

StatementNumber is an integer value that identifies information about an SQL statement.

ColumnNo is the number of the column in the table, starting at 1 with the leftmost column.

Attr is a code indicating which aspect of the column to return.

Return Value

The return value is conditional based on the value of the attribute passed (*Attr*).

Description

The **Server_ColumnInfo** function returns information about the current fetched column in the result set of a remote data source described by a remotely executed **Select statement**. The *StatementNumber* parameter specifies the particular statement handle associated with that connection. The *ColumnNo* parameter indicates the desired column (the columns are numbered from the left starting at 1). *Attr* selects the kind of information that will be returned.

The following table contains the attributes returned to the *Attr* parameter. These types are defined in MAPBASIC.DEF.

Attr value	ID	Server_ColumnInfo( ) returns:
SRV_COL_INFO_NAME	1	String result, the name identifying the column.
SRV_COL_INFO_TYPE	2	Integer result, a code indicating the column type: • SRV_COL_TYPE_NONE (0) • SRV_COL_TYPE_CHAR (1) • SRV_COL_TYPE_DECIMAL (2) • SRV_COL_TYPE_INTEGER (3) • SRV_COL_TYPE_SMALLINT (4) • SRV_COL_TYPE_DATE (5) • SRV_COL_TYPE_LOGICAL (6) • SRV_COL_TYPE_FLOAT (8) • SRV_COL_TYPE_FIXED_LEN_STRING (16) • SRV_COL_TYPE_BIN_STRING (17) See Server Fetch for how MapInfo Professional interprets data types.
SRV_COL_INFO_WIDTH	3	Integer result, indicating maximum number of characters in a column of type SRV_COL_TYPE_CHAR (1) or SRV_COL_TYPE_FIXED_LEN_STRING (16). When using ODBC the null terminator is not counted. The value returned is the same as the server database table column width.
SRV_COL_INFO_PRECISION	4	Integer result, indicating the total number of digits for a SRV_COL_TYPE_DECIMAL (2) column, or -1 for any other column type.
SRV_COL_INFO_SCALE	5	Integer result, indicating the number of digits to the right of the decimal for a SRV_COL_TYPE_DECIMAL (2) column, or -1 for any other column type.
SRV_COL_INFO_VALUE	6	Result type varies. Returns the actual data value from the column of the current row. Long character column values greater than 32,766 will be truncated. Binary column values are returned as a double length string of hexadecimal characters.

Attr value	ID	Server_ColumnInfo( ) returns:
SRV_COL_INFO_STATUS	7	Integer result, indicating the status of the column value: • SRV_NULL_DATA (-1) Returned when the column has no data for that row. • SRV_TRUNCATED_DATA (-2) Returned when there is more data in the column than can be stored in the MapBasic variable. • Positive integer value Number of bytes returned by the server.
SRV_COL_INFO_ALIAS	8	Column alias returned if an alias was used for the column in the query.

Example

```
Dim hdbc, Stmt As Integer
Dim Col As Integer
hdbc = Server_Connect("ODBC", "DLG=1")
Stmt = Server_Execute(hdbc, "Select * from emp")
Server Stmt Fetch NEXT
For Col = 1 To Server_NumCols(Stmt)
   Print Server_ColumnInfo(Stmt, Col, SRV_COL_INFO_NAME) +
   " = " +
   Server_ColumnInfo(Stmt, Col, SRV_COL_INFO_VALUE)
Next
```

See Also:

Server Bind Column statement, **Server Fetch statement**, **Server_NumCols( ) function**

Server Commit statement

Purpose

Causes the current unit of work to be saved to the database. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Server *ConnectionNumber* **Commit**

ConnectionNumber is an integer value that identifies the specific connection.

Description

The **Server Commit** statement makes permanent the effects of all remote SQL statements on the connection issued since the last **Server Begin Transaction statement** to the database. You must have an open transaction initiated by the **Server Begin Transaction statement** before you can use the **Server Commit** statement. Then you must issue a new **Server Begin Transaction statement** following the **Server Commit** statement to begin a new transaction.

Example

```
hdbc = Server_Connect("ODBC", "DLG=1")
Server hdbc Begin Transaction
hstmt = Server_Execute(hdbc, "Update Emp Set salary = salary * 1.5")
Server hdbc Commit
```

See Also:

Server Begin Transaction statement, **Server Rollback statement**

Server_Connect( ) function

Purpose

Establishes communications with a remote data server. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Server_Connect(*toolkit*, *connect_string*)

toolkit is a string value identifying the remote interface, for example, “ODBC”, “ORAINET”. Valid values for *toolkit* can be obtained from the **Server_DriverInfo( ) function**.

connect_string is a string value with additional information necessary to obtain a connection to the database.

Return Value

Integer

Description

The **Server_Connect( )** function establishes a connection to a data source. This function returns a connection number. A connection number is an identifier to the connection. This identifier must be passed to all server statements that you wish to operate on the connection.

The parameter *toolkit* identifies the MapInfo Professional remote interface toolkit through which the connection to a database server will be made. Information can be obtained about the possible values via calls to the **Server_NumDrivers( ) function** and the **Server_DriverInfo( ) function**.

The *connect_string* parameter supplies additional information to the toolkit necessary to obtain a connection to the database. The parameters depend on the requirements of the remote data source being accessed.

The connection string sent to **Server_Connect( )** has the form:

```
attribute=value[;attribute=value...]
```

ⓘ There are no spaces allowed in the connection string.

Passing the DLG=1 connect option provides a connect dialog box with active help buttons.

Microsoft ACCESS Attributes

The attributes used by ACCESS are:

Attribute	Description
DSN	The name of the ODBC data source for Microsoft ACCESS.
UID	The user login ID.
PWD	The user-specified password.
SCROLL	The default value is NO. If SCROLL=YES the ODBC cursor library is used for this connection allowing the ability to fetch first, last, previous, or record n of the database.

An example of a connection string for ACCESS is:

```
"DSN=MI ACCESS;UID=ADMIN;PWD=SECRET"
```

ORACLE ODBC Connection

If your application requires a connection string to connect to a data source, you must specify the data source name that tells the driver which section of the system information to use for the default connection information. Optionally, you may specify *attribute=value* pairs in the connection string to override the default values stored in the system information. These values are not written to the system information.

You can specify either long or short names in the connection string. The connection string has the form:

```
DSN=data_source_name[;attribute=value[;attribute=value]...]
```

An example of a connection string for Oracle is:

```
DSN=Accounting;HOST=server1;PORT=1522;SID=ORCL;UID=JOHN;PWD=XYZZY
```

The paragraphs that follow give the long and short names for each attribute, as well as a description. The defaults listed are initial defaults that apply when no value is specified in either the connection string or in the data source definition in the system information. If you specified a value for the attribute when configuring the data source, that value is the default.

ApplicationUsingThreads (AUT): ApplicationUsingThreads={0 | 1}. Ensures that the driver works with multi-threaded applications. When set to 1 (the initial default), the driver is thread-safe. When using the driver with single-threaded applications, you can set this option to 0 to avoid additional processing required for ODBC thread-safety standards.

ArraySize (AS): The number of bytes the driver uses for fetching multiple rows. Values can be an integer from 1 up to 4 GB. Larger values increase throughput by reducing the number of times the driver fetches data across the network. Smaller values increase response time, as there is less waiting time for the server to transmit data. The initial default is 60,000.

CatalogOptions (CO): CatalogOptions={0 | 1}. Determines whether the result column REMARKS for the catalog functions SQLTables and SQLColumns and COLUMN_DEF for the catalog function SQLColumns have meaning for Oracle. If you want to obtain the actual default value, set CO=1. The initial default is 0.

DataSourceName (DSN): A string that identifies an Oracle data source configuration in the system information. Examples include “Accounting” or “Oracle-Serv1.”

DescribeAtPrepare (DAP): DescribeAtPrepare={0 | 1}. Determines whether the driver describes the SQL statement at prepare time. When set to 0 (the initial default), the driver does not describe the SQL statement at prepare time.

EnableDescribeParam (EDP): EnableDescribeParam={0 | 1}. Determines whether the ODBC API function SQLDescribeParam is enabled, which results in all parameters being described with a data type of SQL_VARCHAR. This attribute should be set to 1 when using Microsoft Remote Data Objects (RDO) to access data. The initial default is 0.

EnableStaticCursorsForLongData (ESCLD): EnableStaticCursorsForLongData={0 | 1}. Determines whether the driver supports long columns when using a static cursor. Using this attribute causes a performance penalty at the time of execution when reading long data. The initial default is 0.

HostName (HOST): HostName={servername | IP_address}. Identifies the Oracle server to which you want to connect. If your network supports named servers, you can specify a host name such as Oracleserver. Otherwise, specify an IP address such as 199.226.224.34.

LockTimeOut (LTO): LockTimeOut={0 | -1}. Determines whether Oracle should wait for a lock to be freed before raising an error when processing a Select…For **Update statement**. When set to 0, Oracle does not wait. When set to -1 (the initial default), Oracle waits indefinitely.

LogonID (UID): The default logon ID (user name) that the application uses to connect to your Oracle database. A logon ID is required only if security is enabled on your database. If so, contact your system administrator to get your logon ID.

Password (PWD): The password that the application uses to connect to your Oracle database.

PortNumber (PORT): Identifies the port number of your Oracle listener. The initial default value is 1521. Check with your database administrator for the correct number.

ProcedureRetResults (PRR): ProcedureRetResults={0 | 1}. Determines whether the driver returns result sets from stored procedure functions. When set to 0 (the initial default), the driver does not return result sets from stored procedures. When set to 1, the driver returns result sets from stored procedures. When set to 1 and you execute a stored procedure that does not return result sets, you will incur a small performance penalty.

SID (SID): The Oracle System Identifier that refers to the instance of Oracle running on the server.

UseCurrrentSchema (UCS): UseCurrentSchema={0 | 1}. Determines whether the driver specifies only the current user when executing SQLProcedures. When set to 0, the driver does not specify only the current user. When set to 1 (the initial default), the call for SQLProcedures is optimized, but only procedures owned by the user are returned.

Oracle Spatial Attributes

Oracle8i Spatial is an implementation of a spatial database from Oracle Corporation. It has some similarities to the previous Oracle SDO implementation, but is significantly different. Oracle8i Spatial maintains the Oracle SDO implementation via a relational schema. However, MapInfo Professional does not support the Oracle SDO relational schema via OCI. MapInfo Professional does support simultaneous connections to Oracle8i through OCI and to other databases through ODBC. MapInfo Professional does not support downloading Oracle8I Spatial geometry tables via ODBC using the current ODBC driver from Intersolv. There is no DSN component.

Attribute	Description
LogonID (UID)	The logon ID (user name) that the application uses to connect to your Oracle database. A logon ID is required only if security is enabled on your database. If so, contact your system administrator to get your logon ID.
Password (PWD)	Your password. This, too, should be supplied by your system administrator.
ServerName (SRVR)	The name of the Oracle server.

An example of a connection string to access an Oracle8i Spatial server using TCP/IP is:

```
"SRVR=FATBOY;UID=SCOTT;PWD=TIGER"
```

SQL SERVER Attributes

If your application requires a connection string to connect to a data source, you must specify the data source name that tells the driver which section in the system information to use for the default connection information. Optionally, you may specify *attribute=value* pairs in the connection string to override the default values stored in system information. These values are not written to the system information.

The connection string has the form:

```
DSN=data_source_name[;attribute=value[;attribute=value]...]
```

An example of a connection string for SQL Server is:

```
DSN=Accounting;UID=JOHN;PWD=XYZZY
```

The paragraphs that follow give the long and short names, when applicable, for each attribute, as well as a description. The defaults listed are initial defaults that apply when no value is specified in either the connection string or in the data source definition in the system information. If you specified a value for the attribute when configuring the data source, that value is the default.

Address: The network address of the server running SQL Server. Used only if the Server keyword does not specify the network name of a server running SQL Server. Address is usually the network name of the server, but can be other names such as a pipe, or a TCP/IP port and socket address. For example, on TCP/IP: 199.199.199.5, 1433 or MYSVR, 1433.

AnsiNPW: AnsiNPW={yes | no}. Determines whether ANSI-defined behaviors are exposed. When set to yes, the driver uses ANSI-defined behaviors for handling NULL comparisons, character data padding, warnings, and NULL concatenation. When set to no, ANSI-defined behaviors are not exposed.

APP: The name of the application calling SQLDriverConnect (optional). If specified, this value is stored in the master.dbo.sysprocesses column program_name and is returned by sp_who and the Transact-SQL APP_NAME function.

AttachDBFileName: The name of the primary file of an attachable database. Include the full path and escape any slash (\) characters if using a C character string variable:

```
AttachDBFileName=c:\\MyFolder\\MyDB.mdf
```

This database is attached and becomes the default database for the connection. To use AttachDBFileName you must also specify the database name in either the SQLDriverConnnect DATABASE parameter or the SQL_COPT_CURRENT_CATALOG connection attribute. If the database was previously attached, SQL Server will not reattach it; it will use the attached database as the default for the connection.

AutoTranslate: AutoTranslate={yes | no}. Determines how ANSI character strings are translated. When set to yes, ANSI character strings sent between the client and server are translated by converting through Unicode to minimize problems in matching extended characters between the code pages on the client and the server.

These conversions are performed on the client by the SQL Server Wire Protocol driver. This requires that the same ANSI code page (ACP) used on the server be available on the client.

These settings have no effect on the conversions that occur for the following transfers:

- Unicode SQL_C_WCHAR client data sent to char, varchar, or text on the server.
- Char, varchar, or text server data sent to a Unicode SQL_C_WCHAR variable on the client.
- ANSI SQL_C_CHAR client data sent to Unicode nchar, nvarchar, or ntext on the server.
- Unicode char, varchar, or text server data sent to an ANSI SQL_C_CHAR variable on the client.
- When set to no, character translation is not performed.
- The SQL Server Wire Protocol driver does not translate client ANSI character SQL_C_CHAR data sent to char, varchar, or text variables, parameters, or columns on the server. No translation is performed on char, varchar, or text data sent from the server to SQL_C_CHAR variables on the client.
- If the client and SQL Server are using different ACPs, then extended characters can be misinterpreted.

DATABASE: The name of the default SQL Server database for the connection. If DATABASE is not specified, the default database defined for the login is used. The default database from the ODBC data source overrides the default database defined for the login. The database must be an existing database unless AttachDBFileName is also specified. If AttachDBFileName is specified, the primary file it points to is attached and given the database name specified by DATABASE.

LANGUAGE: The SQL Server language name (optional). SQL Server can store messages for multiple languages in sysmessages. If connecting to a SQL Server with multiple languages, this attribute specifies which set of messages are used for the connection.

Network: The name of a network library dynamic-link library. The name need not include the path and must not include the .dll file name extension, for example, Network=dbnmpntw.

PWD: The password for the SQL Server login account specified in the UID parameter. PWD need not be specified if the login has a NULL password or when using Windows NT authentication (Trusted_Connection=yes).

QueryLogFile: The full path and file name of a file to be used for logging data about long-running queries.

QueryLog_On: QueryLog_On={yes | no}. Determines whether long-running query data is logged. When set to yes, logging long-running query data is enabled on the connection. When set to no, long-running query data is not logged.

QueryLogTime: A digit character string specifying the threshold (in milliseconds) for logging long-running queries. Any query that does not receive a response in the time specified is written to the long-running query log file.

QuotedID: QuotedID={yes | no}. Determines whether QUOTED_IDENTIFIERS is set ON or OFF for the connection. When set to yes, QUOTED_IDENTIFIERS is set ON for the connection, and SQL Server uses the SQL-92 rules regarding the use of quotation marks in SQL statements. When set to no, QUOTED_IDENTIFIERS is set OFF for the connection, and SQL Server uses the legacy Transact-SQL rules regarding the use of quotation marks in SQL statements.

Regional: Regional={yes | no}. Determines how currency, date, and time data are converted. When set to yes, the SQL Server Wire Protocol driver uses client settings when converting currency, date, datetime, and time data to character data. The conversion is one way only; the driver does not recognize non-ODBC standard formats for date strings or currency values. When set to no, the driver uses ODBC standard strings to represent currency, date, and time data that is converted to string data.

SAVEFILE: The name of an ODBC data source file into which the attributes of the current connection are saved if the connection is successful.

SERVER: The name of a server running SQL Server on the network. The value must be either the name of a server on the network, or the name of a SQL Server Client Network Utility advanced server entry. You can enter “(local)” as the server name on Windows NT to connect to a copy of SQL Server running on the same computer.

StatsLogFile: The full path and file name of a file used to record SQL Server Wire Protocol driver performance statistics.

StatsLog_On: StatsLog_On={yes | no}. Determines whether SQL Server Wire Protocol driver performance data is available. When set to yes, SQL Server Wire Protocol driver performance data is captured. When set to no, SQL Server Wire Protocol driver performance data is not available on the connection.

Trusted_Connection: Trusted_Connection={yes | no}. Determines what information the SQL Server Wire Protocol driver will use for login validation. When set to yes, the SQL Server Wire Protocol driver uses Windows NT Authentication Mode for login validation. The UID and PWD keywords are optional. When set to no, the SQL Server Wire Protocol driver uses a SQL Server username and password for login validation. The UID and PWD keywords must be specified.

UID: A valid SQL Server login account. UID need not be specified when using Windows NT authentication.

WSID: The workstation ID. Typically, this is the network name of the computer on which the application resides (optional). If specified, this value is stored in the master.dbo.sysprocesses column hostname and is returned by sp_who and the Transact-SQL HOST_NAME function.

How to specify as a connection option

There are a few parameters that can be used for POSTgreSQL driver:

Definition	Keyword	Abbreviation
Data source description	Description	Nothing
Name of Server	Servername	Nothing
Postmaster listening port	Port	Nothing
User Name	Username	Nothing
Password	Password	Nothing
Debug flag	Debug	B2
Fetch Max Count	Fetch	A7
Socket buffer size	Socket	A8
Database is read only	ReadOnly	A0
Communication to backend logging	CommLog	B3
PostgreSQL backend protocol	Protocol	A1
Backend enetic optimizer	Optimizer	B4
Keyset query optimization	Ksqo	B5
Send to backend on connection	ConnSettings	A6
Recognize unique indexes	UniqueIndex	Nothing
Unknown result set sizes	UnknownSizes	A9
Cancel as FreeStmt	CancelAsFreeStmt	C1

Definition	Keyword	Abbreviation
Use Declare/Fetch cursors	UseDeclareFetch	B6
Text as LongVarchar	TextAsLongVarchar	B7
Unknowns as LongVarchar	UnknownsAsLongVarchar	B8
Bools as Char	BoolsAsChar	B9
Max Varchar size	MaxVarcharSize	B0
Max LongVarchar size	MaxLongVarcharSize	B1
Fakes a unique index on OID	FakeOidIndex	A2
Includes the OID in SQLColumns	ShowOidColumn	A3
Row Versioning	RowVersioning	A4
Show SystemTables	ShowSystemTables	A5
Parse Statements	Parse	C0
SysTable Prefixes	ExtraSysTablePrefixes	C2
Disallow Premature	DisallowPremature	C3
Updateable Cursors	UpdatableCursors	C4
LF <-> CR/LF conversion	LFConversion	C5
True is -1	TrueIsMinus1	C6
Datatype to report int8 columns as	BI	Nothing
Byte as LongVarBinary	ByteaAsLongVarBinary	C7
Use serverside prepare	UseServerSidePrepare	C8
Lower case identifier	LowerCaseIdentifier	C9
SSL mode	SSLmode	CA
Extra options	AB	Nothing
Abbreviate (simple setup of a recommendation value)	CX	Nothing

See Also:

Server Disconnect statement

Server_ConnectInfo( ) function

Purpose

Retrieves information about the active database connections. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Server_ConnectInfo(*ConnectionNo*, *Attr*)

ConnectionNumber is the integer returned by the **Server_Connect( ) function** that identifies the database connection.

Attr is a code indicating which information to return.

Return Value

String

Description

The **Server_ConnectInfo** function returns information about a database connection. The first parameter selects the connection number (starting at 1). The second parameter selects the kind of information that will be returned. Refer to the following table.

Attr value	ID	Server_ConnectInfo( ) returns:
SRV_CONNECT_INFO_DRIVER_NAME	1	String result, the name identifying the toolkit drivername associated with this connection.
SRV_CONNECT_INFO_DB_NAME	2	String result, returning the database name.
SRV_CONNECT_INFO_SQL_USER_ID	3	String result, returning the name of the SQL user ID.
SRV_CONNECT_INFO_DS_NAME	4	String result, returning the data source name.
SRV_CONNECT_INFO_QUOTE_CHAR	5	String result, returning the quote character.

Example

```
Dim dbname as String
Dim hdbc As Integer
hdbc = Server_Connect("ODBC", "DLG=1")
dbname=Server_ConnectInfo(hdbc, SRV_CONNECT_INFO_DB_NAME)
Print dbname
```

See Also:

Server_Connect( ) function

Server Create Map statement

Purpose

Identifies the spatial information for a server table. It does not alter the table to add the spatial columns. For this release, we have added the option to place Oracle 11g annotation text in MapInfo maps. You can issue this statement from the MapBasic Window in MapInfo Professional.

To support the changes to the Make Table Mappable dialog box, we use the Server <Connection Number> Create Map statement to register the metadata in the MAP CATALOG. To support ANNOTATION TEXT, we have introduced a Text object type. The statement is now:

Syntax

```
Server ConnectionNumber Create Map
   For linked_table
   Type { MICODE columnname | XYINDEX (xcolumnname, ycolumnname) | {
      SPATIALWARE | OR_SP  | SQLSERVERSPATIAL {
      GEOMETRY | GEOGRAPHY} | POSTGIS}
      columnname}
   [ CoordSys... ]
   [ MapBounds { Data | Coordsys | Values ( x1, y1 )( x2, y2 )} ]
   [ ObjectType { Point | Line | Region | Text | ALL } ]
   [ Symbol (...) ]
   [ Linestyle Pen(...) ]
   [ Regionstyle Pen(...) Brush(...) ]
   [ Style Type style_number [ Column column_name ] ]
```

Text supports the creation of the text object for annotation text. The ALL option does not include this text object.

connectionNumber is an integer value that identifies the specific connection.

linked_table is the name of an open, linked ODBC table.

columnname is the name of the column containing the coordinates for the specified type.

xcolumnname is the name of the X column containing longitude value of the coordinate

ycolumnname is the name of the Y column containing latitude value of the coordinate

x1, *y1*, *x2*, *y2* define the coordinate system bounds.

CoordSy clause specifies the coordinate system and projection to be used.

MapBounds clause allows you to specify what to store for the entire/default table view bounds in the MapCatalog. The default is **Data** which calculates the bounds of all the data in the layer. (For programs compiled before 7.5, the default will is **Coordsys**).

Coordsys stores the coordinate system bounds. This is not recommended as it may cause the entire layer default view to appear empty if the Coordsys bounds are significantly greater than the bounds of the actual data. Most users are zoomed out too far to see their data using this option.

Values lets you specify your own bounds values for the MapCatalog.

ObjectType clause specifies the type of object in the table: points, lines, regions, text, or all objects. If no **ObjectType** clause is specified, the default is **Point**. The **Text** option allows for the placement of Oracle Spatial annotation text into a text object., and the type for this option is ORA_SP. The **ALL** option does not include text.

Symbol is a valid **Symbol clause** to specify a point style.

Linestyle Pen is a valid **Pen clause** that specifies the line style to be used for a line object type.

Regionstyle Pen is a valid **Pen clause** and **Brush** is a valid **Brush clause** that specifies the line style and fill style to be used for a region object type.

StyleType sets per-row symbology. *style_number* is a value either 0 or 1. The **Column** keyword and argument must be present when *style_number* is set to 1 (one). When the *style_number* is set to zero the **Column** keyword is ignored and the rendition columns in the MapCatalog are cleared.

Description

The **Server Create Map** statement makes a table linked to a remote database mappable. For a SpatialWare, Oracle Spatial, SQL Server Spatial or PostGIS table, you can make the table mappable for points, lines, or regions. For all other tables, you can make a table mappable for points only. Any MapInfo Professional table may be displayed in a Browser, but only a mappable table can have graphical objects attached to it and be displayed in a Map window.

If Oracle9i is the server and the coordinate system is specified as Lat/Long without specifying the datum, the default datum, World Geodetic System 1984(WGS 84), will be assigned to the Lat/Long coordinate system. This behavior is consistent with the **Server Create Table statement** and Easyloader.

Attribute Types	Description
ORA_SP *columnname*	OracleSpatial
SPATIALWARE	SpatialWare for SQL Server
MICODE	XYINDEX
SQLSERVERSPATIAL GEOMETRY	SQL Server Spatial Geometry
SQLSERVERSPATIAL GEOGRAPHY	SQL Server Spatial Geography
POSTGIS	PostGIS for PostgreSQL

Examples

```
Sub Main
   Dim ConnNum As Integer
   ConnNum = Server_Connect("ODBC",
```

```
      "DSN=SQLServer;DB=QADB;UID=mipro;PWD=mipro")
   Server ConnNum Create Map For "Cities"
   Type SPATIALWARE
   CoordSys Earth Projection 1, 0
   ObjectType All
   ObjectType Point
   Symbol (35,0,12)
   Server ConnNum Disconnect
End Sub
```

The following is an example of the MapBasic statement for the ANNOTEXT_TABLE:

```
Server 1 Create Map For """MIPRO"".""ANNOTEXT_TABLE"""
   Type ORA_SP "TEXTOBJ"
   CoordSys Earth Projection 12, 62, "m", 0 Bounds
      (-34012036.7393, -8625248.51472) (34012036.7393, 8625248.51472)
   mapbounds data
   ObjectType Text
```

See Also:

Server Link Table statement, **Unlink statement**

Server Create Style statement

Purpose

Changes the per object style settings for a mapped table. This statement is similar to the **Server Set Map statement** and returns success or failure. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Server ConnectionNumber Set Map linked_table...
   [ Style Type style_number [ Column column_name ] ]
```

connectionNumber is an integer value that identifies the specific connection.

linked_table is the name of an open linked ODBC table

columnname is the name of the column containing the coordinates for the specified type

StyleType sets per-row symbology. *style_number* is a value either 0 or 1. The **Column** keyword and argument must be present when *style_number* is set to 1 (one). When the *style_number* is set to zero the **Column** keyword is ignored and the rendition columns in the MapCatalog are cleared.

Description

In order to succeed, the MapCatalog must have the structure to support styles. It must contain the columns RENDITIONTYPE, RENDITIONCOLUMN, and RENDITIONTABLE. The command should not succeed if the style columns are not character or varchar columns. The SQL statement itself will probably fail if it tries to set a string value into a column with a different data type.

Example

```
Server 2 Create Map For "qadb:sample.arc"
Type MICODE "mi_sql_micode" ("mi_sql_x","mi_sql_y")
CoordSys Earth Projection 1, 0 ObjectType Point Symbol (35,0,12) Style
Type 1 Column "mi_style"
```

See Also:

Server_Connect( ) function

Server Create Table statement

Purpose

Creates a new table on a specified remote database. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Server ConnectionNumber Create Table TableName
   ( ColumnName ColumnType [,...])
   [ KeyColumn ColumnName ]
   [ObjectColumn ColumnName [Type SQLServerSpatial
      {Geometry | Geography}]]
   [ StyleColumn ColumnName ]
   [ CoordSys... ]
```

ConnectionNumber is an integer value that identifies the specific connection to a database.

TableName is the name of the table as you want it to appear in a database.

ColumnName is the name of a column to create. Column names can be up to 31 characters long, and can contain letters, numbers, and the underscore(_) character. Column names cannot begin with numbers.

ColumnType is the data type associated with the column.

KeyColumn clause specifies the key column of the table.

ObjectColumn clause specifies the spatial geometry/object column of the table.

StyleColumn clause specifies the Per Row Style column, which allows the use of different object styles for each row on the table.

CoordSys… clause specifies the coordinate system and projection to be used.

Description

The **Server Create Table** statement creates a new empty table on the given database of up to 250 columns.

TableName is the name of the table as you want it to appear in database. The name can include a schema name, which specifies the schema that the table belongs to. If no schema name is provided, the table belongs to the default schema. The user is responsible for providing an eligible schema name and must know if the login user has the proper permissions on the given schema. This extension is for SQL Server 2005 only.

The length of *TableName* varies with the type of database. We recommend using 14 or fewer characters for a table name to ensure that it works correctly for all databases. The maximum *TableName* length is 14 characters.

ColumnType uses the same data types defined and provided in the **Create Table statement**. Some types may be converted to the database-supported types accordingly, once the table is created on the database.

If the optional **KeyColumn** clause is specified, a unique index will be created on this column. We recommend using this clause since it is also allows MapInfo Professional to open the table for live access.

The optional **ObjectColumn** clause enables you to create a table with a spatial geometry/object column. If it is specified, a spatial index will also be created on this column. However, if the server does not have the ability to handle spatial geometry/objects, the table will not be created. If the server is an SQL Server with SpatialWare, the table is also spatialized once the table is created. If the Server is Oracle Spatial, spatial metadata is updated once the table is created.

If **Server Create Table** is used and the **ObjectColumn** clause is passed in the statement, you will also have to use the **Server Create Map statement** in order to open the table in MapInfo Professional.

The optional **CoordSys clause** becomes mandatory only if the table is created with spatial object/geometry on Oracle Spatial (Oracle8i or later with spatial option). If Oracle9i is the server and the coordinate system is specified as Lat/Long without specifying the datum, the default datum, World Geodetic System 1984(WGS 84), will be assigned to the Lat/Long coordinate system. The coordinate system must be the same as the one specified in the **Server Create Map statement** when making it mappable. For other DBMS, this clause has no effect on table creation.

The supported databases include Oracle, SQL Server, PostGIS and Microsoft Access. However, to create a table with a spatial geometry/object column, SpatialWare is required for SQL Server and the spatial option is required for Oracle.

Notes on DateTime and Time Data Types

There is no specific change in terms of syntax. We do have following restrictions for the some data types:

The datatypes Time and DateTime are useful but you must consider the database when using them. Most databases do not have a corresponding DBMS TIME types. Before this release, we only supported the Date type. Even the Date was converted to server type If the server did not support Date type. In MapBasic 9.0 and later, this statement only supports the types that the server also supports. Therefore, the Time type is prohibited from this statement for Oracle, SQL Server and

Access, and the Date type data type is prohibited for SQL Server and Access. Those "unsupported" types should be replaced with DateTime if you still want to create the table that contains time information on a column.

For Microsoft SQL Server and Access and verisons of MapInfo Professional older than 9.0, the conversion was done in the background. As of version 9.0, users must choose DATETIME instead of DATE or the operation fails.

Examples

The following examples show how to create a table named ALLTYPES that contains seven columns that cover each of the data types supported by MapInfo Professional, plus the three columns Key, SpatialObject, and Style columns, for a total of ten columns.

For SQL Server with SpatialWare:

```
dim hodbc as integer
hodbc = server_connect("ODBC", "dlg=1")
Server hodbc Create Table ALLTYPES( Field1 char(10),Field2 integer,Field3
SmallInt,Field4 float,Field5 decimal(10,4),Field6 date,Field7 logical)
KeyColumn SW_MEMBER
ObjectColumn SW_GEOMETRY
StyleColumn MI_STYLE
```

For Oracle Spatial:

```
dim hodbc as integer
hodbc = server_connect("ORAINET", "SRVR=cygnus;UID=mipro;PWD=mipro")
Server hodbc Create Table ALLTYPES( Field1 char(10),Field2 integer,Field3
SmallInt,Field4 float,Field5 decimal(10,4),Field6 date,Field7logical)
   KeyColumn MI_PRINX
   ObjectColumn GEOLOC
   StyleColumn MI_STYLE
   Coordsys Earth Projection 1, 0
```

See Also:

Create Map statement, **Server Create Map statement**, **Server Link Table statement**, **Unlink statement**

Server Create Workspace statement

Purpose

Creates a new workspace in the database (Oracle 9i or later). You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Server *ConnectionNumber* **Create**
  Workspace *WorkspaceName*

```
[ Description Description ]
[ Parent ParentWorkspaceName ]
```

ConnectionNumber is an integer value that identifies the specific connection.

WorkspaceName is the name of the workspace. The name is case sensitive, and it must be unique. The length of a workspace name must not exceed 30 characters.

Description is a string to describe the workspace.

ParentWorkspaceName is the name of the workspace which will be the parent of the new workspace *WorkspaceName*. By default, when a workspace is created, it is created from the topmost, or LIVE, database workspace.

Description

This statement only applies to Oracle9i or later. The new workspace *WorkspaceName* is a child of the parent workspace *ParentWorkspaceName* or LIVE if the Parent is not specified.

Refer to the *Oracle9i Application Developer's Guide - Workspace Manager* for more information.

Examples

The following example creates a workspace named MIUSER in the database.

```
Dim hdbc As Integer
hdbc = Server_Connect("ORAINET", "SRVR=TROYNY;UID=MIUSER;PWD=MIUSER")
Server hdbc Create
Workspace "MIUSER"
Description "MIUser private workspace"
```

The following example creates a child workspace under MIUSER in the database.

```
Dim hdbc As Integer
hdbc = Server_Connect("ORAINET", "SRVR=TROYNY;UID=MIUSER;PWD=MIUSER")
Server hdbc Create Workspace "MBPROG" Description "MapBasic project"
Parent "MIUSER"
```

See Also:

Server Remove Workspace statement, **Server Versioning statement**

Server Disconnect statement

Purpose

Shuts down the communication established via the **Server_Connect( ) function** with the remote data server. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Server ConnectionNumber Disconnect
```

ConnectionNumber is an integer value that identifies the specific connection.

Description

The **Server Disconnect** statement shuts down the database connection. All resources allocated with respect to the connection are returned to the system.

Example

```
Dim hdbc As Integer
hdbc = Server_Connect("ODBC", "DLG=1")
Server hdbc Disconnect
```

See Also:

Server_Connect( ) function

Server_DriverInfo( ) function

Purpose

Retrieves information about the installed toolkits and data sources. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Server_DriverInfo(*DriverNo*, *Attr*)

DriverNo is an integer value assigned to an interface toolkit by MapInfo Professional when you start MapInfo Professional.

Attr is a code indicating which information to return.

Return Value

String

Description

The **Server_DriverInfo( )** function returns information about the data sources. The first parameter selects the toolkit (starting at 1). The total number of toolkits can be obtained by a call to the **Server_NumDrivers( ) function**. The second parameter selects the kind of information that will be returned. Refer to the following table.

Attr value	ID	Server_DriverInfo( ) returns:
SRV_DRV_INFO_NAME	1	String result, the name identifying the toolkit. ODBC indicates an ODBC data source. ORAINET indicates an Oracle Spatial connection.

Attr value	ID	Server_DriverInfo( ) returns:
SRV_DRV_INFO_NAME_LIST	2	String result, returning all the toolkit names, separated by semicolons. Specifically, ODBC, ORAINET. The *DriverNo* parameter is ignored.
SRV_DRV_DATA_SOURCE	3	String result, returning the name of the data sources supported by the toolkit. Repeated calls will fetch each name. After the last name for a particular toolkit, the function will return an empty string. Calling the function again for that toolkit will cause it to start with the first name on the list again.

Example

```
Dim dlg_string, source As String
dlg_string = Server_DriverInfo(0, SRV_DRV_INFO_NAME_LIST)
source = Server_DriverInfo(1, SRV_DRV_DATA_SOURCE)
While source <> ""
   Print "Available sources on toolkit " +
   Server_DriverInfo(1, SRV_DRV_INFO_NAME) + ": " +
      source
   source = Server_DriverInfo(1,
      SRV_DRV_DATA_SOURCE)
Wend
```

See Also:

Server_NumDrivers( ) function

Server_EOT( ) function

Purpose

Determines whether the end of the result table has been reached via a **Server Fetch statement**. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Server_EOT(*StatementNumber*)

StatementNumber is the number of the Server Fetch statement you are checking.

Return Value

Logical

Description

The **Server_EOT( )** function returns TRUE or FALSE indicating whether the previous Server Fetch statement encountered a condition where there was no more data to return. Attempting to fetch a previous record immediately after fetching the first record causes this to return TRUE. Attempting to fetch the next record after the last record also returns a value of TRUE.

Example

```
Dim hdbc, hstmt As Integer
hdbc = Server_Connect("ODBC", "DLG=1")
hstmt = Server_Execute(hdbc, "Select * from ADDR")
Server hstmt Fetch FIRST
While Not Server_EOT(hstmt)
   ' Processing for each row of data ...
   Server hstmt Fetch Next
Wend
```

See Also:

Server Fetch statement

Server_Execute( ) function

Purpose

Sends a SQL string to execute on a remote data server. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Server_Execute(*ConnectionNumber*, *server_string*)

ConnectionNumber is an integer value that identifies the specific connection.

server_string is any valid SQL statement supported by the connected server. Refer to the SQL language guide of your server database for information on valid SQL statements.

Return Value

Integer

Description

The **Server_Execute( )** function sends the *server_string* (an SQL statement) to the server connection specified by the *ConnectionNumber*. Any valid SQL statement supported by the active server is a valid value for the *server_string* parameter. Refer to the SQL language guide of your server database for information on valid SQL statements.

This function returns a statement number. The statement number is used to associate subsequent SQL requests, like the **Server Fetch statement** and the **Server Close statement**, to a particular SQL statement.

You should perform a Server Close statement for each **Server_Execute( )** function as soon as you are done using the statement handle. For selects, this is as soon as you are done fetching the desired data. This will close the cursor on the remote server and free up the result set. Otherwise, you can exceed the cursor limit and further executes will fail. Not all database servers support forward and reverse scrolling cursors. For other SQL commands, issue a Server Close statement immediately following the **Server_Execute( )** function.

```
Dim hdbc, hstmt As Integer
hdbc = Server_Connect("ODBC", "DLG=1")
hstmt = Server_Execute(hdbc, "Select * from ADDR")
Server hstmt Close
```

Example

```
Dim hdbc, hstmt As Integer
hdbc = Server_Connect("ODBC", DSN=ORACLE7;DLG=1")
hstmt = Server_Execute (hdbc,
   "CREATE TABLE NAME_TABLE (NAME CHAR (20))")
Server hstmt Close
hstmt = Server_Execute (hdbc,
   "INSERT INTO NAME_TABLE VALUES ('Steve')")
Server Close hstmt
hstmt = Server_Execute ( hdbc,
   "UPDATE NAME_TABLE SET name = 'Tim' ")
Server Close hstmt
Server hdbc Disconnect
```

See Also:

Server Close statement, **Server Fetch statement**

Server Fetch statement

Purpose

Retrieves result set rows from a remote data server. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Server *StatementNumber* **Fetch** [**NEXT** | **PREV** | **FIRST** | **LAST** | [**REC**]*recno*]

or

Server *StatementNumber* **Fetch INTO Table** [**FILE** *path*]

StatementNumber is an integer value that identifies information about an SQL statement.

recno is an integer representing the record to fetch.

path is the path to an existing table.

Description

The **Server Fetch** statement retrieves result set data (specified by the *StatementNumber*) from the database server. For fetching the data one row at a time, it is placed in local storage and can be bound to variables with the **Server Bind Column statement**, or retrieved one column at a time with the Server_ColumnInfo(SRV_COL_INFO_VALUE) function. The other option is to fetch an entire result set into a MapInfo table at once, using the **Into Table** clause.

The **Server Fetch** and **Server Fetch Into** statements halt and set the error code `ERR( ) = ERR_SRV_ESC` if the user presses Esc. This allows your MapBasic application using the **Server Fetch** statements to handle the escape.

Following a **Server Fetch Into** statement, the MapInfo table is committed and there are no outstanding transactions on the table. All character fields greater than 254 bytes are truncated. All binary fields are downloaded as double length hexadecimal character strings. The column names for the downloaded table will use the column alias name if a column alias is specified in the query.

Null Handling

When you execute a **Select statement** and fetch a row containing a table column that contains a null, the following behavior occurs. There is no concept of null values in a MapInfo table or variable, so the default value is used within the domain of the data type. This is the value of a MapBasic variable that is DIMed but not set. However, an Indicator is provided that the value returned was null.

For Bound variables (see **Server Bind Column statement**), a status variable can be specified and its value will indicate if the value was null following the fetch. For unbound columns, SRV_COL_INFO with the *Attr* type SRV_COL_INFO_STATUS will return the status which can indicate null.

Refer to the *MapBasic User Guide* for information on how MapInfo Professional interprets data types.

Error Conditions

The command **`Server`** *`n`* **`Fetch Into table`** generates an error condition if any attempts to insert records into the local MapInfo table fail. The commands **`Server`** *`n`* **`Fetch`** [**`Next`**|**`Prev`**|*`recno`*] generate errors if the desired record is not available.

Example 1

```
' An example of Server Fetch downloading into a MapInfo table
Dim hdbc, hstmt As Integer
hdbc = Server_Connect("ODBC", "DLG=1")
hstmt = Server_Execute(hdbc, "Select * from emp")
Server hstmt Fetch Into "MyEmp"
Server hstmt Close
```

Example 2

```
' An example of Server Fetch using bound variables
Dim hdbc, hstmt As Integer
dim NameVar, AddrVar as String
dim NameStatus, AddrStatus as Integer
```

```
hdbc = Server_Connect("ODBC", "DLG=1")
hstmt = Server_Execute(hdbc, "Select Name, Addr from emp")
Server hstmt Bind Column 1 to NameVar, NameStatus
Server hstmt Bind Column 2 to AddrVar, AddrStatus
Server hstmt Fetch Next
While Not Server_Eot(hstmt)
   Print "Name = " + NameVar + "; Address = " + AddrVar
   Server hstmt Fetch Next
Wend
```

See Also:

Server_ColumnInfo( ) function

Server_GetODBCHConn( ) function

Purpose

Returns the ODBC connection handle associated with the remote database connection. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Server_GetODBCHConn(*ConnectionNumber*)

ConnectionNumber is the integer returned by the **Server_Connect( ) function** that identifies the database connection.

Description

This function returns an integer containing the ODBC connection handle associated with the remote database connection. This enables you to call any function in the ODBC DLL to extend the functionality available through the MapBasic **Server**… statements.

Example

```
'* Find the identity of the Connected database
DECLARE FUNCTION SQLGetInfo LIB "ODBC32.DLL" (BYVAL odbchdbc AS INTEGER,
BYVAL infoflag AS INTEGER, val AS STRING, BYVAL len AS INTEGER, outlen AS
INTEGER) AS INTEGER
Dim rc, outlen, hdbc, odbchdbc AS INTEGER
Dim DBName AS STRING
' Connect to a database
hdbc = Server_Connect("ODBC", "DLG=1")
odbchdbc = Server_GetodbcHConn(hdbc) ' get ODBC connection handle
' Get database name from ODBC
DBName = STRING$(33, "0") ' Initialize output buffer
rc = SQLGetInfo(odbchdbc, 17 , DBName, 40, outlen) ' get ODBC Database
Name
' Display results (database name)
if rc <> 0 THEN
   Note "SQLGetInfo Error rc=" + rc + ", outlen=" + outlen
```

```
else
   Note "Connected to Database: " + DBName
end if
```

See Also:

Server_GetODBCHStmt( ) function

Server_GetODBCHStmt( ) function

Purpose

Return the ODBC statement handle associated with the MapBasic **Server**… statements. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Server_GetODBCHStmt(*StatementNumber*)

StatementNumber is the integer returned by the Server_Execute( ) function that identifies the result set of the SQL statement executed.

Description

This function returns the ODBC statement handle associated with the MapBasic **Server**… statements. This enables you to call any ODBC function to extend the functionality available through the MapBasic **Server**… statements.

Example

```
' Find the Number of rows affected by an Update
Dim rc, outlen, hdbc, hstmt, odbchstmt AS INTEGER
Dim RowsUpdated AS INTEGER
' Find the Number of rows affected by an Update
DECLARE FUNCTION SQLRowCount LIB "ODBC32.DLL" (BYVAL odbchstmt AS INTEGER,
rowcnt AS INTEGER) AS INTEGER
hdbc = Server_Connect("ODBC", "DLG=1")
hstmt = Server_Execute(hdbc, "UPDATE TIML.CUSTOMER SET STATE='NY' WHERE
STATE='NY'")
odbchstmt = Server_GetodbcHStmt(hstmt)
rc = SQLRowCount(odbchstmt, RowsUpdated)
Note "Updated " + RowsUpdated + " New customers to Tier 1"
```

See Also:

Server_GetODBCHConn( ) function

Server Link Table statement

Purpose

Creates a linked table. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax 1

```
Server Link Table
   SQLQuery
   Using ConnectionString
   Into TableName
   Toolkit Toolkitname
   [ File FileSpec ]
   [ ReadOnly ]
   [ Autokey { Off | On }]
```

Syntax 2

```
Server ConnectionNumber Link Table
   SQLQuery
   Toolkit toolkitname
   Into TableName
   [ File FileSpec ]
   [ ReadOnly ]
   [ Autokey { Off | On }]
```

ConnectionNumber is an integer value that identifies an existing connection.

SQLQuery is a SQL query statement (in native SQL dialect plus object keywords) that generates a result set. The MapInfo linked table is linked to this result set.

ConnectionString is a string used to connect to a database server. See **Server_Connect( ) function**.

TableName is the alias of the MapInfo table to create.

FileSpec is an optional tab filename. If the parameter is not present, the tab filename is created based on the alias and current directory. If a *FileSpec* is given and a tab file with this name already exists, an error occurs.

ReadOnly indicates that the table should not be edited.

Toolkitname is a string indicating the type of connection, ODBC or ORAINET.

If **Autokey** is set **On**, the table will be opened with key auto-increment option. If **Autokey** is set **Off** or this option is ignored, the table will be opened without key auto-increment.

Description

This statement creates a linked MapInfo table on disk. The table is opened and enqueued. This table is considered a MapInfo base table under most circumstances, except the following: The MapBasic **Alter Table statement** will fail with linked tables. Linked tables cannot be packed. The Pack Table dialog box will not list linked tables. Use the **Server Link Table** syntax to establish a connection to a database server and to link a table. Use the **Server** *ConnectionNumber* **Link Table** to link a table using an existing connection. Linked tables contain information to reestablish connections and identify the remote data to be updated. This information is stored as metadata in the tab file.

The absence of the **ReadOnly** keyword does not indicate that the table is editable. The linked table can be read-only under any of the following circumstances: the result set is not editable; the result set does not contain a primary key; there are no editable columns in the result set; and, the **ReadOnly** keyword is present. If the server is Oracle, **Autokey** indicates if the key auto-increment is used or not.

SQL Query Syntax

The MapInfo keyword **OBJECT** may be used to reference the spatial column(s) within the SQL Query. MapInfo Professional translates the keyword **OBJECT** into the appropriate spatial column(s). A SELECT*FROM *tablename* will always pick up the spatial columns, but if you want to specify a subset of columns, use the keywords **OBJECT**. For example:

```
SELECT col1, col2, OBJECT
FROM tablename
```

will download the two columns plus the spatial object. This syntax will work for any database that MapInfo Professional supports.

MapInfo Professional Spatial Query

MapInfo Professional supports the keyword **WITHIN** which is used for spatial queries. It is used for selecting spatial objects in a table that exists within an area identified by a spatial object. The following two keywords may be used along with the **WITHIN** keyword:

- **CURRENT MAPPER**: entire rectangular area shown in the current Map window.
- **SELECTION**: area within the selection n the current Map window.

The syntax to find all of the rows in a table with a spatial object that exists within the current Map window would be as follows:

```
SELECT col1, col2, OBJECT
FROM tablename
WHERE OBJECT WITHIN CURRENT_MAPPER
```

This syntax will work for any database that MapInfo Professional supports. MapInfo Professional will also execute spatial SQL queries that are created using the native SQL syntax for the spatial database. Valid values for *toolkitname* can be found in **Server_DriverInfo( ) function**.

Examples

```
Declare Sub Main
Sub Main
   Open table "C:\mapinfo\data\states.tab"
   Server Link Table "Select * from Statecap" Using
      "DSN=MS Access;DBQ=C:\MSOFFICE\ACCESS\DB1.mdb"
      Into test File "C:\tmp\test"
   Map From Test,States
End Sub 'Main
Declare Sub Main
Sub Main
   Dim ConnNum As Integer
   ConnNum = Server_Connect("ODBC","DSN=SQS;PWD=sysmal;SRVR=seneca")
   Server ConnNum Link Table
      "Select * from CITY_1"
   Into temp
   Map From temp

   Server ConnNum Disconnect
End Sub
```

The following example creates a linked table.

```
Dim hdbc As Integer
hdbc = Server_Connect("ORAINET", "SRVR=ONTARIO;UID=MIPRO;PWD=MIPRO")
Server hdbc link table
   "Select *  From ""MIPRO"".""SMALLINTEGER"""
   Toolkit "ORAINET"
   Into SMALLINTEGER
   Autokey ON
Map From SMALLINTEGER
```

See Also:

Close Table statement, **Commit Table statement**, **Drop Table statement**, **Rollback statement**, **Save File statement**, **Server Refresh statement**, **Unlink statement**

Server_NumCols( ) function

Purpose

Retrieves the number of columns in the result set. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Server_NumCols(*StatementNumber*)

StatementNumber is an integer value that identifies information about an SQL statement.

Return Value

Integer

Description

The **Server_NumCols( )** function returns the number of columns in the result set currently referenced by *StatementNumber*.

Example

```
Dim hdbc, hstmt As Integer
hdbc = Server_Connect("ODBC", "DLG=1")
hstmt = Server_Execute(hdbc, "Select Name, Addr from emp")
Print "Number of columns = " + Server_NumCols(hstmt)
```

See Also:

Server_ColumnInfo( ) function

Server_NumDrivers( ) function

Purpose

Retrieves the number of database connection toolkits currently installed for access from MapInfo. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

Server_NumDrivers( )

Return Value

Integer

Description

The **Server_NumDrivers( )** function returns the number of database connection toolkits installed for use by MapInfo Professional.

Example

```
Print "Number of drivers = " + Server_NumDrivers( )
```

See Also:

Server_DriverInfo( ) function

Server Refresh statement

Purpose

Resynchronizes a linked or live table with the remote database data. This command can only be run when no edits are pending against the table. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

`Server Refresh` *`TableName`*

TableName is the name of an open MapInfo linked table.

Description

If the connection to the database is currently open then the refresh simply occurs. If the connection is not currently open, then the connection will be made. If there is any information needed, such as a password, the user will be prompted for it.

Refreshing the table involves:

1. If the table contains records, delete all the records and objects from the live or linked table by erasing the files and recreating the table, not by using the MapBasic **Delete statement**.
2. If a connection handle is stored with the TABLE structure, use it. Otherwise, reconnect using the connection string stored in the live or linked table metadata.
3. Convert SQL query stored in metadata to RDBMS-specific query.
4. Execute SQL query on RDBMS.
5. Fetch rows from the RDBMS cursor, filling the table. Put up a MapInfo Professional progress bar during this operation.
6. Close RDBMS cursor.

Example

```
Server Refresh "City_1k"
```

See Also:

Commit Table statement, **Server Link Table statement**, **Unlink statement**

Server Remove Workspace statement

Purpose

Discards all row versions associated with a workspace and deletes the workspace in the database (Oracle 9i or later). You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Server ConnectionNumber Remove
   Workspace WorkspaceName
```

ConnectionNumber is an integer value that identifies the specific connection.

WorkspaceName is the name of the workspace. The name is case sensitive.

Description

This statement only applies to Oracle9i or later. This operation can only be performed on leaf workspaces (the bottom-most workspaces in a branch in the hierarchy). There must be no other users in the workspace being removed.

Examples

The following example removes the MIUSER workspace in the database.

```
Dim hdbc As Integer
hdbc = Server_Connect("ORAINET", "SRVR=TROYNY;UID=MIUSER;PWD=MIUSER")
Server hdbc Remove Workspace "MIUSER"
```

See Also:

Server Create Workspace statement

Server Rollback statement

Purpose

Discards changes made on the remote data server during the current unit of work. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Server ConnectionNumber Rollback
```

ConnectionNumber is an integer value that identifies the specific connection.

Description

The **Server Rollback** statement discards the effects of all SQL statements on the connection back to the **Server Begin Transaction statement**. You must have an open transaction initiated by **Server Begin Transaction statement** before you can use this command.

Example

```
hdbc = Server_Connect("ODBC", "DLG=1")
Server hdbc Begin Transaction

...

' All changes since begin_transaction are about ' to be discarded
Server hdbc Rollback
```

See Also:

Server Begin Transaction statement, **Server Commit statement**

Server Set Map statement

Purpose

Changes the object styles for a mappable ODBC table. This updates the MapCatalog. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Server ConnectionNumber Set Map linked_table
   [ ObjectType { Point | Line | Region | Text | ALL } ]
   [ Symbol(...) ]
   [ Linestyle Pen(...) ]
   [ Regionstyle Pen(...) Brush(...) ]
```

ConnectionNumber is an integer value that identifies the specific connection.

linked_table is the name of an open linked DBMS table.

ObjectType clause specifies the type of object in the table and allows you to specify objects as points, lines, regions, text, or all objects, see **Server Create Map statement** for details.

Symbol is a valid **Symbol clause** to specify a point style.

Linestyle Pen specifies the line style to be used for a line object type.

Regionstyle Pen(…) **Brush**(…) clause specifies the line style and fill style to be used for a region object type.

Description

The **Server Set Map** statement changes the object styles of an open mappable ODBC table. An ODBC table is made mappable with the **Server Create Map statement**.

Example

```
Declare Sub Main
Sub Main
   Dim ConnNum As Integer
   ConnNum = Server_Connect("ODBC", "DSN=SQS;PWD=sys;SRVR=seneca")
   Server ConnNum Set Map "Cities"
      ObjectType Point
      Symbol (35,0,12)
   Server ConnNum Disconnect
End Sub
```

See Also:

Server Create Map statement

Server Versioning statement

Purpose

Version-enables or disables a table on Oracle 9i or later, which creates or deletes all the necessary structures to support multiple versions of rows to take advantage of Oracle Workspace Manager. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Server ConnectionNumber Versioning {
   ON [ History HistoryValue ] |
   OFF [ Force { OFF | ON }]
} Table ServerTableName
```

ON | **OFF** indicates to enable (when it is **ON**) a table versioning or disable (when it is **OFF**) a table versioning.

ConnectionNumber is an integer value that identifies the specific connection.

ServerTableName is the name of the table on Oracle server to be version-enabled/disabled. The length of a table name must not exceed 25 characters. The name is not case sensitive.

History is an optional parameter when version-enabling a table (**ON**).

History clause specifies how to track modifications to *ServerTableName*, for example, lets you timestamp changes made to all rows in a version-enabled table and to save a copy of either all changes or only the most recent changes to each row. *HistoryValue* must be one of the following constant values:

- **SRV_WM_HIST_NONE** (0): No modifications to the table are tracked. (This is the default.)
- **SRV_WM_HIST_OVERWRITE** (1): The with overwrite (W_OVERWRITE) option. A view named ServerTableName_HIST is created to contain history information, but it will show only the most recent modifications to the same version of the table. A history of modifications to the version is not maintained; that is, subsequent changes to a row in the same version overwrite earlier changes. (The CREATETIME column of the TableName_HIST view contains only the time of the most recent update.)

- **SRV_WM_HIST_NO_OVERWRITE** (2): The without overwrite (WO_OVERWRITE) option. A view named ServerTableName_HIST is created to contain history information, and it will show all modifications to the same version of the table. A history of modifications to the version is maintained; that is, subsequent changes to a row in the same version do not overwrite earlier changes.

  However, there are many restrictions on tables to use this option. Please refer the Oracle9i Application Developer's Guide - Workspace Manager for more information.

Force is an optional parameter, when disabling a version-enabled table (**OFF**).

If **Force** is set **ON**, all data in workspaces other than LIVE to be discarded before versioning is disabled. **OFF** (the default) prevents versioning from being disabled if *ServerTableName* was modified in any workspace other than LIVE and if the workspace that modified *ServerTableName* still exists.

Description

This statement only applies to Oracle9i or later. The table, *ServerTableName*, that is being version-enabled must have a primary key defined. Only the owner of a table or a user with the WM_ADMIN role can enable or disable versioning on the table. Tables that are version-enabled and users that own version-enabled tables cannot be deleted. You must first disable versioning on the relevant table or tables. Tables owned by SYS cannot be version-enabled. Refer to the *Oracle9i Application Developer's Guide - Workspace Manager* for more information.

Examples

The following example enables versioning on the MIUUSA3 table.

```
Dim hdbc As Integer
hdbc = Server_Connect("ORAINET", "SRVR=TROYNY;UID=MIUSER;PWD=MIUSER")
Server hdbc Versioning ON Table "MIUUSA3"
```

or

```
Server hdbc Versioning ON History 1 Table "MIUUSA3"
```

The following example disables versioning on the MIUUSA3 table.

```
Dim hdbc As Integer
hdbc = Server_Connect("ORAINET", "SRVR=TROYNY;UID=MIUSER;PWD=MIUSER")
Server hdbc Versioning OFF Force ON Table "MIUUSA3"
```

See Also:

Server Create Workspace statement

Server Workspace Merge statement

Purpose

Applies changes to a table (all rows or as specified in the Where clause) in a workspace to its parent workspace in the database (Oracle 9i or later). You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Server Workspace Merge
   Table TableName
   [ Where WhereClause ]
   [ RemoveData { OFF | ON }]
   [ { Interactive | Automatic merge_keyword }]
```

TableName is the name (alias) of an open MapInfo table from an Oracle9i or later server. The table contains rows to be merged into its parent workspace.

WhereClause is a string that identifies the rows to be merged into the parent workspace.

merge_keyword is a keyword(s) that limit the **Automatic** merge behavior.

Description

This statement only applies to Oracle9i or later. All data that satisfies the *WhereClause* in *TableName* is applied to the parent workspace. Any locks that are held by rows being merged are released. If there are conflicts between the workspace being merged and its parent workspace, this operation provides user options on how to solve the conflict. The merge operation was executed only after all the conflicts were resolved. A table cannot be merged in the LIVE workspace (because that workspace has no parent workspace). A table cannot be merged or refreshed if there is an open database transaction affecting the table.

Refer to *Oracle9i Application Developer's Guide - Workspace Manager* for more information.

WhereClause identifies the rows to be merged into the parent workspace. The clause itself should omit the **Where** keyword. for example, '`MI_PRINX = 20`'. Only primary key columns can be specified in the **Where** clause. The **Where** clause cannot contain a subquery. If *WhereClause* is not specified, all rows in *TableName* are merged.

If **RemoveData** is set **ON**, the data in the table (as specified by *WhereClause*) in the child workspace will be removed. This option is permitted only if workspace has no child workspaces (that is, it is a leaf workspace). **OFF** (the default) does not remove the data in the table in the child workspace.

If there are conflicts between the workspace being merged and its parent workspace, the user must resolve conflicts first in order for merging to succeed. MapInfo Professional allows the user to resolve the conflicts first and then to perform the merging within the process. The **Interactive** and **Automatic** clauses let you control what happens when there is a conflict. These clauses have no effect if there is no conflict between the workspace being merged and its parent workspace.

If the **Interactive** clause is specified, MapInfo Professional displays the Conflict Resolution dialog box in the event of a merge conflict. The conflicts will be resolved one by one or all together based on user choices. After all the conflicts are resolved, the table is merged into its parent based on the user's choices.

ⓘ Due to a system limitation, this option is not available if the server is Oracle9i.

The following table shows the possible values for *merge_keyword* used with the **Automatic** setting.

merge_keyword value	Description
StopOnConflict	In the event of a conflict, MapInfo Professional will stop here. (This is also the default behavior if the statement does not include an **Interactive** clause or an **Automatic** clause.)
RevertToBase	In the event of a conflict, MapInfo Professional reverts to the original (base) values. (it causes the base rows to be copied to the child workspace but not to the parent workspace. However, the conflict is considered resolved; and when the child workspace is merged, the base rows are copied to the parent workspace too.) Note that BASE is ignored for insert-insert conflicts where a base row does not exist; in this case the **Automatic** clause must include UseParent or UseCurrent.)
UseCurrent	In the event of a conflict, MapInfo Professional uses the child workspace values.
UseParent	In the event of a conflict, MapInfo Professional uses the parent workspace values.

Examples

The following example merges changes to the GWMUSA2 table where MI_PRINX=60 in MIUSER to its parent workspace.

```
Server Workspace Merge
   Table "GWMUSA2"
   Where "MI_PRINX = 60"
   Automatic UseCurrent
```

See Also:

Server Workspace Refresh statement

Server Workspace Refresh statement

Purpose

Applies all changes made to a table (all rows or as specified in the **Where** clause) in its parent workspace to a workspace in the database (Oracle 9i or later). You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Server Workspace Refresh
   Table TableName
   [ Where WhereClause ]
   [ { Interactive | Automatic merge_keyword } ]
```

TableName is the name (alias) of an open MapInfo table from an Oracle9i or later server. The table contains rows to be refreshed using values from its parent workspace.

WhereClause identifies the rows to be refreshed from the parent workspace. The clause itself should omit the **WHERE** keyword.

merge_keyword is a string representing keyword(s) that limit the **Automatic** refresh behavior.

Description

This statement only applies to Oracle9i or later. It applies to workspace all changes in rows that satisfy the *WhereClause* in the table in the parent workspace from the time the workspace was created or last refreshed. If there are conflicts between the workspace being refreshed and its parent workspace, this operation provides user options on how to solve the conflict. The refresh operation is executed only after all the conflicts are resolved. A table cannot be refreshed in the LIVE workspace (because that workspace has no parent workspace). A table cannot be merged or refreshed if there is an open database transaction affecting the table.

Refer to the *Oracle9i Application Developer's Guide - Workspace Manager* for more information.

WhereClause identifies the rows to be refreshed from the parent workspace. The clause itself should omit the **WHERE** keyword. For example, `MI_PRINX = 20`. Only primary key columns can be specified in the **Where** clause. The **Where** clause cannot contain a subquery. If *WhereClause* is not specified, all rows in *TableName* are refreshed.

If there are conflicts between the workspace being refreshed and its parent workspace, the user must resolve conflicts first in order for refreshing to succeed. MapInfo Professional allows the user to resolve the conflicts first and then to perform the refreshing within the process. The **Interactive** and **Automatic** clauses let you control what happens when there is a conflict. These clauses has no effect if there is no conflict between the workspace being refreshed and its parent workspace.

If the **Interactive** clause is specified, MapInfo Professional displays the Conflict Resolution dialog box in the event of a refresh conflict. The conflicts will be resolved one by one or all together based on user choices. After all the conflicts are resolved, the table is refreshed into its parent based on the user's choices.

ⓘ Due to a system limitation, this option is not available if the server is Oracle9i.

The following table shows the possible values for *merge_keyword* used with the **Automatic** setting.

merge_keyword value	Description
StopOnConflict	In the event of a conflict, MapInfo Professional will stop here. (This is also the default behavior if the statement does not include an **Interactive** clause or an **Automatic** clause.)
RevertToBase	In the event of a conflict, MapInfo Professional reverts to the original (base) values. (it causes the base rows to be copied to the child workspace but not to the parent workspace. However, the conflict is considered resolved; and when the child workspace is merged to it parent, the base rows will be copied to the parent workspace.) Note that BASE is ignored for insert-insert conflicts where a base row does not exist; in this case the **Automatic** parameter must be followed by UseParent or UseCurrent.)
UseCurrent	In the event of a conflict, MapInfo Professional uses the child workspace values.
UseParent	In the event of a conflict, MapInfo Professional uses the parent workspace values.

Examples

The following example refreshes MIUSER by applying changes made to GWMUSA2 where MI_PRINX=60 in its parent workspace.

```
Server Workspace Refresh
   Table "GWMUSA2"
   Where "MI_PRINX = 60"
   Automatic UseParent
```

See Also:

Server Workspace Merge statement

SessionInfo( ) function

Purpose

Returns various pieces of information about a running session of MapInfo Professional. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

SessionInfo(*attribute*)

attribute is an integer code indicating which session attribute to query.

Return Value

String

Description

The **SessionInfo( )** function returns information about MapInfo Professional's session status. The attribute can be any of the codes listed in the table below. The codes are defined in MAPBASIC.DEF.

attribute code	ID	Return Value
SESSION_INFO_COORDSYS_CLAUSE	1	String result that indicates a session's CoordSys clause.
SESSION_INFO_DISTANCE_UNITS	2	String result that indicates a session's distance units.
SESSION_INFO_AREA_UNITS	3	String result that indicates a session's area units.
SESSION_INFO_PAPER_UNITS	4	String result that indicates a session's paper units.

Error Conditions

ERR_FCN_ARG_RANGE (644) error generated if an argument is outside of the valid range.

Example

```
Include "mapbasic.def"
print SessionInfo(SESSION_INFO_COORDSYS_CLAUSE)
```

Set Adornment statement

Purpose

Modifies the adornment created by the **Create Adornment statement**. You can change the adornment position, its dimensions, and specify a border for it. For the scale bar adornment, you can change its display style, the bar type, units, dimensions, and display a cartographic scale with it.

You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Set Adornment
Window window_id
[ Type adornment_type ]
[ Position {
   [ Fixed [ ( x, y ) [ Units paper_units ] ] ] |
   [ win_position [ Offset (x, y) ] [Units paper_units ] ]
} ]
[ Layout Fixed Position { Frame | Geographic } ]
[ Size [ Width win_width ] [ Height win_height  ] [ Units paper_units ] ]
[ Background [ Brush ... ] [ Pen ... ] ]
[ < SCALEBAR_CLAUSE > ]
```

Where *SCALEBAR_CLAUSE is:*

```
   [ BarType type ]
   [ Ground Units distance_units ]
   [ Display Units paper_units ]
   [ BarLength paper_length ]
   [ BarHeight paper_height ]
   [ BarStyle [ Pen .... ] [ Brush ... ] [ Font ... ] ]
   [ Scale [ { On | Off } ] ]
```

adornment_type can be **scale bar**.

Position can be **Fixed** relative to the mapper upper left regardless of the size of the mapper, or relative to some anchor point on the mapper specified by *win_position*.

(*x, y*) in the **Fixed** clause is position measured from the upper left of the mapper window, which is (0, 0). Using this version of adornment placement, the adornment will be at that position in the mapper as the mapper resizes. For example, a position of (3, 3) inches would be toward the bottom right of a small sized mapper but in the middle of a large sized mapper. As the mapper changes size, the adornment will try to remain completely within the displayed mapper.

paper_units defaults to the MapBasic Paper Unit (see **Set Paper Units statement**).

win_position specify one of the following codes; codes are defined in MAPBASIC.DEF.

```
ADORNMENT_INFO_MAP_POS_TL (0)
ADORNMENT_INFO_MAP_POS_TC (1)
ADORNMENT_INFO_MAP_POS_TR (2)
ADORNMENT_INFO_MAP_POS_CL (3)
ADORNMENT_INFO_MAP_POS_CC (4)
```

```
ADORNMENT_INFO_MAP_POS_CR (5)
ADORNMENT_INFO_MAP_POS_BL (6)
ADORNMENT_INFO_MAP_POS_BC (7)
ADORNMENT_INFO_MAP_POS_BR (8)
```

Offset is the amount the adornment will be offset from the mapper when using one of the docked win_positions.

(x, y) in the **Offset** clause is measured from the anchor position. For example, if the *win_position* is ADORNMENT_INFO_MAP_POS_TL (top left), then the x is to the right and the y is down. If the *win_position* is ADORNMENT_INFO_MAP_POS_BR, then the x position is left and the y position is up. In the center left (ADORNMENT_INFO_MAP_POS_CL) and center right (ADORNMENT_INFO_MAP_POS_CR), the y offset is ignored. In the center position (ADORNMENT_INFO_MAP_POS_CC), the offset is ignored completely (both x and y). In the top center (ADORNMENT_INFO_MAP_POS_TC) and bottom center (ADORNMENT_INFO_MAP_POS_BC) positions, the x offset is ignored. For ADORNMENT_INFO_MAP_POS_ defines, see *win_position.*

Layout Fixed Position determines how an adornment is positioned in a layout when the adornment is using Fixed positioning. If this is set to **Geographic**, then the adornment is placed on the same geographic place on the map frame in the layout as it is in the mapper. If the layout frame changes size, then the adornment will move relative to the frame to match the geographic position. If this is set to **Frame**, then the adornment will remain at a fixed position relative to the frame, as designated in the **Position** clause. If the Position clause positions the adornment at (1.0, 1.0) inches, then the adornment will be placed 1 inch to the left and one inch down from the upper left corner of the frame. Changing the size of the frame will not change the position of the adornment. The default is **Geographic**.

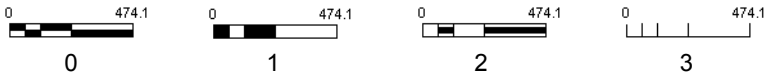
win_width and *win_height* define the size of the adornment. MapInfo Professional ignores these parameters if this is a scale bar adornment, because scale bar adornment size is determined by scale bar specific items, such as *BarLength*.

Brush is a valid **Brush clause**. Only Solid brushes are allowed. While values other than solid are allowed as input without error, the type is always forced to solid. This clause is used only to provide the background color for the adornment.

Pen is a valid **Pen clause**. Due to window clipping (the adornment is a window within the mapper), Pen widths other than 1 may not display correctly. Also, Pen styles other than solid may not display correctly. This clause is designed to turn on (solid) or off (hollow) and set the color of the border of the adornment.

type specify one of the following codes; codes are defined in MAPBASIC.DEF.

```
SCALEBAR_INFO_BARTYPE_CHECKEDBAR   (0)
SCALEBAR_INFO_BARTYPE_SOLIDBAR     (1)
SCALEBAR_INFO_BARTYPE_LINEBAR      (2)
SCALEBAR_INFO_BARTYPE_TICKBAR      (3)
```



***0** Check Bar, **1** Solid Bar, **2** Line Bar, or **3** Tick Bar*

distance_units a unit of measure that the scale bar is to represent:

distance value	Unit Represented
“ch”	chains
“cm”	centimeters
“ft”	feet (also called International Feet; one International Foot equals exactly 30.48 cm)
“in”	inches
“km”	kilometers
“li”	links
“m”	meters
“mi”	miles
“mm”	millimeters
“nmi”	nautical miles (1 nautical mile represents 1852 meters)
“rd”	rods
“survey ft”	U.S. survey feet (used for 1927 State Plane coordinates; one U.S. Survey Foot equals exactly 12/39.37 meters, or approximately 30.48006 cm)
“yd”	yards

paper_units defaults to the MapBasic Paper Unit (see **Set Paper Units statement**).

paper_length a value in *paper_units* to specify how long the scale bar will be displayed. Specify the length of the scale bar to a maximum of 34 inches or 86.3 cm on the printed map.

paper_height a value in *paper_units* to specify how tall the scale bar will be displayed. Specify height of the adornment to a maximum of 44 inches or 111.76cm on the printed map.

Scale set to **On** to include a representative fraction (RF) with the scale bar. (In MapInfo Professional, a map scale that does not include distance units, such as 1:63,360 or 1:1,000,000, is called a **cartographic scale**.)

Font is a valid **Font clause**.

Example

```
set adornment
   window 261727232
   type scalebar
   position 6
   background Brush (2,16777215,16777215)  Pen (1,2,0)
```

```
bartype 0 ground units "km" display units "cm"
barlength 0.978478 barheight 0.078740
barstyle Pen (1,2,0)  Brush (2,0,16777215)  Font ("Arial",0,8,0)
scale on
```

See Also:

Create Adornment statement

Set Application Window statement

Purpose

Sets which window will be the parent of dialog boxes that are yet to be created. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

`Set Application Window` *HWND*

HWND is an integer window handle, which identifies a window.

Description

This statement sets which window is the application window. Once you set the application window, all MapInfo Professional dialog boxes have the application window as their parent. This statement is useful in "integrated mapping" applications, where MapInfo Professional windows are integrated into another application, such as a Visual Basic application.

In your Visual Basic program, after you create a MapInfo Object, send MapInfo Professional a **Set Application Window statement**, so that the Visual Basic application becomes the parent of MapInfo Professional dialog boxes. If you do not issue the **Set Application Window statement**, you may find it difficult to coordinate whether MapInfo Professional or your Visual Basic program has the focus.

Issuing the command `Set Application Window 0` will return MapInfo Professional to its default state. This statement re-parents dialog box windows. To re-parent document windows, such as a Map window, use the **Set Next Document statement**.

If you specify the *HWND* as an explicit hexadecimal value, you must place the characters `&H` at the start of the *HWND*; otherwise, MapInfo Professional will try to interpret the expression as a decimal value. (This situation can arise, for example, when a Visual Basic program builds a command string that includes a **Set Application Window statement**.)

For more information on integrated mapping, see the *MapBasic User Guide*.

See Also:

Set Next Document statement

Set Area Units statement

Purpose

Sets MapBasic's default area unit. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Set Area Units *area_name*

area_name is a string representing the name of an area unit (for example, “acre”).

Description

The **Set Area Units** statement sets MapInfo Professional's default area unit of measure. This dictates the area unit used within MapInfo Professional's SQL Select dialog box. By default, MapBasic uses square miles as an area unit; this unit remains in effect unless a **Set Area Units** statement is issued. The *area_name* parameter must be one of the string values listed in the table below:

Unit Name	Unit Represented
“acre”	acres
“hectare”	hectares
“perch”	perches
“rood”	roods
“sq ch”	square chains
“sq cm”	square centimeters
“sq ft”	square feet
“sq in”	square inches
“sq km”	square kilometers
“sq li”	square links
“sq m”	square meters
“sq mi”	square miles
“sq mm”	square millimeters
“sq rd”	square rods

Unit Name	Unit Represented
“sq survey ft”	square survey feet
“sq yd”	square yards

Example

```
Set Area Units "acre"
```

See Also:

Area( ) function, **Set Distance Units statement**

Set Browse statement

Purpose

Modifies an existing Browser window. You can issue this statement from the MapBasic window in MapInfo Professional.

Syntax

```
Set Browse
   [ Window window_id ]
   [ Grid { On | Off } ]
   [ Row row_num ]
   [ Column column_num ]
   [ Columns Resize ]
```

window_id is the integer window identifier of a Browser window or a Redistricter window.

row_num is a SmallInt value, one or larger; one represents the first row in the table.

column_num is a SmallInt value, zero or larger; zero represent the table's first column.

Description

The **Set Browse** statement controls the settings of an existing Browser window. If no *window_id* is specified, the statement affects the topmost Browser window.

The optional **Window** clause lets you specify which document window to use. If a *window_id* is not specified, then it searches for the most recently used Browser window or Redistricter window. If neither Redistricter nor Browser is the front-most window, but both exist, then Set Browse finds the Browser window instead of the Redistricter window. To specify which window type to use, either include the *window_id* with the Set Browse statement or make the Redistricter window the front-most window before calling Set Browse.

The optional **Grid** clause displays (turns on) or does not display (turns off) the grid lines in a Browser window.

The optional **Row** and **Column** clauses let you specify which row should be the topmost row in the Browser, and which column should be the leftmost column in the Browser.

The optional **Columns** clause lets you set column resizing based on the width of the column header (title) and the contents that are in view. On first display, the Browser window automatically resizes columns to completely contain the data that is visible. When scrolling vertically, the Browser window does not automatically adjust the column width for the new data in view. You must set the Columns clause to make this happen. After recalculating column width, the width does not change while scrolling—columns do not resize to the new data in view. If the user manually resizes a column, then its width does not change.

To change the width, height, or position of a Browser window, use the **Set Window statement**.

Example

```
Dim i_browser_id As Integer
Open Table "world"
Browse * From world
i_browser_id = FrontWindow( )
Set Browse Window i_browser_id Row 47
```

See Also:

Browse statement, **Set Window statement**

Set Buffer Version statement

Purpose

Sets MapInfo Professional to process Buffer operations using an older algorithm that was in use before MapInfo Professional 9.5.1.

Syntax

Set Buffer Version *version_num*

version_num a value of either 950 or 951. A version number higher than 951 generates an error., and a version number lower than 950 uses the older (version 9.5 and later) algorithm.

Description

MapInfo Professional 9.5.1 introduced a new algorithm to yield self-intersecting polygons when processing data. However, Oracle does not consider these objects valid, so it is unable to return these objects to MapInfo Professional or process them. To upload, store, retrieve, or use this data in Oracle, you must find the self-intersecting polygons in the data and remove them, or run the Buffer operations using the older algorithm (in use before MapInfo Professional 9.5.1).

To determine if your data contains self-intersecting polygons, in MapInfo Professional, run your table through the **Check Regions** process that is accessible from the **Objects** menu. To remove unwanted self-intersecting polygons, run a **Clean** operation on the table. Note that Check Regions and Clean take some time to process large tables.

To process your data using the older Buffer algorithms, add this MapBasic command to a workspace, such as startup.wor, or to a MapBasic window at runtime. You cannot add this commands to a MapBasic application directly because it will not compile. However, you can issue it using a Run Command statement within a MapBasic application.

Example

To use Buffer operations from MapInfo Professional 9.5 or earlier, use the MapBasic command:

```
Set Buffer Version 950
```

To use Buffer operations from MapInfo Professional 9.5.1 or later, use the MapBasic command:

```
Set Buffer Version 951
```

See Also:

Set Combine Version statement

Set Cartographic Legend statement

Purpose

Sets redraw functionality on or off, refreshes, sets the orientation to portrait or landscape, selects small or large sample legend sizes, or changes the frame order of an existing cartographic legend created with the **Create Cartographic Legend statement**. (To change the size, position, or title of the Legend window, use the **Set Window statement**.) You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Set Cartographic Legend
   [ Window legend_window_id ]
   Redraw { On | Off }
```

or

```
Set Cartographic Legend
   [ Window legend_window_id ]
   [ Refresh ]
   [ Portrait [ Columns number_of_columns ] |
     Landscape [Lines number_of_lines ]]
   [ Align ]
   [ Style Size { Small | Large }]
   [ Frame Order { frame_id, frame_id, frame_id, ... } ]
```

legend_window_id is an integer window identifier which you can obtain by calling the **FrontWindow( ) function** and the **WindowID( ) function**.

frame_id is the ID of the frame on the legend. You cannot use a layer name. For example, three frames on a legend would have the successive IDs 1, 2, and 3.

number_of_columns specifies the width of the legend.

number_of_lines specifies the height of the legend.

Description

The **Set Cartographic Legend** statement allows you to set redraw functionality on or off, refresh, set the orientation to portrait or landscape, select small or large sample legend sizes, or change the frame order of an existing cartographic legend created with the **Create Cartographic Legend statement**.

If a **Window** clause is not specified MapInfo Professional will use the topmost legend window.

Other clauses to are not allowed if **Redraw** is used.

The **Refresh** keyword causes the Legend window to refresh. Tables for refreshable frames will be re-scanned for styles. The **Portrait** or **Landscape** keywords cause frames in the Legend window to be laid out in the appropriate order.

Align causes styles and text across all frames, regardless of whether the Legend window is in portrait, landscape, or custom layout, to be re-aligned.

The **Frame Order** clause reorders the frames in the legend.

Example

If you used the **Create Cartographic Legend statement** to select large sample legend sizes, the following example will refresh the foreground legend window to show large legend sizes:

```
Set Cartographic Legend Window WindowID(0) Refresh Portrait Align Style
Size Large
```

See Also:

Add Cartographic Frame statement, **Alter Cartographic Frame statement**, **Create Cartographic Legend statement**, **Remove Cartographic Frame statement**

Set Combine Version statement

Purpose

Sets MapInfo Professional to process Combine operations using an older algorithm that was in use before MapInfo Professional 9.5.1.

Syntax

Set Combine Version *version_num*

version_num a value of either 950 or 951. A version number higher than 951 generates an error., and a version number lower than 950 uses the older (version 9.5 and older) algorithm.

Description

MapInfo Professional 9.5.1 introduced a new algorithm to yield self-intersecting polygons when processing data. However, Oracle does not consider these objects valid, so it is unable to return these objects to MapInfo Professional or process them. To upload, store, retrieve, or use this data in Oracle, you must find the self-intersecting polygons in the data and remove them, or run the Combine operations using the older algorithm (in use before MapInfo Professional 9.5.1).

To determine if your data contains self-intersecting polygons, in MapInfo Professional, run your table through the **Check Regions** process that is accessible from the **Objects** menu. To remove unwanted self-intersecting polygons, run a **Clean** operation on the table. Note that Check Regions and Clean take some time to process large tables.

To process your data using the older Combine algorithms, add this MapBasic command to a workspace, such as startup.wor, or to a MapBasic window at runtime. You cannot add this commands to a MapBasic application directly because it will not compile. However, you can issue it using a Run Command statement within a MapBasic application.

Example

To use Combine operations from MapInfo Professional 9.5 or earlier, use the MapBasic command:

```
Set Combine Version 950
```

To use Combine operations from MapInfo Professional 9.5.1 or later, use the MapBasic command:

```
Set Combine Version 951
```

See Also:

Set Buffer Version statement

Set Command Info statement

Purpose

Stores values in memory; other procedures can call the **CommandInfo( ) function** to retrieve the values. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Set Command Info *attribute* **To** *new_value*

attribute is a code used by the **CommandInfo( ) function**, such as CMD_INFO_ROWID (2).

new_value is a new value; its data type must match the data type that is associated with the *attribute* code (for example, if you use CMD_INFO_ROWID (2), specify a positive integer for *new_value*).

Description

Ordinarily, the CommandInfo( ) function returns values that describe recent system events. The **Set Command Info** statement stores a value in memory, so that subsequent calls to the CommandInfo( ) function returns the value that you specified, instead of returning information about system events.

Example

Suppose your program has a **SelChangedHandler procedure**. Within the procedure, the following function call determines the ID number of the row that was selected or de-selected:

```
CommandInfo(CMD_INFO_ROWID)
```

When MapInfo Professional calls the **SelChangedHandler procedure** automatically, MapInfo Professional initializes the data values read by the CommandInfo( ) function. Now suppose you want to call the **SelChangedHandler procedure** explicitly, using the Call statement—perhaps for debugging purposes. Before you issue the **Call statement**, issue the following statement to "feed" a value to the **CommandInfo( ) function**:

```
Set Command Info CMD_INFO_ROWID To 1
```

See Also:

CommandInfo( ) function, **Set Handler statement**

Set Connection Geocode statement

Purpose

Configures a connection to a remote service with options for geocoding. The connection needs to have been already created using the **Open Connection statement**. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Set Connection connection_number Geocode
   [ Batch Size batch_size ]
   [ ResultCode MarkMultiple [ On | Off ] ]
   [ MixedCase [ On | Off ] ]
   [ Match
      [ StreetName [ On | Off ] [,]]
      [ StreetNumber [ On | Off ] [,]]
      [ Municipality [ On | Off ] [,]]
      [ CountrySubdivision[ On | Off ] [,]]
      [ CountrySecondarySubdivision [ On | Off ] [,] ]
      [ PostalCode [ On | Off ] [,] ]
      [ MunicipalitySubdivision [ On | Off ] [,] ]
      | All [ On | Off ] [,]]]
   [ Fallback
      [ Geographic [ On | Off ] [,] ]
      [ PostalCode [ On | Off ] [,] ] ]
```

```
[ Dictionary
   [ All | Address | User | Prefer ( Address | User ) ]
[ Offset [ [ Center offset_num_expr Units distance_unit_name ]
   [ End offset_num_expr Units distance_unit_name ]
[ PassThrough name value, name value, ... ]
```

connection_number is a number that specifies the connection handle created using the **Open Connection statement**.

batch_size is an integer expression that specifies the maximum number of records that are sent to the service at one time.

offset_num_expr is a numeric expression which specifies the offset from either the corner (end) or the center of the street. These values are just to offset the point returned from the center of the street or the end of the street respectively.

distance_unit_name is a String that represents the units in which *offset_num_expr* are expressed.

name is a the name part of the parameter pair that is passed through to the geocoding service.

value is the value part of the parameter pair that is passed through to the geocoding service.

Description

The **Set Connection** statement is used to assign geocode preferences already defined so that each geocode request does not need to reiterate the preferences defined. A **Set Connection** statement is composed of six different sub-clauses. These clauses are **Batch**, **Match**, **Fallback**, **Dictionary**, **Offset**, and **PassThrough**.

Batch Clause

The **Batch** clause determines the maximum number of records that are sent to the service at one time. This allows you to optimize the processing of records to balance the amount of time needed by the local computer and the external service. If this number is high, you will have longer local downtime while the service processes the records. If this number is low, the user has a better opportunity to cancel the request. Once a batch is sent to the service it cannot be cancelled. If you cancel the command, any remaining batches are not processed

Match Clause

If a specific **Match** is set, the geocoder only considers inputs that fully match the name in the geocode data as a close match. For example, if **Match StreetName** is **On**, the geocoder does not regard street name inputs that do not match the name in the geocode data, as close matches.

For the individual preferences under **Match** the default is **On**. So if a particular preference is stated, it is the same as setting it to **On**. For example, `Set Connection connectionHandle Geocode Match Municipality` is equivalent to `Set Connection connectionHandle Geocode Match Municipality On`.

Match StreetName indicates whether or not the street name should be relaxed when trying to match.

Match StreetNumber indicates whether or not the address number should be relaxed when trying to match.

Match Municipality indicates whether or not the municipality should be relaxed when trying to match.

Match CountrySubdivision indicates whether or not the country subdivision (usually a state or province) should be relaxed when trying to match.

Match CountrySecondarySubdivision indicates whether or not the country secondary subdivision should be relaxed when trying to match.

Match PostalCode indicates whether or not the postal code should be relaxed when trying to match.

Match MunicipalitySubdivision indicates whether or not the municipality subdivision should be relaxed when trying to match.

Match All sets all the match properties to **On** or **Off** . Note this can be used in combination with other match options. For example, `Match All On, Match PostalCode Off` turns all the match parameters on and just the postal code match is turned off.

FallBack Clause

Fallback Geographic indicates whether or not to geocode to the geographic centroid for the input address if a street level geocode cannot be performed. This value is only appropriate when your address to be geocoded includes a street address. If the record does not contain a street address, this value has no impact.

Fallback Postal indicates whether or not to geocode to the postal centroid if a street level geocode cannot be performed.

Dictionary Clause

Dictionary indicates the combination of MapMarker address dictionary and configured user dictionaries to use during the geocode process. The five possible choices for the **Dictionary** clause are:

- **Dictionary All** means use both the user and address dictionaries.
- **Dictionary Address** means use only the address dictionary.
- **Dictionary User** means use only the User dictionary.
- **Dictionary Prefer Address** means use both dictionaries and prefer the address dictionary.
- **Dictionary Prefer User** means use both dictionaries and prefer the User dictionary.

Offset Clause

Offset End indicates the distance that a point location is adjusted from a street corner.

Offset Center indicates the distance that a point location is adjusted from a street center line.

Units is a String that describes the units in which **Offset Center** and **Offset End** are measured. See **Set Distance Units statement** for the list of available unit names.

PassThrough Clause

PassThrough is a set of name/value pairs that are sent to the geocoder. These are pairs are geocode service specific and are documented by the particular geocode service.

Example

The following example sets a connection to a geocoder with some match options turned on.

```
set connection MapMarkerHandle1 geocode match streetname, streetnumber,
municipality, municipalitysubdivision, postalcode, countrysubdivision
```

The following example adds a **PassThrough** clause to a **Set Connection** statement. This particular example turns on CASS certified results in the US.

```
PassThrough "KEY_CASS_RULES"  "true"
```

See Also:

Geocode statement, **Open Connection statement**

Set Connection Isogram statement

Purpose

Allows a user to set options for an Isogram connection. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Set Connection connection_handle Isogram
   [ Banding [ On | Off ] ]
   [ MajorRoadsOnly [ On | Off ] ]
   [ MaxOffRoadDistance distance_value Units distance_units ]
   [ ReturnHoles [ On | Off ] ] [ MajorPolygonOnly [ On | Off ] ]
   [ SimplificationFactor simplification ]
   [ PointsOnly [ On | Off ] ]
   [ DefaultAmbientSpeed ambient_speed
     Units distance_units Per time_units ]
   [ DefaultPropagationFactor propagation_factor ]
   [ Batch Size batch_size ]
```

connection_handle is a the number of the connection returned from the **Open Connection statement**.

distance_value is a Float value that specifies the maximum distance travel will be allowed to go off roads in the network.

distance_units is a string that specifies the distance units in which the specific *distance_value* is expressed. For a complete list of valid strings of distance units, see **Set Distance Units statement**.

simplification is a Float value that controls the density of nodes in the output region as a percentage. The value can be from 0 to 1 inclusive.

ambient_speed is a numeric value specifying the default ambient speed. The number is expressed in *distance_units* and *time_units*.

time_units is a string that specifies time units. Valid values are “hr”, “min” and “sec”.

propagation_factor is a Float value specifying the default propagation factor. The value can be from 0 to 1 inclusive.

batch_size is an integer expression that specifies the size of each batch that is sent to the service. The default is 2 and the maximum limit is 50.

Description

The **Set Connection Isogram** statement configures the connection that is to be used for creating an Isogram object (using the **Create Object statement**).

Banding applies only if multiple distances or times are specified in the Isogram operation. If **On**, the regions returned for one point will not overlap. The smaller region is cut out of the result. Thus it represents the time or distance from the smaller region edge to its edge. For example, if 10, 20, and 30 minutes Isograms are requested, the 20 minute Isogram represents the areas accessible from 10 to 20 minutes and the 30 minute Isogram the area from 20 to 30 minutes. If **Off**, all the regions cover the area accessible from 0 to the time or distance specified.

MajorRoadsOnly determines whether or not only major roads are used in the calculation of the Isogram. Isogram generation is substantially quicker when using **MajorRoadsOnly**.

MaxOffRoadDistance specifies the maximum distance travel is allowed to go off roads.

ReturnHoles indicates whether or not holes should be returned in the resulting region.

MajorPolygonOnly indicates that the Region returned has only one outer polygon.

SimplificationFactor specifies the reduction factor for polygon complexity. The simplification factor indicates what percentage of the original points should be returned or that the resulting polygon should be based on. The polygon or set of points may contain many points. The simplification factor is a float number between 0.01 and 1.0 (1 being 100% and 0.01 being 1%). Lower numbers mean fewer points in the region and therefore faster transmission times across the Internet connection. The default value is 0.05.

PointsOnly specifies whether or not records that contain non-point objects should be skipped.

DefaultAmbientSpeed is used only when specifying time. A syntax example is:

```
DefaultAmbientSpeed  12 "mi" Per "hr"
```

DefaultPropagationFactor determines the off-road network percentage of the remaining cost (distance) for which off network travel is allowed when finding the maximum distance boundary. Roads not identified in the network can be driveways or access roads, among others. The propagation factor is a percentage of the cost used to calculate the distance between the starting point and the maximum distance. **DefaultPropagationFactor** is used only for Distances.

The default value for this property is 0.16.

The acceptable range is between 0.01 and 1.

Batch Size sets the number of records to send to the server to be processed at once. This may affect performance and responsiveness. If a large request is sent it will take longer for the Isogram to be returned and therefore longer for MapInfo Professional to respond to cancel requests and update the Progress Bar dialog box. A lower number improves responsiveness of the command and lowers the chance of a time-out and service failure. The default value is 2.

Example

The following example shows a Set Connection Isogram statement.

```
Set Connection iConnect Isogram
Banding On MajorRoadsOnly On MaxOffRoadDistance 2 Units "mi"
ReturnHoles On MajorPolygonOnly On SimplificationFactor .05
DefaultAmbientSpeed 50 Units "mi" Per "hr" DefaultPropagationFactor .2
Batch Size 2 Point On
```

See Also:

Create Object statement, **Open Connection statement**

Set CoordSys statement

Purpose

Sets the coordinate system used by MapBasic. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Set CoordSys...
```

CoordSys… is a coordinate system clause.

Description

The **Set CoordSys** statement sets MapBasic's coordinate system. By default, MapBasic uses a Longitude/Latitude coordinate system. This means that when geographic functions (such as the **CentroidX( ) function** and the **ObjectNodeX( ) function**) return x- or y-coordinate values, the values represent longitude or latitude degree measurements by default. A MapBasic program can issue a **Set CoordSys** statement to specify a different coordinate system; thereafter, values returned by geographic functions will automatically reflect the new coordinate system.

The **Set CoordSys** statement does not affect a Map window. To set a Map window's projection or coordinate system, you must issue a **Set Map…CoordSys** statement.

The CoordSys clause has optional **Table** and **Window** sub-clauses that allow you to reference the coordinate system of an existing table or window. See **CoordSys clause** for more information.

Example

The following **Set CoordSys** statement would set the coordinate system to an un-projected, Earth-based system.

```
Set CoordSys Earth
```

The next **Set CoordSys** statement would set the coordinate system to an Albers equal-area projection.

```
Set CoordSys Earth
Projection 9,7,"m",-96.0,23.0,20.0, 60.0, 0.0, 0.0
```

The **Set CoordSys** statement below prepares MapBasic to work with objects from a Layout window. You must use a Layout coordinate system before querying or creating Layout objects.

```
Set CoordSys Layout Units "in"
```

> ⓘ Once you have issued the **Set CoordSys Layout** statement, the MapBasic program will continue to use the Layout coordinate system until you explicitly change the coordinate system back. Subsequently, you should issue a **Set CoordSys Earth** statement before attempting to query or create any objects on Earth maps.

See Also:

CoordSys clause, **Set Area Units statement**, **Set Distance Units statement**, **Set Paper Units statement**

Set Date Window statement

Purpose

Displays a date window that converts two-digit input into four-digit years. It also allows you to change the default to one that best suits your data. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Set Date Window { *nYear* | **Off** }

nYear is a SmallInt from 0 to 99 that specifies the year above which is assigned to the previous century (19*xx*) and below which is assigned to the next century (20*xx*). (For example, by specifying *nYear* as 70, a 2-digit year of 70 and above corresponds to the years 1970-1999, and a 2-digit year of 69 and below correspond to the years 2000-2069.)

Off turns date windowing off. Two-digit years will be converted to the current century (based on system time/calendar settings).

Description

From the MapBasic window, the session setting will be initialized from the Preference setting and updated when the preference is changed. Running the **Set Date Window** statement from the MapBasic window will change the behavior of input, but will not update the System Preference that is saved when MapInfo Professional exits.

The session setting is affected by running **Set Date Window** in the MapBasic window, in any workspace file including STARTUP.WOR, and any integrated mapping application that runs the command via the MapInfo Professional application interface.

When the **Set Date Window** command is run from within a MapBasic program (also as **Run Command statement**) only the program's local context is updated with the new setting. The session and preference settings remain unchanged. The program's local context is initialized from the session setting. This is similar to how number and date formatting works. They are set/accessed per program if a program is running, otherwise they set/access global settings.

Enter a number from 0-99. The number you enter displays in the statements below the prompt that indicate whether the date will display with the prefix 19 or 20.

For example if you enter the number 50, the statements will indicate that:

Years entered as 00-49 become 2000-2049.

Years entered as 50-99 become 1950-1999.

Example

In the following example the variable Date1 = 19890120, Date2 = 20101203 and MyYear = 1990.

```
DIM Date1, Date2 as Date
DIM MyYear As Integer
   Set Format Date "US"
   Set Date Window 75
      Date1 = StringToDate("1/20/89")
      Date2 = StringToDate("12/3/10")
      MyYear = Year("12/30/90")
```

See Also:

DateWindow( ) function

Set Datum Transform Version statement

Purpose

This statement allows user to switch between using old and new datum conversion. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Set Datum Transform Version version_number
```

version_number is an integer value. If version_number is equal or more than 800, than updated datum transformation algorithms are used. Otherwise old algorithm is used.

Description

By default, MapInfo Professional uses updated datum conversion algorithms. These algorithms are more in line with algorithms used by other software packages and it is recommended not to changed version from default. In previous versions of MapInfo Professional we used optimized for speed algorithms and our results were slightly different from other software packages/tools

Set Digitizer statement

Purpose

Establishes the coordinates of a paper map on a digitizing tablet; also turns Digitizer Mode on or off. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax 1

```
Set Digitizer
   ( mapx1, mapy1 ) ( tabletx1, tablety1 ) [ Label name ] ,
   ( mapx2, mapy2 ) ( tabletx2, tablety2 ) [ Label name ]
   [, ... ]
   CoordSys...
   [ Units... ]
   [ Width tabletwidth ]
   [ Height tabletheight ]
   [ Resolution xresolution, yresolution ]
   [ Button click_button_num, double_click_button_num ]
   [ Mode { On | Off } ]
```

Syntax 2

```
Set Digitizer Mode { On | Off }
```

mapx parameters specify East-West Earth positions on the paper map.

mapy parameters specify North-South Earth positions on the paper map.

tabletx parameters specify tablet right-left positions corresponding to the *mapx* values.

tablety parameters specify tablet up-down positions corresponding to the *mapy* values.

name is and optional label for the control points.

The **CoordSys clause** specifies the coordinate system used by the paper map.

click_button_num is the number of the puck button that simulates a click action.

double_click_button_num is the number of the puck button that simulates a double-click.

Description

The **Set Digitizer** statement controls the same settings as the Digitizer Setup dialog box in MapInfo Professional's Map menu. These settings relate to a specific paper map that the user has attached to the tablet. The **Set Digitizer** statement does not relate to other digitizer setup options, such as communications port or baud rate settings; those settings must be configured outside of a MapBasic application.

The **Set Digitizer** statement tells MapInfo Professional the coordinate system used by the paper map, and specifies two or more control points. Each control point consists of a map coordinate pair (for example, longitude, latitude) followed by a tablet coordinate pair. The tablet coordinate pair represents the position on the tablet corresponding to the specified map coordinates. Tablet coordinates represent the distance, in native digitizer units (such as thousandths of an inch), from the point on the tablet to the tablet's upper left corner.

The CoordSys clause specifies the coordinate system used by the paper map. For more details, see **CoordSys clause**.

The **Set Digitizer** statement ignores the **Bounds** portion of the CoordSys clause.

The **Width**, **Height**, and **Resolution** clauses are for MapInfo Professional internal use only. MapInfo Professional stores these clauses, when necessary, in workspaces. MapBasic programs do not need to specify these clauses.

Turning Digitizer Mode On or Off

Once the digitizer is configured, the user can toggle Digitizer Mode on or off by pressing the D key. To toggle Digitizer Mode from a MapBasic program, specify

```
Set Digitizer Mode On
```

or

```
Set Digitizer Mode Off
```

To determine whether Digitizer Mode is currently on or off, call **SystemInfo**(SYS_INFO_DIG_MODE), which returns TRUE if Digitizer Mode is on.

When Digitizer Mode is on and the active window is a Map window, the digitizer cursor (a large crosshair) appears in the window; the digitizer and the mouse have separate cursors.

If Digitizer Mode is off, or if the active window is not a Map window, the digitizer cursor does not display and the digitizer controls the mouse cursor (if your digitizer driver provides mouse emulation).

See Also:

CoordSys clause, **SystemInfo( ) function**

Set Distance Units statement

Purpose

Sets the distance unit used for subsequent geographic operations, such as the **Create Object statement**. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Set Distance Units *unit_name*

unit_name is the name of a distance unit (for example, “m” for meters).

Description

The **Set Distance Units** statement sets MapBasic's linear unit of measure. By default, MapBasic uses a distance unit of “mi” (miles); this distance unit remains in effect unless a **Set Distance Units** statement is issued. Some MapBasic statements take parameters representing distances. For example, the Create Object statement’s **Width** clause may or may not specify a distance unit. If the **Width** clause does not specify a distance unit, the Create Object statement uses the distance units currently in use (either miles or whatever units were set by the latest **Set Distance Units** statement).

The *unit_name* parameter must be one of the values from the table below:

unit_name value	Unit Represented
“ch”	chains
“cm”	centimeters
“ft”	feet (also called International Feet; one International Foot equals exactly 30.48 cm)
“in”	inches
“km”	kilometers
“li”	links
“m”	meters
“mi”	miles
“mm”	millimeters
“nmi”	nautical miles (1 nautical mile represents 1852 meters)
“rd”	rods

unit_name value	Unit Represented
“survey ft”	U.S. survey feet (used for 1927 State Plane coordinates; one U.S. Survey Foot equals exactly 12/39.37 meters, or approximately 30.48006 cm)
“yd”	yards

Example

```
Set Distance Units "km"
```

See Also:

Distance( ) function, **ObjectLen( ) function**, **Set Area Units statement**, **Set Paper Units statement**

Set Drag Threshold statement

Purpose

Sets the length of the delay that the user experiences when dragging graphical objects. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Set Drag Threshold *pause*

pause is a floating-point number representing a delay, in seconds; default value is 1.0.

Description

When a user clicks on a map object to drag the object, MapInfo Professional makes the user wait. This delay prevents the user from dragging objects accidentally. The **Set Drag Threshold** statement sets the duration of the delay.

Example

```
Set Drag Threshold 0.25
```

Set Event Processing statement

Purpose

Temporarily turns event processing on or off, to avoid unnecessary screen updates. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Set Event Processing { **On** | **Off** }

Description

The **Set Event Processing** statement lets you suspend, then resume, processing of system events.

If several successive statements modify a window, MapInfo Professional may redraw that window once for each MapBasic statement. Such multiple window redraws are undesirable because they make the user wait. To eliminate unnecessary window redraws, you can issue the statement:

```
Set Event Processing Off
```

Then issue all statements that apply to window maintenance (for example, the **Set Map statement**), and then issue the statement:

```
Set Event Processing On
```

Every **Set Event Processing Off** statement should have a corresponding **Set Event Processing On** statement to restore event processing. In environments which perform cooperative multi-tasking, leaving event processing off can prevent other software applications from multi-tasking.

You also can suppress the redrawing of a Map window by issuing a **Set Map…Redraw Off** statement, which has an effect similar to the **Set Event Processing Off** statement. However, the **Set Map statement** only affects the redrawing of one Map window, while the **Set Event Processing** statement affects the redrawing of all MapInfo Professional windows.

Set File Timeout statement

Purpose

Causes MapInfo Professional to retry file i/o operations when file-sharing conflicts occur. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Set File Timeout *n*

n is a positive integer, zero or greater, representing a duration in seconds.

Description

Ordinarily, if an operation cannot proceed due to a file-sharing conflict, MapInfo Professional displays a Retry/Cancel dialog box. If a MapBasic program issues a **Set File Timeout** statement, MapInfo Professional automatically retries the operation instead of displaying the Retry/Cancel dialog box.

If n is greater than zero, retry processing is enabled. Thereafter, whenever the user attempts to read a table that is busy (for example, a table that is being saved by another user), MapInfo Professional repeatedly tries to access the table. If, after *n* seconds, the table is still unavailable, MapInfo Professional displays a Retry/Cancel dialog box. Note that the Retry/Cancel dialog box is not trappable; the dialog box appears regardless of whether an error handler has been enabled.

If *n* is zero, retry processing is disabled. Thereafter, if MapInfo Professional attempts to access a table that is busy, the Retry/Cancel dialog box appears immediately.

Do not use the **Set File Timeout** statement and the OnError error-trapping feature at the same time. In places where an error handler is enabled, the file-timeout value should be zero.

In places where the file-timeout value is greater than zero, error trapping should be disabled. For more information on file-sharing issues, see the *MapBasic User Guide*.

Example

```
Set File Timeout 100
```

Set Format statement

Purpose

Affects how MapBasic processes Strings that represent dates or numbers. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax 1

```
Set Format Date { "US" | "Local" }
```

Syntax 2

```
Set Format Number { "9,999.9" | "Local" }
```

Description

Users can configure various date and number formatting options by using control panels that are provided with the operating system. For example, a Windows user can change system date formatting by using the control panel provided with Windows.

Some MapBasic functions, such as the **Str$( ) function**, are affected by these system settings. In other words, some functions are unpredictable, because they produce different results under different system configurations.

The **Set Format** statement lets you force MapBasic to ignore the user's formatting options, so that functions such as the **Str$( ) function** behave in a predictable manner.

Statement	Effect on your MapBasic application
`Set Format Date "US"`	MapBasic uses Month/Day/Year date formatting regardless of how the user's computer is set up.
`Set Format Date "Local"`	MapBasic uses whatever date-formatting options are configured on the user's computer.

Statement	Effect on your MapBasic application
`Set Format Number "9,999.9"`	The **Format$( ) function** uses U.S. number formatting options (decimal separator is a period; thousands separator is a comma), regardless of how the user's computer is configured.
`Set Format Number "Local"`	The **Format$( ) function** uses the number formatting options set up on the user's computer.

Syntax 1 (**Set Format Date**) affects the output produced under the following circumstances: Calling the **StringToDate( ) function**; passing a date to the **Str$( ) function**; or performing an operation that causes MapBasic to perform automatic conversion between dates and strings (for example, issuing a **Print statement** to print a date, or assigning a date value to a string variable).

Syntax 2 (**Set Format Number**) affects the output produced by the **Format$( ) function** and the **FormatNumber$( ) function**. Applications compiled with MapBasic 3.0 or earlier default to U.S. formatting. Applications compiled with MapBasic 4.0 or later default to "Local" formatting. To determine the formatting options currently in effect, call the **SystemInfo( ) function**. Each MapBasic application can issue **Set Format** statements without interfering with other applications.

Example

Suppose a date variable (*date_var*) contains the date June 11, 1995. The function call:

```
Str$( date_var )
```

may return "06/11/95" or "95/11/06" depending on the date formatting options set up on the user's computer. If you use the **Set Format Date "US"** statement before calling the **Str$( ) function**, you force the **Str$( ) function** to follow U.S. formatting (M/D/YY), which makes the results predictable.

See Also:

Format$( ) function, **FormatNumber$( ) function**, **Str$( ) function**, **StringToDate( ) function**, **SystemInfo( ) function**

Set Graph statement

Purpose

Modifies an existing Graph window. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax (5.5 and Later Graphs)

```
Set Graph
   [ Window window_id ]
   [ Title title_text ]
   [ SubTitle subtitle_text ]
   [ Footnote footnote_text ]
   [ TitleSeries titleseries_text ]
```

```
   [ TitleGroup titlegroup_text ]
   [ TitleAxisY1 titleaxisy1_text ]
   [ TitleAxisY2 titleaxisy2_text ]
```

window_id is the window identifier of a Grapher window.

title_text is the title that appears at the top of the Grapher window.

subtitle_text is the graph subtitle text.

footnote is the graph footnote text.

titleseries_text is the graph titleseries text.

titlegroup_text is the graph title group text.

titleaxisY1_text is the text for Y axis title.

titleaxisY2 is the text for Y2.

Syntax (Pre-5.5 Graphs)

```
Set Graph
   [ Window window_id ]
   [ Type { Area | Bar | Line | Pie | XY } ]
   [ Stacked { On | Off } ]
   [ Overlapped { On | Off } ]
   [ Droplines { On | Off } ]
   [ Rotated { On | Off } ]
   [ Show3d { On | Off } ]
   [ Overlap overlap_percent ]
   [ Gutter gutter_percent ]
   [ Angle angle ]
   [ Title graph_title [ Font... ] ]
   [ Series series_num
      [ Pen... ]
      [ Brush... ]
      [ Line... ]
      [ Symbol... ]
      [ Title series_title ] ]
      [ Wedge wedge_num
         [ Pen... ]
         [ Brush... ] ] ]
   [ { Label | Value } Axis
      [ { Major | Minor } Tick { Cross | Inside | None | Outside } ]
      [ { Major | Minor } Grid { On | Off } Pen... ]
      [ Labels { None | At Axis } [ Font... ] ]
      [ Min { min_value | Auto } ]
      [ Max { max_value | Auto} ]
      [ Cross { cross_value | Auto } ]
      [ { Major | Minor } Unit { unit_value | Auto } ]
      [ Pen... ]
      [ Title axis_title [ Font... ] ] ]
   [ Legend
      [ Title legend_title [ Font... ] ]
```

```
[ Subtitle legend_subtitle [ Font... ] ]
[ Range [ Font... ] ] ]
```

window_id is the window identifier of a Grapher window.

overlap_percent is the percentage value, from zero to 100, dictating bar overlap.

gutter_percent is a percentage value, from zero to 100, dictating space between bars.

angle is a number from zero to 360, representing the starting angle of a pie chart.

graph_title is the title that appears at the top of the Grapher window.

axis_title is a title that appears on one of the axes of the Grapher window.

min_value is the minimum value to show along the appropriate axis.

max_value is the maximum value to show along the appropriate axis.

cross_value is the value at which the axes should cross.

unit_value is the unit increment between labels on an axis.

series_num is an integer identifying which series of a graph to modify (for example, 2, 3, …).

series_title is the name of a series; this appears next to the pen/brush sample in the Legend.

legend_title and *legend_subtitle* are text strings which appear in the Legend.

Line clause specifies a line style.

Brush is a valid **Brush clause** to specify fill style.

Pen is a valid **Pen clause** to specify the fill's border.

Symbol is a valid **Symbol clause** to specify a point style.

Font is a valid **Font clause** specifies a text style.

Description

The **Set Graph** statement alters the settings of an existing Graph window. If no *window_id* is specified, the statement affects the topmost Graph. This statement allows a MapBasic program to control those options which an end-user would set through MapInfo Professional's Graph menu, as well as some options which a user would set through the Customize Legend dialog box.

Between sessions, MapInfo Professional preserves Graph settings by storing a **Set Graph** statement in the workspace file. Thus, to see an example of the **Set Graph** statement, you could create a Graph, save the workspace (for example, GRAPHER.WOR), and examine the workspace in a MapBasic text edit window. You could then cut/copy and paste to put the **Set Graph** statement in your MapBasic program file. To change the width, height, or position of a Graph window, use the **Set Window statement**.

Example

5.5 and later graphs:

```
include 'mapbasic.def'
graph_id = WindowId(4) ' window code for a graph is 4
Set Graph
```

```
   Window graph_id
   Title "United States"
   SubTitle "1990 Population"
   Footnote "Values from 1990 Census"
   TitleGroup "States"
TitleAxisY1 "Population"
```

pre 5.5 graphs:

The following example illustrates how the **Set Graph** statement can customize a Grapher, as well as customizing the Grapher-related items that appear in the Legend window. The **Graph statement** creates a Graph window which graphs two columns (orders_rcvd and orders_shipped) from the Selection table.

Note that the **Graph statement** actually specifies three columns; data from the first column (sales_rep) is used to label the graph.

```
Open Window Legend
   Set Window Legend
      Position (3.0, 1.6) Width 3.3 Height 0.750000
Graph sales_rep,orders_rcvd,orders_shipped
   From selection
   Position (0.2, 0.1) Width 4.5 Height 3.9
'
' The 1st Set Graph statement customizes the type of
' graph and the main title of the graph
'
Set Graph
   Type Bar Stacked Off Overlapped Off
   Droplines Off Rotated Off Show3d Off
   Overlap 30 Gutter 10 Angle 0
   Title "Orders Received vs. Orders Shipped"
   Font ("Arial",1,18,0)
'
' the next Set Graph sets all of the attributes of
' the Label axis (since we earlier chose Rotated
' off, this is the x axis).
'
Set Graph Label Axis
   Major Tick Outside
   Major Grid Off Pen (1,2,117440512)
   Minor Tick None
   Minor Grid Off Pen (1,2,117440512)
   Min 1.0 Max 5.0
      Cross 1.0 Major unit 1.0 Minor unit 0.5
   Labels At Axis Font ("Arial",0,8,0)
   Pen (1,2,117440512)
   Title "Salesperson" Font ("Arial",0,8,0)
   '
   ' the above title ("Salesperson") appears
   ' along the grapher's x-axis
   '
'
```

```
' next Set Graph sets attributes of value (y) axis
'
Set Graph Value Axis
   Major Tick Outside
   Major Grid Off Pen (1,2,117440512)
   Minor Tick None
   Minor Grid Off Pen (1,2,117440512)
   Min 0.0 Max 300000.0
      Cross 0.0 Major unit 50000.0 minor unit 25000.0
   Labels At Axis Font ("Arial",0,8,0)
   Pen (1,2,117440512)
   Title "Order amounts ($)" Font ("Arial",0,8,0)
'
' the above title ("Order amounts...") appears
' along the grapher's y-axis
'
'
' The next set graph customizes graphical styles
' for series 2. This dictates what color bars will
' appear to represent the orders_rcvd column data.
' Also controls what description will appear in the
' legend
'
' Since this is a bar graph, the Brush is the style
' of prime importance; if this was a line graph,
' the Line and Symbol clauses would be important).
'
Set Graph Series 2
   Brush (8,255,16777215)
   Line (1,2,0,255) Symbol (32,255,12)
   Title "Orders Received ($)"
    '
   'the above title will appear in the legend...
   '
'
' The next set graph customizes the styles
' used by series 3 (orders_shipped).
'
Set Graph Series 3
   Brush (2,12632256,201326591)
   Line (1,2,0,0) Symbol (34,12632256,12)
   Title "Orders Shipped ($)'
   '
   ' the above title will appear in the legend...
   '
'
' the last Set Graph statement dictates what
' Grapher-related title and subtitle will appear
' in the Legend window, as well as what fonts will
' be used in the legend.
'
Set Graph Legend
```

```
Title "Orders Received vs. Orders Shipped"
Font ("Arial",0,10,0) 'set the title font
Subtitle "(by salesperson)"
Font ("Helv",0,8,0) 'set subtitle font
'set the font used for range descriptions
Range font ("Arial",2,8,0)
```

See Also:

Graph statement, **Set Window statement**

Set Handler statement

Purpose

Enables or disables the automatic calling of system handler procedures, such as the **SelChangedHandler procedure**.

Restrictions

You cannot issue this statement through the MapBasic window.

Syntax

Set Handler *handler_name* { **On** | **Off** }

handler_name is the name of a system handler procedure, such as **SelChangedHandler procedure**.

Description

Ordinarily, if you include a system handler procedure in your program, MapInfo Professional calls the handler procedure automatically, whenever a related system event occurs. For example, if your program contains a **SelChangedHandler procedure**, MapInfo Professional calls the procedure automatically, every time the Selection changes.

Use the **Set Handler** statement to disable the automatic calling of system handler procedures within your MapBasic program.

The **Set Handler…Off** statement does not have any effect on explicit procedure calls (using the **Call statement**).

Example

The following example shows how a **Set Handler** statement can help to avoid infinite loops.

```
Sub SelChangedHandler
   Set Handler SelChangedHandler Off

   ' Issuing a Select statement here
   ' will not cause an infinite loop.
```

```
   Set Handler SelChangedHandler On
End Sub
```

See Also:

SelChangedHandler procedure, ToolHandler procedure

Set Layout statement

Purpose

Modifies an existing Layout window. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Set Layout
   [ Window window_id ]
   [ Center ( center_x, center_y ) ]
   [ Extents { To Fit | ( pages_across, pages_down ) } ]
   [ Pagebreaks { On | Off } ]
   [ Frame Contents { Active | On | Off } ]
   [ Ruler { On | Off } ]
   [ Zoom { To Fit | zoom_percent } ]
   [ {Objects Alpha alpha_value } | { Objects Translucency
      translucency_percent }]
```

window_id is the window identifier of a Layout window.

center_x is the horizontal layout position currently at the middle of the Layout window.

center_y is the vertical layout position currently at the middle of the Layout window.

pages_across is the number of pages (one or more) horizontally that the layout should span.

pages_down is the number of pages (one or more) vertically that the layout should span.

zoom_percent is a percentage indicating the Layout window's size relative to the actual page.

alpha_value is an integer value representing the alpha channel value for translucency. Values range from 0-255 where 0 is completely transparent and 255 is completely opaque. Values between 0-255 make the objects in the layout display translucently.

translucency_percent is an integer value representing the percentage of translucency for the objects in a layout. Values range between 0-100. 0 is completely opaque. 100 is completely transparent.

Specify either **Alpha** or **Translucency** but not both, since they are different ways of specifying the same result. If you specify multiple keywords, the last value will be used.

Description

The **Set Layout** statement controls the settings of an existing Layout window. If no *window_id* is specified, the statement affects the topmost Layout window. This statement allows a MapBasic program to control those options which a user would set through MapInfo Professional's Layout menu.

The **Center** clause specifies the location on the layout which is currently at the center of the Layout window.

The **Extents** clause controls how many pages (for example, how many sheets of paper) will constitute the page layout. The following clause:

```
Set Layout Extents To Fit
```

configures the layout to include however many pages are needed to ensure that all objects on the layout will print. Alternately, the **Extents** clause can specify how many pages wide or tall the page layout should be. For example, the following statement would make the page layout three pages wide by two pages tall:

```
Set Layout Extents (3, 2)
```

If the layout consists of more than one sheet of paper, the **Pagebreaks** clause controls whether the Layout window displays page breaks. When page breaks are on (the default), MapInfo Professional displays dotted lines to indicate the edges of the pages.

The **Frame Contents** clause controls when and whether MapInfo Professional refreshes the contents of the layout frames. A page layout typically contains one or more frame objects; each frame can display the contents of an existing MapInfo Professional window (for example, a frame can display a Map window). As you change the window(s) on which the layout is based, you may or may not want MapInfo Professional to take the time to redraw the Layout window. Some users want the Layout window to constantly show the current contents of the client window(s); however, since Layout window redraws take time, some users might want the Layout window to redraw only when it is the active window.

The following statement tells MapInfo Professional to always redraw the Layout window, when necessary, to reflect changes in the client window(s):

```
Set Layout Frame Contents On
```

The following statement tells MapInfo Professional to only redraw the Layout window when it is the active window:

```
Set Layout Frame Contents Active
```

The following statement tells MapInfo Professional to never redraw the Layout window:

```
Set Layout Frame Contents Off
```

When **Frame Contents** are set **Off**, each frame appears as a plain rectangle with a simple description (for example, “World Map”).

The **Ruler** clause controls whether MapInfo Professional displays a ruler along the top and left edges of the Layout window. By default, **Ruler** is **On**.

The **Zoom** clause specifies the magnification factor of the page layout; in other words, it enlarges or reduces the window's view of the layout. For example, the following statement specifies a zoom setting of fifty percent:

```
Set Layout Zoom 50.0
```

When a page layout is displayed at fifty percent, that means that an actual sheet of paper is twice as wide and twice as high as it is represented on-screen (in the Layout window). Note that the page layout can show extreme close-ups, for the sake of allowing accurate detail work. Accordingly, a Layout window displayed at 200 percent will show a magnification of the page. The **Zoom** clause can specify a zoom value anywhere from 6.25% to 800%, inclusive. The **Zoom** clause does not need to specify a specific percentage. The following statement tells MapInfo Professional to set the zoom level so that the entire page layout will appear in the Layout window at one time:

```
Set Layout Zoom To Fit
```

Once a Layout window's frame object has been selected, a MapBasic program could issue a **Run Menu Command statement** to perform a Move to back or Move to front operation. Also, since frame objects are (in some senses) conventional MapInfo Professional graphical objects, MapBasic's **Alter Object statement** lets an application reset the pen and brush styles associated with frame objects.

To change the width, height, or position of a Layout window, use the **Set Window statement**.

Example

```
Set Layout
   Zoom To Fit Extents To Fit
   Ruler Off
   Frame Contents On
```

See Also:

Alter Object statement, **Create Frame statement**, **Layout statement**, **Run Menu Command statement**, **Set Window statement**

Set Legend statement

Purpose

Modifies the Theme Legend window. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Set Legend
   [ Window window_id ]
   [ Layer { layer_id | layer_name | Prev }
      [ Display { On | Off } ]
      [ Shades { On | Off } ]
```

```
      [ Symbols { On | Off } ]
      [ Lines { On | Off } ]
      [ Count { On | Off } ]
      [ Title { Auto | layer_title [ Font... ] } ]
      [ SubTitle { Auto | layer_subtitle [ Font... ] } ]
      [ Style Size { Large | Small }]
      [ Columns number_of_columns ]
      [ Ascending { On | Off } | Order { Ascending | Descending |
        Custom }]
      [ Ranges { Auto | [ Font... ]
        [ Range { range_identifier | default } ]
        range_title [ Display { On | Off } ] }
        [ , ... ]
      ]
   ]
   [ , ... ]
```

window_id is the integer window identifier of a Map window.

layer_id is a SmallInt that identifies a layer of the map.

layer_name is a string that identifies a map layer.

layer_title, *layer_subtitle* are character strings which will appear in the theme legend.

range_title is a text string describing one range in a layer that is shaded by value.

Description

The **Set Legend** statement controls the appearance of the contents in MapInfo Professional's Theme Legend window. To change the width, height, or position of the Legend window, use the **Set Window statement**.

Between sessions, MapInfo Professional preserves theme legend settings by storing a **Set Legend** statement in the workspace file. To see an example of the **Set Legend** statement, you could create a Map, create a theme legend, save the workspace (for example, LEGEND.WOR), and examine the workspace in a MapBasic text editor window. You could then cut/copy and paste to put the **Set Legend** statement in your MapBasic program file.

Although MapInfo Professional can maintain a large number of Map windows, only one Theme Legend window exists at any given time. The Theme Legend window displays information about the active Map. Thus, the **Set Legend** statement's *window_id* clause identifies one of the Map windows in use, not the Legend window. If no *window_id* is specified, the statement affects the legend settings for the topmost Map window.

The **Layer** clause specifies which layer's theme legend should be modified. The **Layer** clause can identify a layer by its specific number (for example, specify 2 to control the theme legend of the second map layer), by its name, or by specifying **Layer Prev**. The **Layer Prev** clause tells MapBasic to modify whatever map layer was last created or modified through a **Set Shade statement** or **Shade statement**.

If a Map window contains two or more thematic layers, the **Set Legend** statement can include one **Layer** clause for each thematic layer.

The remainder of the options for the **Set Legend** statement all pertain to the **Layer** clause; that is, all of the clauses described below are actually sub-clauses within the **Layer** clause.

The **Count** clause dictates whether each line of the theme legend should include a count, in parentheses, of how many of the table's records belong to that range. The **Shades**, **Symbols** and **Lines** clauses dictate which types of graphic objects appear in each line of the theme legend. If the statement includes the **Shades On** clause, each line of the theme legend will include a sample fill pattern. If the statement includes the **Symbols On** clause, each line of the theme legend will include a sample symbol marker. If the statement includes the **Lines On** clause, each line of the theme legend will include a sample line style.

The **Title** clause specifies what title, if any, will appear above the range information in the theme legend. Similarly, the **Subtitle** clause specifies a subtitle. The title and the subtitle are each limited to thirty-two characters. If a theme legend includes a title, a subtitle, and range information, the objects will appear in that order—the title first, then the subtitle below it, then the range information below the subtitle. If the optional **Auto** clause is used, the text is automatically generated for each theme.

The **Font** clause specifies a text style.

The **Ascending On** clause arranges the range descriptions in ascending order. If this optional clause is omitted, the default order of the ranges is descending.

The **Ranges** clause describes the text that will accompany each line in the theme legend. Each range description consists of a text string (*range_title*) followed by a **Display** clause. The *Display* clause (*Display On* or *Display Off*) dictates whether that range will be displayed in the theme legend. Note If the *Auto* clause is not used, the *Ranges* clause must include a *range_title Display* clause for each range in the thematic map, even if some of the ranges are not to be displayed.

If a map layer is a graduated symbols theme, there should be exactly two *range_title* **Display** clauses. If a map layer is shaded as a dot density theme, there should be exactly one *range_title* **Display** clause. Otherwise, there should be one more *range_title* **Display** clause than there are ranges; this is because the theme legend reserves one line for an artificial range known as “all others”. The all-others range represents any and all objects which do not belong to any of the other ranges.

The **Order** and **Range** clauses will increase the workspace version to the current version. Old workspaces will still parse correctly as there is still support for the original **Ascending** clause. If the order is not custom, MapInfo Professional will write out the original **Ascending** clause and NOT increase the workspace version.

The **Order** clause is another way to specify legend label order of ascending or descending as well as new custom order. However, the original **Ascending** clause is still available for backwards compatibility. You can use either the **Order** clause, or the **Ascending** clause, but not both (both clauses cannot be included in the same MapBasic statement or you will get a syntax error).

The **Custom** option for the **Order** clause is allowed only for Individual Value themes. An error will occur if you try to custom order other theme types. The error is “`Custom legend label order is only allowed for Individual Value themes.`”

When the **Order** is **Custom**, each range in the **Ranges** clause must include a range identifier, otherwise a syntax error will occur. The range identifier must come before the range title and **Display** clause. The range identifier is the same const string or value used by the **Values** clause in the **Shade statement** that creates the Individual Value theme. The range identifier for the "all others" category is 'default'.

Every category in the theme must be included, including the default or "all others" category, otherwise an error will occur. The error is "`Incorrect number of ranges specified for custom order`."

The default or "all others" category may also be reordered, although the best place to place this argument is at the end or beginning of the **Ranges** clause.

If the range identifier does not refer to a valid category an error will occur. The error is "`Invalid range value for custom order`."

The **Style Size** clause facilitates thematic swatches to appear in different sizes.

The **Columns** clause allows you to specify the width of the legend. *number_of_columns* indicates the column width.

See Also:

Map statement, **Open Window statement**, **Set Map statement**, **Set Window statement**, **Shade statement**

Set LibraryServiceInfo statement

Purpose

Resets the current Library Service related attributes. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Set LibraryServiceInfo
   { URL url }
```

url is a valid Library Service URL.

Description

URL is a valid **URL clause** to specify the Library Service URL.

Example

```
Include "mapbasic.def"
declare sub main
Set LibraryServiceInfo URL
"http://localhost:8080/LibraryService/LibraryService"
end sub
```

See Also:

LibraryServiceInfo( ) function, **URL clause**

Set Map statement

Purpose

Modifies an existing Map window. You can issue this statement from the MapBasic Window in MapInfo Professional. The Set Map statement has an extensive set of clauses, so syntax descriptions are organized by topic.

Syntax

```
Set Map
   [ Window window_id ]
   [ MAP_BEHAVIOR_CLAUSES ]
   [ VIEW_CLAUSES ]
   [ LAYER_PROPERTY_CLAUSES ]
   [ LABEL_CLAUSES ]
   [ STYLE_OVERRIDE_CLAUSES ]
   [ LABEL_OVERRIDE_CLAUSES ]
   [ GROUPLAYER_PROPERTY_CLAUSES ]
   [ ORDER_LAYERS_CLAUSES ]
   [ COORSYS_CLAUSES ]
   [ IMAGE_CLAUSES ]
   [ LAYER_ACTIVATE_CLAUSES ]
```

window_id is the integer window identifier of a Map window.

MAP_BEHAVIOR_CLAUSES see **Changing the Behavior of the Entire Map**

VIEW_CLAUSES see **Changing the Current View of the Map**

LAYER_PROPERTY_CLAUSES see **Managing Individual Layer Properties and Appearance**

LABEL_CLAUSES see **Managing Individual Label Properties**

STYLE_OVERRIDE_CLAUSES see **Adding Style Overrides to a Layer**.

LABEL_OVERRIDE_CLAUSES see **Adding Overrides for Layer Labels**.

GROUPLAYER_PROPERTY_CLAUSES see **Managing Group Layers**

ORDER_LAYERS_CLAUSES see **Ordering Layers**

COORSYS_CLAUSES see **Managing the Coordinate System of the Map**

IMAGE_CLAUSES see **Managing Image Properties**

LAYER_ACTIVATE_CLAUSES see **Managing Hotlinks**.

Description

The **Set Map** statement controls the settings of a Map window. If no *window_id* is specified, the statement affects the topmost Map window. This statement allows a MapBasic program to control options a user would set through MapInfo Professional's **Map > Layer Control**, **Map > Change View**, and **Map > Options** menu items. For example, the **Set Map** statement lets you configure which map layer is editable, and lets you set the map's zoom distance or scale.

Set Map controls the contents of a Map window, not the size or position of the window's frame. To change the size or position of a Map window, use the **Set Window statement**.

Between sessions, MapInfo Professional preserves Map settings by storing a **Set Map** statement in a workspace file. To see an example of the **Set Map** statement, create a map, save the workspace (for example, MAPPER.WOR), and examine the workspace in a text editor, such as Notepad.

The order of the clauses in a **Set Map** statement is very important. Entering the clauses in an incorrect order can generate a syntax error.

See Also:

Add Map statement, **Map statement**, **MapperInfo( ) function**, **Remove Map statement**, **Set Window statement**, **LayerInfo( ) function**, **LayerListInfo function**, **LayerStyleInfo( ) function**, **StyleOverrideInfo( ) function**, **LabelOverrideInfo( ) function**

Changing the Behavior of the Entire Map

The following clauses affect the behavior of the map, such as units, clipping object behavior, and redraw behavior.

Syntax

```
Set Map
   [ Window window_id ]
   [ Clipping [ Object clipper] [{ Off | On}]
      [Using { Display { PolyObj  |  All } | Overlay } ] ]
   [ Preserve { Scale | Zoom } ]
   [ Area Units area_unit ]
   [ Distance Units dist_unit ]
   [ Display { Scale | Position | Zoom } ]
   [ Redraw { On | Off | Suspended } ]
   [ Move Nodes { value | Default } ]
```

window_id is the integer window identifier of a Map window.

clipper is an Object expression; only the portion of the map within the object will display. See the description in the Clipping section for more information.

area_unit is a string representing the name of an area unit used to display area calculations (for example, “sq mi” for square miles, “sq km” for square kilometers; see **Set Area Units statement** for a list of unit names). For example:

```
Set Map Area Units "sq km"
```

dist_unit is a string expression, specifying the units for the map (such as “mi” for miles, “m” for meters; see **Set Distance Units statement** for a list of available unit names). This is an optional parameter. If not present, the distance units from the table's coordinate system are used.

value can be 0 or 1. If the value is 0, duplicate nodes are not moved. If the value is 1, any duplicate nodes within the same layer will be moved.

Description

Clipping sets a clipping object for the Map window; corresponds to MapInfo Professional's **Map > Set Clip Region** command. Once a clipping region is set, enable or disable clipping by specifying **Clipping On** or **Clipping Off**.

```
Set Map Clipping Object obj_variable_name
```

There are three modes that can be used for Clipping. Using the **Overlay** mode will use the MapInfo Professional **Objects > Erase Outside** functionality to produce the clipping. Polylines and Regions will be clipped at the Region boundary. Points and Labels will be completely displayed only if the point or label point lie inside the Region. Text is always displayed and never clipped. Styles for all objects are never clipped. Using the **Display All** mode, the Windows display will provide the clip region functionality. All objects (including points, labels, and text) will be clipped at the Region boundary. All styles will be clipped at the region boundary. This is the default mode.

Using the **Display PolyObj** mode the Windows display will provide the clip region functionality for Polylines and Regions only. Styles for Polylines and Regions will be clipped at the region boundary. Points and Labels will be completely displayed only if the point or label point lie inside the Region. Text is always displayed and never clipped. Styles for points, labels and text are never clipped.

In general, the Windows display functionality found in **Display All** and **Display PolyObj** provides better performance than the Overlay functionality. For example:

```
Set Map Clipping Object obj_variable_name Using Display All
```

Display dictates what type of information should appear on the status bar when the Map window is active. **Display Zoom** displays the current zoom (the width of the area displayed). **Display Scale** displays the current scale. **Display Position** displays the position of the cursor (for example, decimal degrees of longitude/latitude).

```
Set Map Display Position
```

Preserve controls how the Map window behaves when the user re-sizes the window. If you specify **Preserve Zoom** then MapInfo Professional redraws the entire Map window whenever the user re-sizes the window. If you specify **Preserve Scale** then MapInfo Professional only redraws the portion of the window that needs to be redrawn. These options correspond to settings in MapInfo Professional's Options dialog box (**Map > Options**).

Redraw disables or enables the automatic redrawing of the Map window. If you issue a **Set Map Redraw Off** statement, subsequent statements can affect the map (for example, **Set Map**, **Add Map Layer**, **Remove Map Layer**) without causing MapInfo Professional to redraw the Map window. After

making all necessary changes to the Map window, issue a **Set Map Redraw On** statement to restore automatic redrawing (at which time, MapInfo Professional will redraw the map once to show all changes).

Some actions, such as panning and zooming, can cause MapInfo Professional to redraw a Map window even after you specify **Redraw Off**. If you find that the **Redraw Off** syntax does not prevent window redraws, you may want to use the **Set Event Processing Off statement**.

Redraw has three options, **On**, **Off** and **Suspended**. The **Suspended** keyword will draw a visual cue suggesting the state of map redraws, on the map window. For example:

```
Set Map Redraw Suspended
```

The user can put the maps into a suspended state by clicking a button at the bottom of the Layer Control window.

Move Nodes can be 0 or 1. If the value is 0, duplicate nodes are not moved. If the value is 1, any duplicate nodes within the same layer will be moved. If a **Move Node** *value* is specified, that window is considered to be using a custom value. To return to using the default (from the mapper preference), specify **Move Nodes Default**.

Once **Set Map Move Nodes** value has been used, that map has a custom setting. If a Map window has a custom setting, the Map window preference will not be used. The Map window preference will apply to new Map windows and any non-customized Map windows. The setting for an existing Map window can be customized by using the **Set Map Move Nodes value** MapBasic statement.

Example

The following program opens two tables, opens a Map window to show both tables, and then performs a **Set Map** statement to make changes to the Map window:

```
Open Table "world"
Open Table "cust1993" As customers
Map From customers, world

Set Map
   Center (-100, 40) 'center map over mid-USA
   Zoom 4000 Units "mi" 'show entire USA
   Preserve Zoom 'preserve zoom when resizing
   Display Position 'show lat/long on status bar
```

Changing the Current View of the Map

The following clauses affect the current view—in other words, where the map is centered, and how large an area is displayed in the Map window.

Syntax

```
Set Map
   [ Window window_id ]
```

```
[ Center ( longitude, latitude ) [ Smart Redraw ] ]
[ Zoom {
   zoom_distance [ Units dist_unit ] | Entire [ Layer layer_id ] } ]
[ Pan pan_distance [ Units dist_unit ]
   { North | South | East | West } [ Smart Redraw ] ]
[ Scale screen_dist [ Units dist_unit ] For map_dist
   [ Units dist_unit ] ]
```

window_id is the integer window identifier of a Map window.

longitude, *latitude* is the new center point of the map.

zoom_distance is a numeric expression dictating how wide an area to display.

dist_unit is a string expression, specifying the units for the map (such as “mi” for miles, “m” for meters; see **Set Distance Units statement** for a list of available unit names). This is an optional parameter. If not present, the distance units from the table's coordinate system are used.

layer_id identifies which layer to modify; can be a SmallInt (for example, use 1 to specify the top map layer other than Cosmetic) or a string representing the name of a table displayed in the map.

pan_distance is a distance to pan the map.

screen_dist and *map_dist* specify a map scale (for example, screen_dist = 1 inch, map_dist = 1 mile).

Description

Center controls where the map will be centered within the Map window. For example: New York City is located (approximately) at 74 degrees West, 41 degrees North. The following **Set Map** statement centers the map in the vicinity of New York City. Coordinates are specified in decimal degrees, not Degrees/Minutes/Seconds.

```
Set Map Center (-74.0, 41.0)
```

A **Set Map…Center** statement causes the entire window to redraw, unless you include the optional **Smart Redraw** clause. For details on **Smart Redraw**, see below (under Pan).

Pan moves the Map window's view of the map. For example, the following statement moves the map view 100 kilometers north:

```
Set Map Pan 100 Units "km" North
```

Ordinarily, the **Set Map…Pan** statement redraws the entire Map window. If you include the optional **Smart Redraw** clause, MapInfo Professional only redraws the portion of the map that needs to be redrawn (as if the user had re-centered the map using the window scrollbars or the Grabber tool).

```
Set Map Pan 100 Units "km" North Smart Redraw
```

CAUTION: **if you include the Smart Redraw clause, the Map window always moves in multiples of eight pixels. Because of this behavior, the map might not move as far as you requested. For example, if you try to pan North by 100 km, the map might actually pan some other distance—perhaps 79.5 kilometers—because that other distance represents a multiple of eight-pixel increments.**

Scale zooms in or out so that the map has the scale you specify. For example, the following statement zooms the map so that one inch on the screen shows an area ten miles across.

```
Set Map Scale 1 Units "in" For 10 Units "mi"
```

Zoom dictates how wide an area should be displayed in the Map. For example, the following statement adjusts the zoom level, to display an area 100 kilometers wide.

```
Set Map Zoom 100 Units "km"
```

If the **Zoom** clause includes the keyword **Entire**, MapInfo Professional zooms the map to show all objects in a Map layer (or all objects in all map layers):

```
Set Map Zoom Entire Layer 2 'show all of layer 2
Set Map Zoom Entire 'show the whole map
```

Managing Individual Layer Properties and Appearance

The following clauses affect layers. Layer properties are optional in the **Set Map statement**.

Syntax

```
Set Map
   [ Window window_id ]
   [ Layer layer_id
      [ LAYER_ACTIVATE_CLAUSES ]
      [ Editable { On | Off } ]
      [ Selectable { On | Off } ]
      [ Zoom ( min_zoom, max_zoom ) [ Units dist_unit ] [{ On | Off }] ]
      [ Arrows { On | Off } ]
      [ Centroids { On | Off } ]
      [ Default Zoom ]
      [ Nodes { On | Off }]
      [ Inflect num_inflections [ by percent ] at
         color:value [, color:value ]
         [ Round rounding_factor ] ]
         [ Contrast contrast_value ]
         [ Brightness brightness_value ]
         [ {Alpha alpha_value }|{ Translucency translucency_percent }]
      ]
      [ Transparency { Off | On } ]
      [ Color transparent_color_value ]
      [ GrayScale { On | Off } ]
      [ Relief { On | Off } ]
      [ LABEL_CLAUSES ]
      [ LAYER_OVERRIDE_CLAUSES ]
      [ Display { Off | Graphic | Global } ]
      [ Global Line... ] [, Line...] ...
      [ Global Pen... ] [, Pen...] ...
      [ Global Brush... ] [, Brush...] ...
      [ Global Symbol... ] [, Symbol...] ...
      [ Global Font... ] [, Font...] ...
   ]
```

window_id is the integer window identifier of a Map window.

layer_id identifies which layer to modify; can be a SmallInt (for example, use 1 to specify the top map layer other than Cosmetic) or a string representing the name of a table displayed in the map.

LAYER_ACTIVATE_CLAUSES is a shorthand notation, not a MapBasic Keyword. See Layer Activate Clause described under **Managing Hotlinks on page 671**.

min_zoom is a numeric expression, identifying the minimum zoom at which the layer will display.

max_zoom is a numeric expression, identifying the maximum zoom at which the layer will display.

dist_unit is a string expression, specifying the units for the map (such as “mi” for miles, “m” for meters; see **Set Distance Units statement** for a list of available unit names). This is an optional parameter. If not present, the distance units from the table's coordinate system are used.

num_inflections is a numeric expression, specifying the number of *color:value* inflection pairs used in a Grid theme.

color is an expression of color using the **RGB( ) function**.

value is an inflection that is displayed in the paired color.

rounding_factor is a numeric expression, specifying the rounding factor applied to the inflection values.

contrast_value a value of 0 to 100 representing contrast. This value corresponds to the slider on the Grid Appearance dialog box, which is available when modifying a grid theme from the Modify Thematic Map dialog box.

brightness_value a value of 0 to 100 representing brightness. This value corresponds to the slider on the Grid Appearance dialog box, which is available when modifying a grid theme from the Modify Thematic Map dialog box.

we never describe the *contrast_value* or *brightness_value* arguments. They are both numbers from 0 to 100, which specify contrast and brightness; these correspond to the sliders in the Grid Appearance dialog box, which is accessible when you use the Modify Thematic Map dialog box to modify a Grid theme.

alpha_value is an integer value representing the alpha channel value for translucency. Values range from 0-255. 0 is completely transparent. 255 is completely opaque. Values between 0-255 make the image layer display translucent.

translucency_percent is an integer value representing the percentage of translucency for a vector, raster, or grid image layer. Values range between 0-100. 0 is completely opaque. 100 is completely transparent.

Specify either **Alpha** or **Translucency** but not both, since they are different ways of specifying the same result. If you specify multiple keywords, the last value will be used.

transparent_color_value a specific color value. **Transparency** allows raster image layers to display in a transparent mode where pixels of a certain color (*transparent_color_value*) do not draw.

LABEL_CLAUSES is a shorthand notation, not a MapBasic keyword, see **Managing Individual Label Properties**.

LAYER_OVERRIDE_CLAUSES is a shorthand notation, not a MapBasic keyword, see **Adding Style Overrides to a Layer**.

Description

Editable sets the Editable attribute for the appropriate Layer. At any given time, only one of the mapper's layers may have the **Editable** attribute turned on. Note that turning on a layer's **Editable** attribute automatically turns on that layer's **Selectable** attribute. The following **Set Map** statement turns on the **Editable** attribute for first non-cosmetic layer:

```
Set Map
   Layer 1 Editable On
```

Selectable sets whether the given layer should be selectable through operations such as Radius-Search. Any or all of the Map layers can have the **Selectable** attribute on. The following **Set Map** statement turns on the **Selectable** attribute for the first non-cosmetic map layer, and turns off the **Selectable** attribute for the second and third map layers:

```
Set Map
   Layer 1 Selectable On
   Layer 2 Selectable Off
   Layer 3 Selectable Off
```

Zoom configures the zoom-layering of the specified layer. Each layer can have a zoom-layering range; this range, when enabled, tells MapInfo Professional to only display the Map layer when the map's zoom distance is within the layering range. The following statement sets a range of 0 to 10 miles for the first non-Cosmetic layer.

```
Set Map
   Layer 1 Zoom (0, 10) Units "km" On
```

The **On** keyword activates zoom layering for the layer. To turn off zoom layer, specify **Off** instead.

Arrows turns the display of direction arrows on or off.

Centroids turns the display of centroids on or off.

Inflect overrides the inflection color:value pairs that are stored in the grid (.MIG) file.

Nodes turns the display of nodes on or off.

Relief turns relief shading for a grid on or off. The grid must have relief shade information calculated for it for this clause to have any effect. Relief shade information can be calculated for a grid with the **Relief Shade statement**.

Display controls how the objects in the layer are displayed. When you specify **Display Off**, the layer does not appear in the Map. When you specify **Display Graphic**, the layer's objects appear in their default style, as saved in the table. When you specify **Display Global**, all objects appear in the global styles assigned to the layer. These global styles can be assigned through the optional **Global** sub-clauses. The following statement displays layer 1 with green line and fill styles:

```
Set Map
   Layer 1 Display Global
   Global Line(1, 2, GREEN)
```

```
Global Pen (1, 2, GREEN)
Global Brush (2, GREEN, WHITE)
```

Global Line specifies the style used to display line and polyline objects. A **Line** clause is identical to a **Pen clause**, except for the use of the keyword **Line** instead of **Pen**.

Global Pen is a valid **Pen clause** that specifies the style used to display the borders of filled objects.

Global Symbol is a valid **Symbol clause** that specifies the style used to display point objects.

Global Brush is a valid **Brush clause** that specifies the style used to display filled objects.

Global Font is a valid **Font clause** that specifies the font used to display text objects.

The Global clauses support stacked styles as a comma separated list of like style clauses. For example, the following displays points with a global stacked symbol style:

```
Set Map Layer 1 Display Global Global Symbol (32,16777136,24),
Symbol (36,255,14)
```

The following statement adds a global stacked line style to a layer:

```
Set Map Layer 1 Display Global
Zoom (0, 10000) Units "mi"
Global Line (4, 193, 16711680), Line (2, 193, 16711680)
```

Settings That Have a Permanent Effect on a Map Layer

The **Default Zoom** clause is a special clause that modifies a table, rather than a Map window. Use the **Default Zoom** clause to reset a table's default zoom distance and center position settings to the window's current zoom and center point.

Every mappable table has a default zoom distance and center position. When the user first opens a Map window, MapInfo Professional sets the window's initial zoom distance and center position according to the zoom and center settings stored in the table.

If a **Set Map…Layer** statement includes the **Default Zoom** clause, MapInfo Professional stores the Map window's current zoom distance and center point in the named table. For example, the following statement stores the Map window's zoom and center settings in the table that comprises the first map layer:

```
Set Map Layer 1 Default Zoom
The Default Zoom clause takes effect immediately; no Save operation is
required.
```

Examples

The following statement turns on the display of arrows, centroids, and nodes for layer 1:

```
Set Map
Layer 1 Arrows On Centroids On Nodes On
```

The following statement displays layer 1 in its default style:

```
Set Map
   Layer 1 Display Graphic
```

Managing Individual Label Properties

The following clauses affect label properties for a layer. This set of clauses apply to a layer. For layer clauses, see **Managing Individual Layer Properties and Appearance**.

Syntax

```
Set Map
[ Window window_id ]
[ Layer layer_id
   [ Label
      [ Line { Simple | Arrow | None } ]
      [ Position [ Center ] [{ Above | Below }] [ {Left | Right }]]
      [ Auto Retry { On | Off } ]
      [ Font... ] [ Pen... ]
      [ With label_expr ]
      [ Parallel { On | Off } ] [ Follow Path ] [ Percent Over percent ]
      [ Visibility { On | Off | Zoom( min_vis, max_vis )
         [ Units dist_unit ] } ]
      [ Auto  { On | Off }  ]
      [ Overlap  { On | Off }  ]
      [ PartialSegments { On | Off } ]
      [ Duplicates  { On | Off }  ]
      [ Max [ number_of_labels ] ]
      [ Offset offset_amount ]
      [ Default ]
      [ LabelAlpha alpha_value ]
      [ LABEL_OVERRIDE_CLAUSES ]
   ]
   [ Object ID
    [ Table alias ]
    [ Visibility { On | Off } ]
    [ Anchor ( anchor_x, anchor_y ) ]
    [Text text_string ]
    [ Position [ Center ] [ {Above | Below }] [{ Left | Right } ] ]
    [ Font... ] [ Pen... ]
    [ Line { Simple | Arrow | None } ]
    [ Angle text_angle ] [ Follow Path ]
    [ Offset offset_amount ]
    [ Callout ( callout_x, callout_y ) ]
    [, Object... ]
   ]
]
```

window_id is the integer window identifier of a Map window.

layer_id identifies which layer to modify; can be a SmallInt (for example, use 1 to specify the top map layer other than Cosmetic) or a string representing the name of a table displayed in the map.

label_expr is the expression to use for creating labels.

Parallel is a setting with the following attributes:

- **Parallel Off** = horizontal labels, not rotated with line
- **Parallel On** = labels rotated with line
- **Follow Path** clause = create curved label, path auto calculated once and stored until location edited
- **Percent Over** When curved labels are longer than the geometry they name, this is the amount (expressed as a percentage) of overhang permitted. For example, a sample entry might be:

```
Set Map Layer 1 Label Follow Path
Percent Over 40
```

This attribute only applies to curved labels.

min_vis, *max_vis* are numbers specifying the minimum and maximum zoom distances within which the labels will display.

dist_unit is a string expression, specifying the units for the map (such as “mi” for miles, “m” for meters; see **Set Distance Units statement** for a list of available unit names). This is an optional parameter. If not present, the distance units from the table's coordinate system are used.

number_of_labels is an integer representing the maximum number of labels MapInfo Professional will display for the layer. If you omit the *number_of_labels* argument, there is no limit.

offset_amount is a number from zero to 200 (representing a distance in points), causing the label to be offset from its anchor point.

alpha_value is a SmallInt that represents the alpha value of the labels in this layer. It is a value between 0-255 where 0 is completely transparent and 255 is completely opaque. Values in between display labels translucently.

LABEL_OVERRIDE_CLAUSES is a shorthand notation, not a MapBasic keyword, see **Adding Overrides for Layer Labels**.

ID is an integer that identifies an edited label; generated automatically when the user saves a workspace. A label's *ID* equals the row ID of the object that owns the label.

alias is the name of a table that is part of a seamless map. The **Table** *alias* clause generates an error if this layer is not a seamless map.

anchor_x, *anchor_y* are map coordinates, specifying the anchor position for the label.

text_string is a string that will become the text of the label.

text_angle is an angle, in degrees, indicating the rotation of the text.

callout_x, *callout_y* are map coordinates, specifying the end of the label call-out line.

Description

The **Label** clause controls a map layer's labeling options. The **Label** clause has the following sub-clauses:

Line sets the type of call-out line, if any, that should appear when a label is dragged from its original location. You can specify **Line Simple**, **Line Arrow**, or **Line None**. For example:

```
Set Map Layer 1
   Label Line Arrow
```

Position controls label positions with respect to the positions of object centroids. For example, the following statement sets labels above and to the right of object centroids.

```
Set Map Layer 1
   Label Position Above Right
```

Auto Retry lets users to apply a placement algorithm that will try multiple label positions until a position is found that does not overlap any other label, or until all positions are exhausted.

- *When Writing Workspaces*, if the Auto Retry feature is On, we write Auto Retry On to the workspace after the Position clause (but the order isn't important), and increase the workspace version to 9.5 or later. If the feature is Off, we do not write anything to the workspace and do not increase the version number. A version 9.5 or later workspace can have Auto Retry Off in it, but we do not explicitly write it out, to avoid increasing the version unnecessarily.
- *When Reading Workspaces* If Auto Retry On or Auto Retry Off is in the workspace, it must be a version 9.5 or later workspace, otherwise a syntax error occurs. If Auto Retry is On, different positions are tried to place the label. If Auto Retry is Off, no retry is attempted - this is the default behavior. Overlap must be Off to enable the Auto Retry feature. If Overlap is On and Auto Retry On/Off are in the same LABELCLAUSE, the Auto Retry mechanism is initialized but ignored, so overlapping labels are allowed.

Font is a valid **Font clause** to specify a text style used in labels.

Follow Path is used when referring to curved labels

Pen is a valid **Pen clause** to specify the line style to use for call-out lines. Call-out lines only appear if you specify **Line Simple** or **Line Arrow**, and if the user drags a label from its original location.

```
Set Map Layer 1
   Label Line Arrow
      Pen( 2, 1, 255)
```

With specifies the expression used to construct the text for the labels. For example, the following statement specifies a labeling expression which uses the **Proper$( ) function** to control capitalization in the label.

```
Set Map Layer 1
   Label With Proper$(Cityname)
```

Parallel controls whether labels for line objects are rotated, so that the labels are parallel to the lines.

```
Set Map Layer 1
   Label Parallel On
```

Visibility controls whether labels are visible for this layer. Specify **Visibility Off** to turn off label display for both default labels and user-edited labels. Specify **Visibility Zoom…** to set the labels to display only when the map is within a certain zoom distance. The following example sets labels to display when the map is zoomed to 2 km or less.

```
Set Map Layer 1
   Label Visibility Zoom (0, 2) Units "km"
```

Auto controls whether automatic labels display. If you specify **Auto Off**, automatic labels will not display, although user-edited labels will still display.

Overlap controls whether MapInfo Professional draws labels that would overlap existing labels. To prevent overlapping labels, specify **Overlap Off**.

PartialSegments controls whether MapInfo Professional labels an object when the object's centroid is not in the visible portion of the map. If you specify **PartialSegments On** (which corresponds to selecting the **Label Partial Objects** check box in MapInfo Professional), MapInfo Professional labels the visible portion of the object. If you specify **PartialSegments Off**, an object will only be labeled if its centroid appears in the Map window.

Duplicates controls whether MapInfo Professional allows two or more labels that have the same text. To prevent duplicate labels, specify **Duplicates Off**.

Max *number_of_labels* sets the maximum number of labels that MapInfo Professional will display for this layer. If you omit the *number_of_labels* argument, MapInfo Professional places no limit on the number of labels.

Offset *offset_amount* specifies an offset distance, so that MapInfo Professional automatically places each label away from the object's centroid. The *offset_amount* argument is an integer from zero to 50, representing a distance in points. If you specify **Offset 0** labels appear immediately adjacent to centroids. If you specify **Offset 10** labels appear 10 points away. The offset setting is ignored when the **Position** clause specifies centered text.

The following statement allows overlapping labels, placed to the right of object centroids, with a horizontal offset of 10 points:

```
Set Map Layer 1
   Label Overlap On Position Right Offset 10
```

Default resets all of the labels for this layer to their default values. The following statement deletes all edited labels from the top layer in the Map window, restoring the layer's default labels:

```
Set Map Layer 1 Label Default
```

The **Object** clause allows you to edit labels. For example, if you edit labels in MapInfo Professional and then save a workspace, the workspace contains **Object** clauses to represent the edited labels. The **Set Map** statement contains one **Object** clause for each edited label.

To see examples of the **Object** clause, edit a map's labels, save a workspace, and examine the workspace in a text editor.

Adding Style Overrides to a Layer

Purpose

The Override Add clause creates a new style override definition for a layer if none exists, or appends to the existing list of style override definitions. A style override allows you to change map styles based on the current zoom level of the map.

Syntax

```
Set Map
   [ Window window_id ]
   [ Layer layer_id
      [ Zoom ( min_zoom, max_zoom ) [ Units unit_dist ] ]
      [ Display { Off | Graphic | Global } ]
      [ Global Pen...[, Pen...]...]
      [ Global Line...[, Line...]...]
      [ Global Symbol...[, Symbol...]...]
      [ Global Brush...[, Brush...]...]
      [ Global Font...]
      [ { Alpha alpha_value }| { Translucency translucency_percent } ]
      [ STYLEOVERRIDE_CLAUSE ] ...
   ]
```

Where *STYLEOVERRIDE_CLAUSE* is:

```
[ [ Style ] Override Add [override_name] {
   [ Using [ Window window_id ] Layer layer_id {
      All | Override { override_index | override_name } }
   ]
   |
   Zoom ( min_zoom, max_zoom )
   [ Units dist_unit ]
   [ { Alpha alpha_value } | { Translucency translucency_percent } ]
   [ Enable { On | Off } ]
   [ Arrows { On | Off } ]
   [ Centroids { On | Off } ]
   [ Nodes { On | Off } ]
   [ Line...] [, Line... ]
   [ Pen...] [, Pen... ] ...
   [ Symbol... ] [, Symbol... ] ...
   [ Brush... ] [, Brush... ] ...
   [ Font... ] [, Font... ] ...
} ]
```

window_id is the integer window identifier of a Map window.

layer_id identifies which layer to modify; can be a SmallInt (for example, use 1 to specify the top map layer other than Cosmetic) or a string representing the name of a table displayed in the map.

override_index is an integer index (1-based) for the override definition within the layer. Each override is tied to an zoom range and is ordered so that the smallest zoom range value is on top (index 1).

override_name is the user specified override name.

min_zoom is a numeric expression, identifying the minimum zoom at which the style override will come into effect

max_zoom is a numeric expression, identifying the maximum zoom at which the style override will come into effect

dist_unit is a string expression, specifying the units for the map (such as “mi” for miles, “m” for meters; see **Set Distance Units statement** for a list of available unit names). This is an optional parameter. If not present, the distance units from the table's coordinate system are used.

alpha_value is an integer value representing the alpha channel value for translucency. Values range from 0-255. 0 is completely transparent. 255 is completely opaque. Values between 0-255 make the image layer display translucent.

translucency_percent is an integer value representing the percentage of translucency for a vector, raster, or grid image layer. Values range between 0-100. 0 is completely opaque. 100 is completely transparent.

Specify either **Alpha** or **Translucency** but not both, since they are different ways of specifying the same result. If you specify multiple keywords, the last value will be used.

Description

The display style zoom range lets you set up display style overrides that only apply within a limited range of zoom levels. There can be multiple display style zoom ranges per layer. To have the line styles change when zooming in on the map, set up multiple display style zoom ranges and assign a different style override to each of them.

The layer zoom range turns off the layer altogether if you zoom in or out too far. There can only be one of these per layer.

Arrows turns the display of direction arrows on or off.

Centroids turns the display of centroids on or off.

Nodes turns the display of nodes on or off.

Line specifies the style used to display line and polyline objects. A **Line** clause is identical to a **Pen clause**, except for the use of the keyword **Line** instead of **Pen**.

Pen is a valid **Pen clause** that specifies the style used to display the borders of filled objects.

Symbol is a valid **Symbol clause** that specifies the style used to display point objects.

Brush is a valid **Brush clause** that specifies the style used to display filled objects.

Font is a valid **Font clause** that specifies the font used to display text objects.

Using is for a one-time copy (only the overridden properties get copied) to set the initial property value of an layer override. The source and target layer do not maintain a connection.

Each vector layer supports more than one style override and more than one label override. Every style override has its own zoom range that is not allowed to overlap with any other style override for the same layer. Every label override also has its own zoom range that is not allowed to overlap any other label override for the same layer. However, style and label overrides can share or have overlapping zoom ranges between each other.

When an override comes into view (when the map's zoom range is within an override zoom range) then the map styles or labels are displayed using the override properties rather than the layers base set of style and label properties.

Style overrides do not display beyond the limits of the layer display zoom range regardless of what bounds the style override zoom range defines. Likewise, label overrides do not display beyond the limits of the layer's label zoom range, or the layer's display zoom range, regardless of what bounds the label override defines.

For more information about style overrides for layers, see **Modifying Style Overrides for a Layer** and **Enabling, Disabling, or Removing Overrides for a Layer**. See also, **LayerStyleInfo( ) function** and **StyleOverrideInfo( ) function**.

Examples

The following statement adds an override to a layer:

```
Set Map Layer 1
   Style Override Add Zoom (0, 10000) Units "mi" Line (2, 193, 16711680)
```

The following statement adds multiple style overrides to a layer:

```
Set Map Layer 1 Display Global
   Zoom (1, 10000) Units "mi"
   Global Line (1, 193, 16711680)
   Style Override Add Zoom (1, 1000) Units "mi" Line (4, 193, 16711680),
      Line (2, 193, 16711680)
   Style Override Add Zoom (1000, 10000) Units "mi"
      Line (2, 193, 16711680)
```

Example: copy a style from one map to another map

To copy a style from one map to another map:

```
Set Map Layer 1 Style Override Add Using Window 81132792 Layer 1 All
```

Examples: adding styles from another layer

The following statement adds an override for layer 2 using style named *layer1_style2* from layer 1:

```
Set Map Layer 2
   Style Override Add Using Layer 1 layer1_style1
```

The following statement adds an override for layer 2 using style 3 from layer 1:

```
Set Map Layer 2
   Style Override Add layer2_style2 Using Layer 1 Override 3
```

The following statement copies over all the overrides information from layer 2 to layer 3:

```
Set Map Layer 3 Display Global
   Global Line (1, 193, 16711680)
   Style Override Add Using Layer 2 All
```

Modifying Style Overrides for a Layer

Excluding the Add keyword from the Override clause modifies the properties for an existing multiple style override definition within a layer specified by the integer index (1-based) or the override name.

Syntax

```
Set Map
   [ Window window_id ]
   [ Layer layer_id
      [ MODIFYSTYLEOVERRIDE_CLAUSE ]
      [ MODIFYSTYLEOVERRIDE_CLAUSE ] ...
   ]
```

Where *MODIFYSTYLEOVERRIDE_CLAUSE* is:

```
[ [ Style ] Override { override_index | override_name } {
   [ Zoom ( min_zoom, max_zoom )
   [ Units dist_unit ]
   [ { Alpha alpha_value } | { Translucency translucency_percent } ]
   [ Enable { On | Off } ]
   [ Arrows { On | Off } ]
   [ Centroids { On | Off } ]
   [ Nodes { On | Off } ]
   [ Line...] [, Line... ]
   [ Pen...] [, Pen... ] ...
   [ Symbol... ] [, Symbol... ] ...
   [ Brush... ] [, Brush... ] ...
   [ Font... ]
} ]
```

window_id is the integer window identifier of a Map window.

layer_id identifies which layer to modify; can be a SmallInt (for example, use 1 to specify the top map layer other than Cosmetic) or a string representing the name of a table displayed in the map.

override_index is an integer index (1-based) for the override definition within the layer. Each override is tied to an zoom range and is ordered so that the smallest zoom range value is on top (index 1).

override_name is the user specified override name.

min_zoom is a numeric expression, identifying the minimum zoom at which the style override will come into effect

max_zoom is a numeric expression, identifying the maximum zoom at which the style override will come into effect

dist_unit is a string expression, specifying the units for the map (such as “mi” for miles, “m” for meters; see **Set Distance Units statement** for a list of available unit names). This is an optional parameter. If not present, the distance units from the table's coordinate system are used.

alpha_value is an integer value representing the alpha channel value for translucency. Values range from 0-255. 0 is completely transparent. 255 is completely opaque. Values between 0-255 make the image layer display translucent.

translucency_percent is an integer value representing the percentage of translucency for a vector, raster, or grid image layer. Values range between 0-100. 0 is completely opaque. 100 is completely transparent.

Specify either **Alpha** or **Translucency** but not both, since they are different ways of specifying the same result. If you specify multiple keywords, the last value will be used.

Description

Arrows turns the display of direction arrows on or off.

Centroids turns the display of centroids on or off.

Nodes turns the display of nodes on or off.

Line specifies the style used to display line and polyline objects. A **Line** clause is identical to a **Pen clause**, except for the use of the keyword **Line** instead of **Pen**.

Pen is a valid **Pen clause** that specifies the style used to display the borders of filled objects.

Symbol is a valid **Symbol clause** that specifies the style used to display point objects.

Brush is a valid **Brush clause** that specifies the style used to display filled objects.

Font is a valid **Font clause** that specifies the font used to display text objects.

For more information about style overrides for layers, see **Adding Style Overrides to a Layer** and **Enabling, Disabling, or Removing Overrides for a Layer**. See also,**LayerStyleInfo( ) function** and **StyleOverrideInfo( ) function**.

Example

```
Set Map Layer 1 Style Override 1 Alpha 119
```

Enabling, Disabling, or Removing Overrides for a Layer

If multistyle overrides are defined for a layer, they are enabled by default.

To enable or disable multistyle overrides for a layer, use the Override clauses with either the On or Off option.

To remove an existing override definition for a layer, use Override Remove clause.

Syntax: enable or disable overrides

```
Set Map
   [ Window window_id ]
   [ Layer layer_id [ [ Style ] Override { On | Off } ] ]
```

Syntax: remove overrides

```
Set Map
   [ Window window_id ]
   [ Layer layer_id
      [ [ Style ] Override Remove {
         All | override_index [, override_index,]
      } ]
   ]
```

window_id is the integer window identifier of a Map window.

layer_id identifies which layer to modify; can be a SmallInt (for example, use 1 to specify the top map layer other than Cosmetic) or a string representing the name of a table displayed in the map.

override_index is an integer index (1-based) for the override definition within the layer. Each override is tied to an zoom range and is ordered so that the smallest zoom range value is on top (index 1).

Description

If multistyle overrides are defined for a layer, they are enabled by default. To disable but not delete them use the Overrides clause with either the On or Off option.

For more information about style overrides for layers, see **Adding Style Overrides to a Layer** and **Modifying Style Overrides for a Layer**.

Example

```
Set Map Layer 1 Style Override Remove 3, 2
Set Map Layer 1 Style Override Remove All
```

Adding Overrides for Layer Labels

The Label Override clause adds a zoom range to an existing label override definition or creates a new override definition for labels.

Syntax

```
Set Map
   [ Window window_id ]
   [ Layer layer_id
      [ Label
         [ Visibility { On| Off | Zoom ( min_vis, max_vis )
            [ Units dist_unit ] ] }
         ]
         [ Line { Simple | Arrow | None } ]
         [ Position [ Center ] [ { Above | Below } ] [ { Left | Right } ] ]
```

```
        [ Auto Retry { On | Off } ]
        [ Font...] [ Pen... ]
        [ With label_expr ]
        [ Parallel { On | Off } ] [ Follow Path ] [Percent Over percent ]
        [ Auto { On | Off } ]
        [ Overlap { On | Off } ]
        [ PartialSegments { On | Off } ]
        [ Duplicates { On | Off } ]
        [ Max [ number_of_labels ] ]
        [ Offset offset_amount ]
        [ Default ]
        [ LabelAlpha alpha_value ]
        [ LABEL_OVERRIDE_CLAUSE ]
        [ LABEL_OVERRIDE_CLAUSE ] ...
     ]
  ]
```

Where *LABELOVERRIDE_CLAUSE* is:

```
[ [ Label ] Override Add [ labeloverride_name ] {
   [ Using [ Window window_id ] Layer layer_id {
      All | Override {labeloverride_index | labeloverride_name }
   } ] |
   Zoom ( min_vis, max_vis )
   [ Enable { On | Off } ]
   [ Units dist_unit ]
   [ Line { Simple | Arrow | None } ]
   [ Position [ Center ] [ { Above | Below } ] [ { Left | Right } ] ]
   [ Auto Retry { On | Off } ]
   [ Font...] [ Pen... ]
   [ With label_expr ]
   [ Parallel { On | Off } ] [ Follow Path ] [Percent Over percent ]
   [ Auto { On | Off } ]
   [ Overlap { On | Off } ]
   [ PartialSegments { On | Off } ]
   [ Duplicates { On | Off } ]
   [ Max [ number_of_labels ] ]
   [ Offset offset_amount ]
   [ LabelAlpha alpha_value ]
} ]
```

window_id is the integer window identifier of a Map window.

layer_id identifies which layer to modify; can be a SmallInt (for example, use 1 to specify the top map layer other than Cosmetic) or a string representing the name of a table displayed in the map.

labeloverride_index is an integer index (1-based) for the override definition within the layer. Each label override is tied to an zoom range and is ordered so that the smallest zoom range value is on top (index 1).

labeloverride_name is the user specified override name.

min_vis, *max_vis* are numbers specifying the minimum and maximum zoom at which the style override will come into effect.

dist_unit is a string expression, specifying the units for the map (such as “mi” for miles, “m” for meters; see **Set Distance Units statement** for a list of available unit names). This is an optional parameter. If not present, the distance units from the table's coordinate system are used.

label_expr is the expression to use for creating labels.

percent when curved labels are longer than the geometry they name, this is the amount (expressed as a percentage) of overhang permitted.

number_of_labels is an integer representing the maximum number of labels MapInfo Professional will display for the layer. If you omit the *number_of_labels* argument, there is no limit.

offset_amount is a number from zero to 200 (representing a distance in points), causing the label to be offset from its anchor point.

alpha_value is an integer value representing the alpha channel value for translucency. Values range from 0-255. 0 is completely transparent. 255 is completely opaque. Values between 0-255 make the labels display translucent.

Description

A label property zoom range sets up labeling properties that vary with the map's zoom level. There can be multiple label properties zoom ranges per layer. To change the labeling expression when zooming in on the map, or to see the label font grow larger when zooming in, set up multiple label properties zoom ranges.

The label zoom range turns off the labels if you zoom in or out too far. There can only be one of these per layer.

Line sets the type of call-out line, if any, that should appear when a label is dragged from its original location. You can specify **Line Simple**, **Line Arrow**, or **Line None**. For example:

Position controls label positions with respect to the positions of object centroids. For example, the following statement sets labels above and to the right of object centroids.

Auto Retry lets users to apply a placement algorithm that will try multiple label positions until a position is found that does not overlap any other label, or until all positions are exhausted.

- *When Writing Workspaces*, if the Auto Retry feature is On, we write Auto Retry On to the workspace after the Position clause (but the order isn't important), and increase the workspace version to 9.5 or later. If the feature is Off, we do not write anything to the workspace and do not increase the version number. A version 9.5 or later workspace can have Auto Retry Off in it, but we do not explicitly write it out, to avoid increasing the version unnecessarily.
- *When Reading Workspaces* If Auto Retry On or Auto Retry Off is in the workspace, it must be a version 9.5 or later workspace, otherwise a syntax error occurs. If Auto Retry is On, different positions are tried to place the label. If Auto Retry is Off, no retry is attempted - this is the default behavior. Overlap must be Off to enable the Auto Retry feature. If Overlap is On and Auto Retry On/Off are in the same label clause, the Auto Retry mechanism is initialized but ignored, so overlapping labels are allowed.

Font is a valid **Font clause** to specify a text style used in labels.

With specifies the expression used to construct the text for the labels. For example, the following statement specifies a labeling expression which uses the **Proper$( ) function** to control capitalization in the label.

Parallel controls whether labels for line objects are rotated, so that the labels are parallel to the lines.

Auto controls whether automatic labels display. If you specify **Auto Off**, automatic labels will not display, although user-edited labels will still display.

Overlap controls whether MapInfo Professional draws labels that would overlap existing labels. To prevent overlapping labels, specify **Overlap Off**.

PartialSegments controls whether MapInfo Professional labels an object when the object's centroid is not in the visible portion of the map. If you specify **PartialSegments On** (which corresponds to selecting the **Label Partial Objects** check box in MapInfo Professional), MapInfo Professional labels the visible portion of the object. If you specify **PartialSegments Off**, an object will only be labeled if its centroid appears in the Map window.

Duplicates controls whether MapInfo Professional allows two or more labels that have the same text. To prevent duplicate labels, specify **Duplicates Off**.

Max *number_of_labels* sets the maximum number of labels that MapInfo Professional will display for this layer. If you omit the *number_of_labels* argument, MapInfo Professional places no limit on the number of labels.

Offset *offset_amount* specifies an offset distance, so that MapInfo Professional automatically places each label away from the object's centroid. The *offset_amount* argument is an integer from zero to 50, representing a distance in points. If you specify **Offset 0** labels appear immediately adjacent to centroids. If you specify **Offset 10** labels appear 10 points away. The offset setting is ignored when the **Position** clause specifies centered text.

For more information about style overrides for layers, see **Modifying Layer Label Overrides** and **Enabling, Disabling, or Removing Overrides for Layer Labels**. See also, **LabelOverrideInfo( ) function**.

Examples

The following example overrides label style:

```
Set Map Layer 1 Label Zoom (1, 100000) with State
   Override Add Zoom (1, 1000) with State_Name
   Override Add Zoom (1000, 10000) with State
```

The following example adds a new style:

```
Set Map Layer 1 Label Override Add Zoom (10000, 100000)
   with State Overlap Off
```

Modifying Layer Label Overrides

When the add keyword is excluded from the Label Override clause it will modify the properties for an existing multiple label override definition within a layer specified by the integer index (1-based) or the override name.

```
Set Map [Layer layer_id
   [Label...
      [ MODIFYLABEL_OVERRIDE_CLAUSE ]
      [ MODIFYLABEL_OVERRIDE_CLAUSE ] ...
   ]
]
```

Where *MODIFYLABEL_OVERRIDE_CLAUSE* is:

```
[ Override { labeloverride_index | labeloverride_name } {
   [ Zoom ( min_vis, max_vis ) [ Units dist_unit ] ]
   [ Enable { On | Off } ]
   [ Line { Simple | Arrow | None } ]
   [ Position [ Center ] [ { Above | Below } ] [ { Left | Right } ] ]
   [ Auto Retry { On | Off } ]
   [ Font...] [ Pen... ]
   [ With label_expr ]
   [ Parallel { On | Off } ] [ Follow Path ] [Percent Over percent ]
   [ Auto { On | Off } ]
   [ Overlap { On | Off } ]
   [ PartialSegments { On | Off } ]
   [ Duplicates { On | Off } ]
   [ Max [ number_of_labels ] ]
   [ Offset offset_amount ]
   [ LabelAlpha alpha_value ]
} ]
```

window_id is the integer window identifier of a Map window.

layer_id identifies which layer to modify; can be a SmallInt (for example, use 1 to specify the top map layer other than Cosmetic) or a string representing the name of a table displayed in the map.

labeloverride_index is an integer index (1-based) for the override definition within the layer. Each label override is tied to an zoom range and is ordered so that the smallest zoom range value is on top (index 1).

labeloverride_name is the user specified override name.

min_vis, *max_vis* are numbers specifying the minimum and maximum zoom at which the style override will come into effect

dist_unit is a string expression, specifying the units for the map (such as “mi” for miles, “m” for meters; see **Set Distance Units statement** for a list of available unit names). This is an optional parameter. If not present, the distance units from the table's coordinate system are used.

label_expr is the expression to use for creating labels.

percent when curved labels are longer than the geometry they name, this is the amount (expressed as a percentage) of overhang permitted.

number_of_labels is an integer representing the maximum number of labels MapInfo Professional will display for the layer. If you omit the *number_of_labels* argument, there is no limit.

offset_amount is a number from zero to 200 (representing a distance in points), causing the label to be offset from its anchor point.

alpha_value is an integer value representing the alpha channel value for translucency. Values range from 0-255. 0 is completely transparent. 255 is completely opaque. Values between 0-255 make the labels display translucent.

Description

Line sets the type of call-out line, if any, that should appear when a label is dragged from its original location. You can specify **Line Simple**, **Line Arrow**, or **Line None**. For example:

Position controls label positions with respect to the positions of object centroids. For example, the following statement sets labels above and to the right of object centroids.

Auto Retry lets users to apply a placement algorithm that will try multiple label positions until a position is found that does not overlap any other label, or until all positions are exhausted.

- *When Writing Workspaces*, if the Auto Retry feature is On, we write Auto Retry On to the workspace after the Position clause (but the order isn't important), and increase the workspace version to 9.5 or later. If the feature is Off, we do not write anything to the workspace and do not increase the version number. A version 9.5 or later workspace can have Auto Retry Off in it, but we do not explicitly write it out, to avoid increasing the version unnecessarily.
- *When Reading Workspaces* If Auto Retry On or Auto Retry Off is in the workspace, it must be a version 9.5 or later workspace, otherwise a syntax error occurs. If Auto Retry is On, different positions are tried to place the label. If Auto Retry is Off, no retry is attempted - this is the default behavior. Overlap must be Off to enable the Auto Retry feature. If Overlap is On and Auto Retry On/Off are in the same LABELCLAUSE, the Auto Retry mechanism is initialized but ignored, so overlapping labels are allowed.

Font is a valid **Font clause** to specify a text style used in labels.

With specifies the expression used to construct the text for the labels. For example, the following statement specifies a labeling expression which uses the **Proper$( ) function** to control capitalization in the label.

Parallel controls whether labels for line objects are rotated, so that the labels are parallel to the lines.

Auto controls whether automatic labels display. If you specify **Auto Off**, automatic labels will not display, although user-edited labels will still display.

Overlap controls whether MapInfo Professional draws labels that would overlap existing labels. To prevent overlapping labels, specify **Overlap Off**.

PartialSegments controls whether MapInfo Professional labels an object when the object's centroid is not in the visible portion of the map. If you specify **PartialSegments On** (which corresponds to selecting the **Label Partial Objects** check box in MapInfo Professional), MapInfo Professional labels the visible portion of the object. If you specify **PartialSegments Off**, an object will only be labeled if its centroid appears in the Map window.

Duplicates controls whether MapInfo Professional allows two or more labels that have the same text. To prevent duplicate labels, specify **Duplicates Off**.

Max *number_of_labels* sets the maximum number of labels that MapInfo Professional will display for this layer. If you omit the *number_of_labels* argument, MapInfo Professional places no limit on the number of labels.

Offset *offset_amount* specifies an offset distance, so that MapInfo Professional automatically places each label away from the object's centroid. The *offset_amount* argument is an integer from zero to 50, representing a distance in points. If you specify **Offset 0** labels appear immediately adjacent to centroids. If you specify **Offset 10** labels appear 10 points away. The offset setting is ignored when the **Position** clause specifies centered text.

For more information about style overrides for layers, see **Adding Overrides for Layer Labels** and **Enabling, Disabling, or Removing Overrides for Layer Labels**. See also, **LabelOverrideInfo( ) function**.

Example

The following example modifies style:

```
Set Map Layer 1 Label Override 3 Zoom (1000, 10000)
   Line Arrow Pen (2, 1, 255)
```

Enabling, Disabling, or Removing Overrides for Layer Labels

If multilabel overrides are defined for a layer, they are enabled by default.

To enable or disable label overrides for a layer, use the Label Override clauses with either the On or Off option.

To remove an existing label override definition for a layer, use Label Override Remove clause.

Syntax: enable or disable overrides

```
Set Map
   [ Window window_id ]
   [ Layer layer_id
      [ Label [ Overrides { On | Off } ] ]
   ]
```

Syntax: remove overrides

```
Set Map
   [ Window window_id ]
   [ Layer layer_id
      [ Label [ Override Remove { All | labeloverride_index
         [, labeloverride_index ... ] } ]
      ]
   ]
```

window_id is the integer window identifier of a Map window.

layer_id identifies which layer to modify; can be a SmallInt (for example, use 1 to specify the top map layer other than Cosmetic) or a string representing the name of a table displayed in the map.

labeloverride_index is an integer index (1-based) for the override definition within the label. Each label override is tied to an zoom range and is ordered so that the smallest zoom range value is on top (index 1).

For more information about style overrides for layers, see **Adding Overrides for Layer Labels** and **Modifying Layer Label Overrides**. See also, **LabelOverrideInfo( ) function**.

Example

The following examples remove styles:

```
Set Map Layer 1 Label Override Remove 3
Set Map Layer 1 Label Override Remove All
```

Managing Group Layers

The following clauses affect group layers. Group properties are set like layer properties as part of the optional group layer clause in the **Set Map statement**.

For layer clauses, see **Managing Individual Layer Properties and Appearance**.

Syntax 1 (Group)

```
Set Map
   [ Window window_id ]
   [ GroupLayer group_id [ Display { On | Off } ]
      [ Title "new_ friendly_name" ] ]
```

Syntax 2 (Ungroup)

```
Set Map
   [ Window window_id ]
   [ GroupLayer group_id  [ Ungroup [ All ] ] ]
```

window_id is the integer window identifier of a Map window.

group_id can be either the numeric ID or a string name. The name would refer to the first group found in the list with that name.

Description

Ungroup removes the group layer but insert all the children of the group list into the parent list. *All* keyword will ungroup all nested group layers and insert all children into the parent list. Draw order will be maintained.

Examples

GroupLayer “Tropics” (group 1)
    Tropic_Of_Capricorn (layer 1)
    Tropic_Of_Cancer (layer 2)
Wgrid15 (layer 3)
GroupLayer “World Places” (group 2)
    WorldPlaces (layer 4)
    WorldPlacesMajor (layer 5)
    WorldPlaces_Capitals (layer 6)
Airports (layer 7)

`Set Map GroupLayer 1 Ungroup` results in this list (with group and layer ID's renumbered):

Tropic_Of_Capricorn (layer 1)
Tropic_Of_Cancer (layer 2)
Wgrid15 (layer 3)
GroupLayer “World Places” (group 2)
    WorldPlaces (layer 4)
    WorldPlacesMajor (layer 5)
    WorldPlaces_Capitals (layer 6)
Airports (layer 7)

whereas

Set Map GroupLayer 1 Ungroup All results in this list:

```
Tropic_Of_Capricorn (layer 1)
Tropic_Of_Cancer (layer 2)
Wgrid15 (layer 3)
WorldPlaces (layer 4)
WorldPlacesMajor (layer 5)
WorldPlaces_Capitals (layer 6)
Airports (layer 7)
```

Title will rename to the group layer to the string contained in *grouplayer_id_string*. The following renames group layer 2 in the Layer Control layer list as Hello World:

```
Set Map GroupLayer 2 Title “Hello World”
```

Ordering Layers

The following clauses move a layer and group layer to a specific location in the layer list.

Syntax

```
Set Map
   [ Window window_id ]
   [ Order layer_id, [ , layer_id ... ] ]
   [
      [ GroupLayers group_layer_id [, group_layer_id… ] ]
```

```
    [ Layers layer_id [, layer_id ] ] . . .
    [ DestGroupLayer group_layer_id [ Position position ] ]
]
```

window_id is the integer window identifier of a Map window.

layer_id is a number identifying a map layer to modify, according to that layer's original position in the map, where 1 (one) is the top-most layer number (the layer which draws last, and therefore always appears on top).

group_layer_id is a number identifying a group layer to modify, according to its original position in the map.

position is 1-based index within the destination group of where to insert the list of layers being moved. The default position is the first position in the group (position = 1).

Description

The Cosmetic layer is a special layer, with a layer number of zero. The Cosmetic layer is always drawn last; thus, a zero should not appear in an **Order** clause. For example: given a Map window with four layers (not including the Cosmetic layer), the following **Set Map** statement will reverse the order of the topmost two layers:

```
Set Map Order 2, 1, 3, 4
```

Set Map Order resets the order in which map layers are drawn. It moves layers, such as 3, 2, and 1 in the following example, to the top of the layer list, removing them from whatever group they might have been in.

```
Set Map Order Layers 3, 2, 1
```

However, using the GroupLayers and Layers clauses lets you specify moving layers and/or whole groups.

The optional **DestGroupLayer** specifies the group to insert the list of one or more layers and groups into, and at what position. This clause can also be used with the older syntax to specify the exact location to insert the layers. If missing, it means the groups and/or layers are inserted into the top level list at the first position (as it was assumed with the old syntax). However you can specify the top level list with a group ID = 0.

The **position** is the 1-based index within the destination group of where to insert the list of layers being moved. If the position is omitted it is assumed to be the first position in the group (position = 1).

If the position given exceeds the number of items in the destination group, the new layers and/or groups will be inserted at the end of the destination group.

Layer and group IDs may be the numeric ID or name. Group IDs range from 0 to the total number of groups in the list.

Once the list is reordered all IDs are renumbered sequentially from the top down.

Thematic layers and their reference base layer must always remain in a contiguous sequence, so Set Map Order will not allow you to insert layers within a set of thematic layers. If the Position specified would insert layers within a set of thematic layers, the layers will instead be inserted above or below the set, which ever is closest to the original Position.

Managing the Coordinate System of the Map

The following clauses affect the coordinate system of the map and distance type in use.

Syntax

```
Set Map
   [ Window window_id ]
   [ CoordSys... ]
   [ Distance Type { Spherical | Cartesian } ]
   [ XY Units  xy_unit  [ { Display  Decimal  {On | Off } |
     Display  Grid  [ { MGRS | USNG [ Datum datumid ] }  ] } ] ]
```

window_id is the integer window identifier of a Map window.

xy_unit is a string representing the name of an x/y coordinate unit (for example, “m” for meters, “degree” for degrees). If the **XY Units** are in degrees, the **Display Decimal** clause specifies whether to display in decimal degrees. Set to **On** to display in decimal degrees or **Off** to set in degrees, minutes, or seconds. Set **Display Grid** to display in Military grid reference format.

datumid is a numeric expression representing the datum id. It must evaluate to one of the following values:

```
DATUMID_NAD27  (62)
DATUMID_NAD83  (74)
DATUMID_WGS84 (104)
```

ⓘ DATUMID_* are defines in MapBasic.def. WGS84 and NAD83 are treated as equivalent.

Description

CoordSys clause Assigns the Map window a different coordinate system and projection. For details on the syntax of a CoordSys clause, see **CoordSys clause**.

The MapBasic coordinate system must be set explicitly with a **Set CoordSys statement** and can be retrieved with the **SessionInfo( ) function**.

ⓘ When a **Set Map** statement includes a CoordSys clause, the MapBasic application's coordinate system is automatically set to match the map's coordinate system.

This example only alters the map's coordinate system and units; the MapBasic coordinate system is unaffected:

```
Set Map XY Units "m" CoordSys Earth Projection 8,
33, "m", -55.5, 0, 0.9999, 304800, 0
```

Distance Type is either **Spherical** or **Cartesian**. All distance, length, perimeter, and area calculations for objects contained in the Map window will be performed using one of these calculation methods. Note that if the coordinate system of the Map window is NonEarth, then the calculations will be performed using Cartesian methods regardless of the option chosen, and if the coordinate system of the Map window is Latitude/Longitude, then calculations will be performed using Spherical methods regardless of the option chosen.

XY Units specifies the type of coordinate unit used to display x-, y-coordinates (for example, when the user has specified that the map should display the cursor position on the status bar). The unit name can be “degree” (for degrees longitude/latitude) or a distance unit such as “m” for meters.

If the **XY Units** are in degrees, the **Display Decimal** clause specifies whether to display in decimal degrees (**On**) or in degrees, minutes, seconds (**Off**). **Display Grid** will display coordinates in Military Grid reference system format no matter how the **XY Units** are specified.

```
Set Map XY Units "m" Display Grid
Set Map XY Units "degree" Display Grid
Set Map XY Units "degree" Display Decimal On
Set Map XY Units "degree" Display Decimal Off
```

The following statement specifies meters as the coordinate unit:

```
Set Map XY Units "m"
```

Managing Image Properties

The following clauses affect image reprojection and resampling.

Syntax

```
Set Map
   [ Window window_id ]
   [ Image Reprojection { None | Always | Auto } ]
   [ Image Resampling { CubicConvolution | NearestNeighbor } ]
```

window_id is the integer window identifier of a Map window.

Description

Image Reprojection has three options, **Always** and **Auto** (for Automatic) and **None**.

- **None** means that MapInfo Professional treats raster layers as it has in pre-version 8.5 versions by conforming the vector layers to the raster layer.
- **Always** means that reprojection is always done; specifically, coordinates are calculated using precise formulae and pixels are resampled using “cubic convolution” or “nearest neighbor”.
- **Auto** means that use of reprojection is decided based on how the destination image rectangle looks after having been transformed into the source image space. If it looks as a “rigorous” rectangle (two sides are parallel to x-axis and two sides parallel to y-axis), then the old MapInfo Professional code works, for example standard Windows functions are used for only stretching the source image in both directions. This is the fastest way of drawing resulting images. If the above is not the case (stretching is not enough because of non-linearities and/or skew of the destination image rectangle transformed into the source image space), the reprojection code works.

Image Resampling has two options, **Cubic Convolution** and **Nearest Neighbor**.

- **CubicConvolution** is a method of resampling images providing for the best “restoration” of pixel values unavailable in a source image (because of its discreteness). Here, a pixel of the destination image is calculated based on the pixel values in a 4x4 window centered at the “basic” pixel in the source image. The coordinates (real numbers, in general) of the basic

pixel are calculated for every pixel of the destination image based on special optimized procedure. Pixels within the above window are weighted in a special way based on the mantissas of basic pixel coordinates.

- **NearestNeighbor** is a method of resampling images by merely putting the value of the basic pixel from a source image into the current pixel position.

Managing Hotlinks

The hotlink settings are persisted via the **Set Map statement** Layer Activate clause, which supports multiple hotlink definitions. This includes the ability to add new items, modify the attribute of existing items, remove and reorder items. For a discussion of how MapInfo Professional supports legacy syntax, see **Exceptions to Support Backwards Compatibility**.

Purpose

The purpose of Activate is to allow you to define new hotlinks. You use a hotlink to launch a file or a URL from a Map window.

Syntax

```
Set Map
[ Window window_id ]
[ Layer layer_id
  [ Activate LAYER_ACTIVATE_CLAUSES ] ]
```

Where *LAYER_ACTIVATE_CLAUSES* is:

```
Using launch_expr  [ On  {  Labels  |  Objects | Labels Objects } ]
   [ Relative Path { On | Off } ] [ Enable { On | Off }]
[ Alias expression ]
[,LAYER_ACTIVATE_CLAUSES ]
```

window_id is the integer window identifier of a Map window.

layer_id identifies which layer to modify; can be a SmallInt (for example, use 1 to specify the top map layer other than Cosmetic) or a string representing the name of a table displayed in the map.

launch_expr is an expression that will resolve to the name of the file to launch when the object is activated.

expression the placeholder of the actual file name expression being set (any URL or filename).

Description

Relative Path lets you define links to files stored in locations relative to the tables. For example: if the table C:\DATA\STATES.TAB contains HotLinks to workspace files that are stored in directories under c:\data. The workspace file for New York, NEWYORK.WOR, is stored in c:\data\ny and the

HotLink associated with New York is “NY\NEWYORK.WOR”. Setting Relative Path to **On** tells MapInfo Professional to prefix the HotLink string with the location of the .tab file, in this case resulting in the launch string “C:\DATA\NY\NEWYORK.WOR”.

> ⓘ HotLinks identified as URLs are not modified before launch, regardless of the **Relative Path** setting. The ShellAPI function path's URL is used to determine if a HotLink is a URL.

Enable clause has two options **On** and **Off**. When set to **On**, it enables the hotlink definition and when set to **Off**, it disables the hotlink definition.

For an individual hotlink the Enable clause allow the user to "turn off" a hotlink while preserving the definition. (In versions prior to 10.0, the user disabled the hotlink by setting the expression to "", losing the original expression.)

An active object is an object in a Map window that has a URL or filename associated with it. Clicking on an active object with the HotLink Tool will launch the associated URL or file. For example, if the string `http://www.boston.com` is associated with a point object on the map, then clicking the point, or its label, will result in the default browser being started with the site `http://www.boston.com`. You can associate other types of files with map objects; MapInfo workspace (.wor), table (.tab) or application (.mbx) files, Word documents (.doc), executable files (.exe), etc. Any type of file that the system knows how to “launch” can be associated with a map object. From version 10.0 onwards another clause “Alias” has been added for hotlinks. This alias clause is used to set an expression, which will basically be the placeholder of the actual File Name Expression being set. In the current hotlinks implementation, the FileName Expression can be set to any URL or filename. It has been found that URL’s can be very long and hence when an user clicks on an active Hotlinks object having multiple hotlink definitions, it becomes difficult to show the lengthy URL’s in the popup window. To solve this problem, the Alias Expression has been added to the GUI. The similar work is performed by the Alias keyword in MapBasic. When an active object has multiple hotlink definitions, if you set the Alias Expression to a valid expression, then the popup window shows the Alias Name, instead of the lengthy URL.

> ⓘ The Alias expression is displayed in the popup window, only if it is set to something other than the default value “None”. Thus hotlink definitions can have Alias expression set or not. Any hotlink created using MapBasic without the alias keyword, will have the Alias Expression in the GUI have a value of “None”.

This version of the command wipes out any existing definitions and creates one or more new definitions. The **Using** clause is required and *launch_expr* must not be an empty string (for example, ""). When the **Enable** clause is included and set to Off, the hotlink definition will be disabled.

The **On**, **Relative Path**, **Enable** and **Alias** clauses are optional.

For more information about Hotlinks, see **Adding New HotLink Definitions**, **Modifying Existing HotLink Definitions**, **Removing HotLink Definitions**, and **Reordering HotLink Definitions**.

Exceptions to Support Backwards Compatibility

The Using clause can be omitted, but only from the first HotLink definition. The Using expression can be empty (""), but only for the first HotLink definition.

No *Using* Clause

Both of the following commands omit the *Using* clause, and in 850 this has the consequence of updating the properties of the one/only hotlink def, even if the user has never issues a command to set the *Using* clause. As of 900 these commands are a problem because map layers are created without any hotlink definitions.

- Activate On Objects
- Activate Relative Path On

To solve this problem, MapInfo Professional allows empty expressions, but only for the first hotlink definition. As was the case in pre-900 versions, a hotlink with an empty expression is effectively disabled. Omitting the *Using* clause, generates an error unless the command originates from a pre-900 application or workspace.

Empty *Using* clause

The following command sets the hotlink expression to an empty string, which essentially disables hotlink capability for the layer. In fact, the default launch expression is the empty string, so the hotlink definition has no affect until the expression is set to a non-empty string. This works as a way to enable/disable a hotlink in 850. In 900 we support the notion of enabling/disabling via explicit syntax in the Set map Layer Activate Enable On/Off command, and don't really want to support hotlink definition with an empty expression string.

```
Set Map Layer 1 Activate Using ""
```

This statement allows empty expressions, but only for the first hotlink definition. As was the case in pre-9.0 versions, a hotlink with an empty expression is effectively disabled.

When a Set Map Layer Activate command is encounter with no Using clause or an empty Using clause, the action depends on the current state of the layer's hotlinks.

The table following contains examples of different scenarios.

Action	Number of HotLinks	Result
Activate Using " "	One or More	Sets the first hotlink's expression to empty string. The definition is effectively disabled until the expression is set to a non-empty value.
Activate Using " "	Zero	Creates a new hotlink definition and sets its expression to empty. The definition is effectively disabled until the expression is set to a non-empty value.

Activate On Objects	One or More	If issued from a pre-9.0 application or workspace: • Update the first hotlink definition with the values specified in the command. Does not affect the expression. If issued from a 9.0 (or later) application or workspace: • Generates a "Missing required Using clause" syntax error
Activate On Object	Zero	If issued from a pre-9.0 application or workspace: • Creates a new hotlink definition, set its expression to empty and apply the values specified in the command. The definition will effectively be disabled until the expression is set to a non-empty value. If issued from a 9.0 (or later) application or workspace • Generates a "Missing required Using clause" syntax error

Note that the same actions will apply when reading in table metadata.

Example

```
Set Map Layer 1 Activate Using Url1 On Objects Relative Path Off Enable
On, Using Url2 On Objects Relative Path On Enable On
```

Adding New HotLink Definitions

The following clause adds a new hotlink definition to the map. For a detailed description of the Activate clause, see **Managing Hotlinks**.

Syntax

```
Set Map
[ Window window_id ]
[ Layer layer_id
  [ Activate Add [First] LAYER_ACTIVATE_CLAUSES ] ]
```

Where *LAYER_ACTIVATE_CLAUSES* is:

```
Using launch_expr  [ On  {  Labels  |  Objects | Labels Objects } ]
[ Relative Path { On | Off } ] [ Enable { On | Off }]
[ Alias expression ]  [,LAYER_ACTIVATE_CLAUSES ]
```

window_id is the integer window identifier of a Map window.

layer_id identifies which layer to modify; can be a SmallInt (for example, use 1 to specify the top map layer other than Cosmetic) or a string representing the name of a table displayed in the map.

launch_expr must not be an empty string (for example, "").

expression the placeholder of the actual file name expression being set (any URL or filename).

Description

First is optional to insert the new items at the beginning of the list.

Enable clause has two options **On** and **Off**. When set to **On**, it enables the hotlink definition and when set to **Off**, it disables the hotlink definition.

Examples

```
Set Map Layer 1 Activate Add Using URL1 On Objects Relative Path On Alias
URL, Using URL2 On Objects Enabled Off
Set Map Layer 1 Activate Add First Using URL1 On Objects
```

Notes

MapBasic 9.0 and later of this command contain a hotlink def at the end of the hotlink list.

Modifying Existing HotLink Definitions

The following clause modifies hotlink definition on the map.For a detailed description of the Activate clause, see **Managing Hotlinks**.

Syntax

```
Set Map
[ Window window_id ]
[ Layer layer_id
 [ Activate Modify MODIFY_CLAUSES ] ]
```

Where *MODIFY_CLAUSES* is:

```
hotlink_id  { [ Using launch_expr ]
   [ On  {  Labels  |  Objects | Labels Objects } ]
   [ Relative Path { On | Off } ] [ Enable { On | Off }]
   [ Alias expression ] }
[, MODIFYCLAUSE ]
```

window_id is the integer window identifier of a Map window.

layer_id identifies which layer to modify; can be a SmallInt (for example, use 1 to specify the top map layer other than Cosmetic) or a string representing the name of a table displayed in the map.

hotlink_id is an integer index (1-based) that specifies the hotlink definition to modify. At least one *hotlink_id* must be specified.

launch_expr must not be an empty string (for example, "").

expression the placeholder of the actual file name expression being set (any URL or filename).

Description

Enable clause has two options **On** and **Off**. When set to **On**, it enables the hotlink definition and when set to **Off**, it disables the hotlink definition.

Examples

```
Set Map Layer 1 Activate Modify 1 Using URL1 On Objects Alias URL, 2
Relative Path Off
Set Map Layer 1 Activate Modify 2 On Objects, 4 On Labels
Set Map Layer 1 Activate Modify 3 Relative Path On Enable Off
Set Map Layer 1 Activate Modify 2 Enable Off, 3 Enable On
```

Removing HotLink Definitions

The following clause removes a new hotlink definition from the map.For a detailed description of the Activate clause, see **Managing Hotlinks**.

Syntax

```
Set Map
[ Window window_id ]
[ Layer layer_id
   [Activate Remove {
      All | hotlink_id  [ , hotlink_id, hotlink_id, ... ]
   } ]
]
```

window_id is the integer window identifier of a Map window.

layer_id identifies which layer to modify; can be a SmallInt (for example, use 1 to specify the top map layer other than Cosmetic) or a string representing the name of a table displayed in the map.

hotlink_id is an integer index (1-based) that specifies the hotlink definition to modify. At least one *hotlink_id* must be specified.

Description

All specifies that all hotlink definitions are removed.

Examples

```
Set Map Layer 1 Activate Remove 2, 4
Set Map Layer 1 Activate Remove All
```

Reordering HotLink Definitions

The following clause reorders hotlink definitions on the map.For a detailed description of the Activate clause, see **Managing Hotlinks**.

Syntax

```
Set Map
[ Window window_id ]
[ Layer layer_id
   [Activate Order  hotlink_id [ ,hotlink_id, hotlink_id, ... ] ] ]
```

window_id is the integer window identifier of a Map window.

layer_id identifies which layer to modify; can be a SmallInt (for example, use 1 to specify the top map layer other than Cosmetic) or a string representing the name of a table displayed in the map.

hotlink_id is an integer index (1-based) that specifies the hotlink definition to modify. At least one *hotlink_id* must be specified.

Example

```
Set Map Layer 1 Activate Order 2, 3, 1
```

Set Map3D statement

Purpose

Change the settings of an existing 3DMap window. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Set Map3D
   [ Window window_id ]
   [ Camera [ Zoom factor | Pitch angle | Roll angle | Yaw angle |
     Elevation angle Position (x,y,z) | FocalPoint (x,y,z) ] ]
   [ Orientation ( vu_1, vu_2, vu_3, vpn_1, vpn_2, vpn_3,
     clip_near, clip_far )]]
   [ Light [ Position ( x, y, z | Color lightcolor ] ]
   [ Resolution ( res_x, res_y ) ]
   [ Scale grid_scale ]
   [ Background backgroundcolor ]
   [ Refresh ]
```

mapper_creation_string specifies a command string that creates the mapper textured on the grid.

factor specifies the amount to set the zoom.

angle is an angle measurement in degrees. The horizontal angle in the dialog box ranges from 0-360 degrees and rotates the maps around the center point of the grid. The vertical angle in the dialog box ranges from 0-90 and measures the rotation in elevation from the start point directly over the map.

res_x, *res_y* is the number of samples to take in the x- and y-directions. These values can increase to a maximum of the grid resolution. The resolution values can increase to a maximum of the grid x,y dimension. If the grid is 200x200 then the resolution values will be clamped to a maximum of 200x200. You cannot increase the grid resolution, only specify a subsample value.

grid_scale is the amount to scale the grid in the z-direction. A value >1 will exaggerate the topology in the z-direction, a value < 1 will scale down the topological features in the z-direction.

backgroundcolor is a color to be used to set the background and is specified using the **RGB( ) function**.

Description

The **Set Map3D** statement changes the settings of an already created 3D Map. If the original tables from which the 3D Map was created were modified either by adding labels or by modifying geometry, **Refresh** will capture the changes in the mapper and recreate the 3D map based on those changes.

Camera specifies the camera position and orientation.

Pitch adjusts the camera's current rotation about the x axis centered at the camera's origin.

Roll adjusts the camera's current rotation about the z axis centered at the camera's origin.

Yaw adjusts the camera's current rotation about the y axis centered at the camera's origin.

Elevation adjusts the current camera's rotation about the X Axis centered at the camera's focal point.

Position indicates the camera/light position.

FocalPoint indicates the camera/light focal point.

Orientation specifies the cameras ViewUp (*vu_1*, *vu_2*, *vu_3*), ViewPlane Normal (*vpn_1*, *vpn_2*, *vpn_3*) and Clipping Range (*clip_near*, *clip_far*), used specifically for persistence of view.

Resolution is the number of samples to take in the x- and y-directions. These values can increase to a maximum of the grid resolution. The resolution values can increase to a maximum of the grid x,y dimension. If the grid is 200x200 then the resolution values will be clamped to a maximum of 200x200. You can't increase the grid resolution, only specify a subsample value.

Units specifies the units the grid values are in. Do not specify this for unit-less grids (for example, grids generated using temperature or density). This option needs to be specified at creation time. If there are units associated with your grid values, they have to specified when you create the 3DMap. You cannot change them later with **Set Map3D** or the Properties dialog box.

Refresh regenerates the texture from the original tables.

Example

```
Dim win3D as Integer
Create Map3D Resolution(75,75) Resolution(100,100) Scale 2 Background
RGB(255,0,0)
win3D = FrontWindow( )
Set Map3D Window win3D Resolution(150,100) Scale 0.75 Background
RGB(255,255,0)
Changes the original 3DMap window's resolution in the X and Y, the scale
to de-emphasize the grid in the Z direction (< 1) and change the
background color to yellow.
```

See Also:

Create Map3D statement, **Map3DInfo( ) function**

Set Next Document statement

Purpose

Re-parents a MapInfo Professional document window (for example, so that a Map window becomes a child window of a Visual Basic application). You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Set Next Document
   { Parent HWND | Style style_flag | Parent HWND Style style_flag }
```

HWND is an integer window handle, identifying a parent window.

style_flag is an integer code (see table below), indicating the window style.

Description

This statement is used in Integrated Mapping applications. For an introduction to Integrated Mapping, see the *MapBasic User Guide*.

To re-parent an MapInfo Professional window, issue a **Set Next Document** statement, and then issue one of these window-creation statements: **Map statement**, **Browse statement**, **Graph statement**, **Layout statement**, or **Create Legend statement**.

Include the **Parent** clause to identify an existing window, which will become the parent of the MapInfo Professional window you are about to create. Include the **Style** clause to specify a window style. If you are creating a document window, such as a Map window, include both clauses.

The *style_flag* argument must be one of the codes from the following table; codes are defined in MAPBASIC.DEF.

style_flag code	ID	Effect on the next document window:
WIN_STYLE_STANDARD	0	This code resets the style flag to its default value. If you issue a **Set Next Document Style 1** statement, but then you change your mind and do not want to use the child window style, issue a **Set Next Document Style 0** statement to reset the style.
WIN_STYLE_CHILD	1	Next window is created as a child window.
WIN_STYLE_POPUP_FULLCAPTION	2	Next window is created as a popup window, but with a full-height title bar caption.
WIN_STYLE_POPUP	3	Next window is created as a popup window with a half-height title bar caption.

The parent and style settings remain in effect until you create a new window. The new window adopts the parent and style settings you specified; then MapInfo Professional reverts to its default parent and style settings for any subsequent windows. To re-parent more than one window, issue a separate **Set Next Document** statement for each window you will create.

The **Create ButtonPad statement** resets the parent and style settings, although the new ButtonPad is not re-parented.

This statement re-parents document windows. To re-parent dialog box windows, use the **Set Application Window statement**. To re-parent special windows such as the Info window, use the **Set Window statement**.

Example

The sample program LEGENDS.MB uses the following statements to create a Theme Legend window inside of a Map window.

```
Dim win As Integer
win = FrontWindow( )
...
Set Next Document
   Parent WindowInfo(win, WIN_INFO_WND)
   Style 1
Create Legend From Window win
```

See Also:

Set Application Window statement, **Set Window statement**

Set Paper Units statement

Purpose

Sets the paper unit of measure that describes screen window sizes and positions. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

`Set Paper Units` *unit*

unit is a string representing the name of a paper unit (for example, “cm” for centimeters).

Description

The **Set Paper Units** statement changes MapBasic's paper unit of measure.

Paper units are small units of linear measure, such as “mm” (millimeters). MapBasic's uses “in” (inches) as the default paper unit; this remains MapBasic's paper unit unless a **Set Paper Units** statement is issued.

Some MapBasic statements (for example, the **Set Window statement**) include **Position**, **Width**, and **Height** clauses, through which a MapBasic program can reset the size or the position of windows on the screen. The numbers that you specify in **Position**, **Width**, and **Height** clauses use MapBasic's paper units. For example, the **Set Window statement** `Set Window Width 5` resets the width of a window. The window's new width depends on the paper unit in use; if MapBasic is currently using “in” as the paper unit, the **Set Window statement** makes the Map five inches wide.

If MapBasic is currently using “cm” as the paper unit, the **Set Map statement** makes the Map five centimeters wide.

MapBasic's paper unit is internal, and invisible to the end-user. When a user performs an operation which displays a paper measurement, the unit of measure displayed on the screen is independent of MapBasic's internal paper unit.

The unit parameter must be one of the values listed in the following table:

Unit name	Paper unit represented
“cm”	Centimeters
“in”	Inches
“mm”	Millimeters
“pt”	Points
“pica”	Picas

See Also:

Set Area Units statement, **Set Distance Units statement**

Set Path statement

Purpose

Allows user to change programmatically the path of a special MapInfo Professional directory defined initially in the Preferences dialog to access specific MapInfo files. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

`Set Path` *`current_path_id path`*

current_path_id is one of the following values:

```
PREFERENCE_PATH_TABLE (0)
PREFERENCE_PATH_WORKSPACE (1)
PREFERENCE_PATH_MBX (2)
PREFERENCE_PATH_IMPORT (3)
PREFERENCE_PATH_SQLQUERY (4)
PREFERENCE_PATH_THEMETHEMPLATE (5)
```

```
PREFERENCE_PATH_MIQUERY (6)
PREFERENCE_PATH_NEWGRID (7)
PREFERENCE_PATH_CRYSTAL (8)
PREFERENCE_PATH_GRAPHSUPPORT (9)
PREFERENCE_PATH_REMOTETABLE (10)
PREFERENCE_PATH_SHAPEFILE (11)
PREFERENCE_PATH_WFSTABLE (12)
PREFERENCE_PATH_WMSTABLE (13)
```

path is a string value, indicating the directory or folder to be used for these files.

Description

Set Path statement given the ID of a special MapInfo Preference directory allows to set it programmatically. An example of a special MapInfo directory is the default location to which MapInfo Professional writes out new native MapInfo tables.

Example

```
include "mapbasic.def"
declare sub main
sub main
Set Path PREFERENCE_PATH_WORKSPACE "C:\Temp\"
Print GetCurrentPath$(PREFERENCE_PATH_WORKSPACE)
end sub
```

See Also:
GetPreferencePath$( ) function

Set PrismMap statement

Purpose

Changes the settings of an existing Prism Map window. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Set PrismMap
   [Window window_id ]
   [ Camera [ Zoom factor | Pitch angle | Roll angle | Yaw angle |
     Elevation angle Position (x,y,z) | FocalPoint (x,y,z) ] ]
   [ Orientation ( vu_1, vu_2, vu_3, vpn_1, vpn_2, vpn_3,
     clip_near, clip_far )]]
   [ Light [ Position (x,y,z) | Color lightcolor ] ]
   [ Scale grid_scale ]
   [ Background backgroundcolor ]
   [ Label With infotips_expr ]
   [ Refresh ]
```

window_id is a window identifier a for a mapper window which contains a Grid layer. An error message is displayed if a Grid layer is not found.

mapper_creation_string specifies a command string that creates the mapper textured on the grid.

Camera specifies the camera position and orientation.

angle is an angle measurement in degrees. The horizontal angle in the dialog box ranges from 0-360 degrees and rotates the maps around the center point of the grid. The vertical angle in the dialog box ranges from 0-90 and measures the rotation in elevation from the start point directly over the map.

Pitch adjusts the camera's current rotation about the X-Axis centered at the camera's origin.

Roll adjusts the camera's current rotation about the Z-Axis centered at the camera's origin.

Yaw adjusts the camera's current rotation about the Y-Axis centered at the camera's origin.

Elevation adjusts the current camera's rotation about the X-Axis centered at the camera's focal point.

Position indicates the camera or light position.

FocalPoint indicates the camera or light focal point.

Orientation specifies the cameras ViewUp (*vu_1*, *vu_2*, *vu_3*), ViewPlane Normal (*vpn_1*, *vpn_2*, *vpn_3*), and Clipping Range (*clip_near*, *clip_far*), used specifically for persistence of view.

backgroundcolor is a color to be used to set the background and is specified using the **RGB( ) function**.

infotips_expr is the expression to use for InfoTips.

Refresh regenerates the texture from the original tables.

Description

The **Set PrismMap** statement changes the settings of an already created Prism Map.

Example

The following example changes the original PrismMap window's resolution in the x and y, the scale to de-emphasize the grid in the z-direction (< 1) and changes the background color to yellow.

```
Dim win3D as Integer
Create PrismMap Resolution(75,75) Resolution(100,100) Scale 2 Background
RGB(255,0,0)
win3D = FrontWindow( )
Set PrismMap Window win3D Resolution(150,100) Scale 0.75 Background
RGB(255,255,0)
```

See Also:

Create PrismMap statement, **PrismMapInfo( ) function**

Set ProgressBars statement

Purpose

Disables or enables the display of progress-bar dialog boxes. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Set ProgressBars { On | Off }
```

Description

Some MapBasic statements, such as the Create Object As Buffer statement, automatically display a progress-bar dialog box (a “percent complete” dialog box showing a horizontal bar and a **Cancel** button). To suppress progress-bar dialog boxes, use the **Set ProgressBars Off** statement. By suppressing these dialog boxes, you guarantee that the user will not interrupt the operation by clicking the **Cancel** button. To resume displaying progress-bar dialog boxes, use the **Set ProgressBars On** statement.

If you issue a **Set ProgressBars Off** statement from within a compiled MapBasic application (MBX file), the statement only disables progress-bar dialog boxes caused by the MBX file. Actions taken by the user can still cause progress bars to display. Also, **Run Menu Command statement**s can still cause progress bars to display, because the **Run Menu Command statement** simulates the user selecting a menu command.

To disable progress-bar dialog boxes that are caused by user actions or **Run Menu Command statement**s, type a **Set ProgressBars Off** statement into the MapBasic window (or send the command to MapInfo Professional through OLE Automation or DDE).

If your application minimizes MapInfo Professional (using the Set Window MapInfo Min), you should suppress progress bars. When a progress bar displays while MapInfo Professional is minimized, the progress bar is frozen for as long as MapInfo Professional is minimized. If you suppress the display of progress bars, the operation can proceed, even if MapInfo Professional is minimized.

See Also:

ProgressBar statement, **Run Menu Command statement**

Set Redistricter statement

Purpose

Changes the characteristics of a districts table during a redistricting session. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax 1 (Change)

```
Set Redistricter districts_table
   [ Change district_name
```

```
      [ To new_district_name ] [ Pen... ] [ Brush... ] [ Symbol... ] ]
   [ Add new_district_name [ Pen... ] [ Brush... ] [ Symbol... ] ]
   [ Remove district_name ]
```

Syntax (Order)

```
Set Redistricter districts_table
   Order { "Alpha" | "MRU" | "Unordered" }
```

Syntax 3 (Percentage)

```
Set Redistricter districts_table
   Percentage from { column | row }
```

districts_table is the name of the districts table (for example, Districts).

district_name is a string representing the name of an existing district.

new_district_name is a string representing a new district name, used when adding a district or renaming an existing district.

Pen is a valid **Pen clause** to specify a line style. For example, **Pen MakePen** (*width*, *pattern*, *color*).

Brush is a valid **Brush clause** to specify fill style. For example, **Brush MakeBrush** (*pattern*, *forecolor*, *backcolor*).

Symbol is a valid **Symbol clause** to specify a point style. For example, **Symbol MakeSymbol** (*shape*, *color*, *size*).

Description

Set Redistricter modifies the set of districts that are in use during a redistricting session. To begin a redistricting session, use the **Create Redistricter statement**. For an introduction to redistricting, see the MapInfo Professional documentation.

To add, delete, or modify a district or districts, use Syntax 1. Use the **Change** clause to change the name and/or the graphical style associated with a district. Use the **Add** clause to add a new district. Use the **Remove** clause to remove an existing district; when you remove a district, map objects which had been assigned to that district are re-assigned to the “all others” district.

The *district_name* and *new_district_name* parameters must always be string expressions, even if the district column is numerical. For example, to refer to the district representing the number 33, specify the string expression “33”.

To affect the ordering of the rows in the Districts Browser, use Syntax 2. Specify “Alpha” to use alphabetical ordering. Specify “MRU” if you want the most recently used district to appear on the top row of the Districts Browser. Specify “Unordered” if you want districts to be added to the bottom row of the Districts Browser as they are added.

Examples

Once a redistricting session is in effect, the following statement creates a new district.

```
Set Redistricter Districts
   Add "NorthWest" Brush MakeBrush(2, 255, 0)
```

The following statement renames the “NE” district to “NorthEast.” Note that this type of change can affect the table that is being redistricted. Initially, any rows belonging to the “NE” district have “NE” stored in the district column. After the **Set Redistricter… Change** statement, each of those rows has “NorthEast” stored in that column.

```
Set Redistricter Districts
   Change "NE" To "NorthEast"
```

The following statement removes the “NorthWest” district from the Districts table:

```
Set Redistricter Districts
   Remove "NorthWest"
```

The following statement sets the ordering of rows in the Districts Browser, so that the most recently used districts appear at the top:

```
Set Redistricter Districts
   Order "MRU"
```

See Also:

Create Redistricter statement

Set Resolution statement

Purpose

Sets the object-editing resolution setting; this controls the number of nodes assigned to an object when an object is converted to another object type. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Set Resolution *node_limit*

node_limit is a SmallInt value between 2 and 1,048,570 (inclusive); default is 100.

Description

By default, MapInfo Professional assigns 100 nodes per circle when converting a circle or arc into a region or polyline. Use the **Set Resolution** statement to alter the number of nodes per circle. By increasing the resolution setting, you can produce smoother result objects.

The **Set Resolution** statement affects subsequent operations performed by the user, such as the **Objects > Convert to Regions** command and the **Objects > Convert to Polylines** command. The resolution setting also affects some MapBasic statements and functions, such as the **ConvertToRegion( ) function** and the **ConvertToPline( ) function**. The resolution setting also affects operations where MapInfo Professional performs automatic conversion (for example, Split, Combine).

Buffering operations are not affected by the **Set Resolution** statement. The **Create Object As Buffer statement** and the **Buffer( ) function** both have resolution parameters which allow you to specify buffer resolution explicitly.

See Also:

ConvertToPline( ) function, **ConvertToRegion( ) function**

Set Shade statement

Purpose

Modifies a thematic map layer. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Set Shade
   [ Window window_id ] { map_layer_id | "table ( theme_layer_id )" }
      [ Style Replace { On | Off } ]
   ...
```

window_id is an integer window identifier.

map_layer_id is a SmallInt value, representing the layer number of a thematic layer.

table is the name of the table on which a thematic layer is based.

theme_layer_id is a SmallInt value, one or larger, representing which thematic layer to modify (for example, one represents the first thematic layer created).

Description

After you use the **Shade statement** to create a thematic map layer, you can use the **Set Shade** statement to modify the settings for that thematic layer. Issuing a **Set Shade** statement is analogous to choosing **Map > Modify Thematic Map**. The syntax of the **Set Shade** statement is identical to the syntax of the **Shade statement**, except for the way that the **Set Shade** statement identifies a map layer. A **Set Shade** statement can identify a layer by its layer number, as shown below:

```
Set Shade
   Window i_map_winid
   2
   With Num_Hh_90
   Graduated 0.0:0 11000000:24 Vary Size By "SQRT"
```

Or a **Set Shade** statement can identify a map layer by referring to the name of a table (the base table on which the layer was based), followed by a number in parentheses:

```
Set Shade
   Window i_map_winid
   "States(1)"
   With Num_Hh_90
   Graduated 0.0:0 11000000:24 Vary Size By "SQRT"
```

The number in parentheses represents the number of the thematic layer. To modify the first thematic layer that was based on the States table, specify States(1), etc.

Style Replace On (default) specifies the layers under the theme are not drawn.

Style Replace Off specifies the layers under the theme are drawn, allowing for multi-variate transparent themes.

Style Replace On is the default and provides backwards compatibility with the existing behavior so that the underlying layers are not drawn.

See Also:

Shade statement

Set Style statement

Purpose

Resets the current Pen, Brush, Symbol, or Font style. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Set Style
   { Brush... | Font... | Pen... |
     BorderPen | LinePen | Symbol... }
```

Font is a valid **Font clause** to specify a text style.

Pen is a valid **Pen clause** to specify a line style.

Brush is a valid **Brush clause** to specify fill style.

Symbol is a valid **Symbol clause** to specify a point style.

BorderPen takes a **Pen clause** which specifies a border line style.

LinePen takes a **Pen clause** which specifies a line style.

Description

The **Set Style** statement resets the **Pen**, **Brush**, **Symbol**, or **Font** style currently in use.

The **Pen clause** sets both the line and border pen. To set them individually, use the **LinePen** clause to set the line and the **BorderPen** clause to set the border. When the user draws a new graphical object to a Map or Layout window, MapInfo Professional creates the object using whatever **Font**, **Pen**, **Brush**, and/or **Symbol** styles are currently in use.

Example

Example of **Brush**, **Symbol**, and **Font**:

```
Include "mapbasic.def"
Set Style Brush MakeBrush(64, CYAN, BLUE)
Set Style Symbol MakeSymbol( 9, BLUE, 14)
Set Style Font MakeFont("Arial", 1, 14, BLACK,WHITE)
```

Example of **Pen**:

In this example, the line pen and the border pen are red.

```
Include "mapbasic.def"
Set Style Pen MakePen(3, 9, RED)
```

Example of **LinePen** and BorderPen:

In this example, the line pen is red and the border pen is green.

```
Include "mapbasic.def"
Set Style LinePen MakePen(6, 77, RED)
Set Style BorderPen MakePen(6, 77, GREEN)
```

See Also:

CurrentBrush( ) function, **CurrentFont( ) function**, **CurrentPen( ) function**, **CurrentSymbol( ) function**, **MakeBrush( ) function**, **MakeFont( ) function**, **MakePen( ) function**, **MakeSymbol( ) function**, **RGB( ) function**

Set Table Datum statement

Purpose

Writes the datum index information for native tables into the map file, including the ellipsoid index, 3 shift parameters, 3 rotation parameters, scale parameter. Because some datums are identical, this statement keeps the datum index along with all datum parameters in memory and writes it into mapfile. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Set Table *table_name* **datum** *datum_number*

See Also:

Set Datum Transform Version statement

Set Table statement

Purpose

Configures various settings of an open table. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Set Table tablename
   [ FastEdit { On | Off } ]
   [ Undo { On | Off } ]
   [ ReadOnly ]
   [ Seamless { On | Off } [ Preserve ] ]
```

```
[ UserMap { On | Off } ]
[ UserBrowse { On | Off } ]
[ UserClose { On | Off } ]
[ UserEdit { On | Off } ]
[ UserRemoveMap { On | Off }} ]
[ UserDisplayMap { On | Off } ]
```

table is a string representing the name of the table to be set.

Description

The **Set Table** statement controls settings that affect how and whether a table can be edited. You can use **Set Table** to flag a table as read-only (so that the user will not be allowed to make changes to the table). You can also use **Set Table** to activate or de-activate special editing modes which disable safety mechanisms for the sake of improving editing performance.

Setting FastEdit Mode

Ordinarily, whenever a table is edited (either by the user or by a MapBasic application), MapInfo Professional does not immediately write the edit to the affected table. Instead, MapInfo Professional stores information about the edit to a temporary file known as a transaction file. By writing to a transaction file instead of writing directly to a table, MapInfo Professional gives the user the opportunity to later discard the edits (for example, by choosing **File > Revert**).

If you use the **Set Table** statement to set **FastEdit** mode to **On**, MapInfo Professional writes edit information directly to the table, instead of performing the intermediate step of writing the edit information to a transaction file. Turning on **FastEdit** mode can make subsequent editing operations substantially faster.

While **FastEdit** mode is on, table edits take effect immediately, even if you do not issue a **Commit Table statement**. Use **FastEdit** mode with caution; there is no opportunity to discard edits by choosing **File > Close** or **File > Revert**.

You can only turn **FastEdit** mode on for normal, base tables; you cannot turn on **FastEdit** for a temporary, query table such as Query1. You cannot turn on **FastEdit** mode for a table that already has unsaved changes. You cannot turn on **FastEdit** mode for a linked table.

CAUTION: **While a table is open in FastEdit mode, other network users cannot open that table. After you have completed all edits to be made in FastEdit mode, issue a Commit Table statement or a Rollback statement to reset the file so that other network users can access it.**

If you include the optional **ReadOnly** clause, the table is set to read-only, so that the user cannot edit the table for the remainder of the MapInfo Professional session. The **Set Table** statement does not allow you to turn read-only mode off. You can also activate read-only mode by adding the **ReadOnly** keyword to the **Open Table statement**.

Ordinarily, whenever an edit is made, MapInfo Professional stores information about the edit in memory, so that the user has the option of choosing **Edit > Undo**. If you use the **Set Table** statement to set **Undo** mode to **Off**, MapInfo Professional does not save undo information for each edit; this can make subsequent editing operations substantially faster.

Managing Seamless Tables

A seamless table defines a list of other tables that you can treat as a group. See the MapInfo Professional documentation for an introduction to seamless tables.

The **Seamless** clause enables or disables the seamless behavior for a table. Specify **Seamless Off** to disable seamless behavior, so that you can access the individual rows that define a seamless table. Specify **Seamless On** to restore seamless behavior. If you include the **Preserve** keyword, the effect is permanent; MapInfo Professional writes a change to the table. If you omit the **Preserve** keyword, the effect is temporary, only lasting for the remainder of the session.

The **User…** clauses allow you to limit the actions that the user can perform on a table. These clauses are useful if you want to prevent the user from accidentally opening, closing, or changing tables or windows.

These clauses limit the user-interface only; in other words, **UserMap Off** prevents the user from opening the table in a Map window, but does not prevent a MapBasic program from doing so.

You cannot use these clauses on Cosmetic layers.

Example	Effect
UserMap Off	Table will not appear in the New Map Window or Add Layer dialog boxes.
UserBrowse Off	Table will not appear in the New Browser Window dialog box.
UserClose Off	Table will not appear in the Close Table dialog box.
UserEdit Off	Table will not be editable through the user interface: Browser and Info windows are not editable, and the map layer cannot be made editable.
UserRemoveMap Off	If this table appears in a Map window, the **Remove Layers** button (in the Layer Control window) is disabled for this table.
UserDisplayMap Off	If this table appears in a Map window, the **Visible On/Off** check box (in the Layer Control window) is disabled for this table.

Example

The following statement prevents the World table from appearing in the Close Table dialog box.

```
Set Table World UserClose Off
```

See Also:

TableInfo( ) function

Set Target statement

Purpose

Sets or clears the map editing target object(s). You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Set Target { On | Off }
```

Description

Use the **Set Target** statement to set or clear the editing target object(s); this corresponds to choosing MapInfo Professional's **Objects > Set Target** and **Objects > Clear Target** menu items. Some of MapInfo Professional's advanced editing operations require that an editing target be designated; for example, you must designate an editing target before calling the **Objects Split statement**. For an introduction to using the editing target, see the MapInfo Professional documentation.

Using the **Set Target On** statement corresponds to choosing **Objects > Set Target**. The current set of selected objects becomes the editing target (or an error is generated if no objects are selected).

Using the **Set Target Off** statement corresponds to choosing **Objects > Clear Target**.

See Also:

Objects Combine statement, **Objects Erase statement**, **Objects Intersect statement**, **Objects Overlay statement**, **Objects Split statement**

Set Window statement

Purpose

Changes the size, position, title, or status of a window, and controls the printer, paper size, and margins used by MapInfo Professional. This statement has been updated to accommodate the anti-aliasing choices for vector, text, and image objects. The new code is in bold. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

```
Set Window window_id
   [ Position ( x, y ) [ Units paper_units ] ]
   [ Width win_width [ Units paper_units ] ]
   [ Height win_height [ Units paper_units ] ]
   [ Font... ]
   [ Enhanced { On | Off } ]
   [ Smooth [ Vector { None | Antialias} ] [ Text { None | Antialias} ]
      [ Image { None | Low | High } ] ]
   [ Min | Max | Restore ]
   [ Front ]
```

```
[ Title { new_title | Default } ]
[ Help [ { File help_file | File Default | Off } [ Permanent ] ]
   [ Contents ] [ ID context_ID ] [ { Show | Hide } ]
[ Printer { Default | Name printer_name }
   [ Orientation { Portrait | Landscape } ]
   [ Copies number ]
   [ Papersize number ]
   [ Border { On | Off } ]
   [ TrueColor { On | Off } ]
   [ Dither { Halftone | ErrorDiffusion } ]
   [ Method { Device | Emf | PrintOsbm} ]
   [ Transparency
      [ Raster { Device | ROP } ]
      [ Vector { Device | Internal }] ]
      [ Margins
         [ Left d1 ] [ Right d2 ] [ Top d3 ] [ Bottom d4 ]
            Units paper_units ] } ]
   [ Export { Default |
   [ Border { On | Off } ]
   [ TrueColor { On | Off } ]
   [ Dither { Halftone | ErrorDiffusion } ]
   [ Transparency
      [ Raster { Device | ROP } ]
      [ Vector { Device | Internal }] ]
   [ Scale Patterns { On | Off } ]
   [ Antialiasing { On | Off } ]
   [ Threshold threshold_value ]
   [ MaskSize size_value ]
   [ Filter filter_value ]
]
[ ScrollBars { On | Off } ]
[ Autoscroll { On | Off } ]
[ Parent HWND ]
[ ReadOnly | Default Access ]
[ Table table_name Rec record_number ]
[ Show | Hide ]
[ Smart Pan { On | Off } ]
[ SysMenuClose { On | Off } ]
[ Snap [ Mode { On | Off } ] [ Threshold { pixel_tolerance | Default }]
```

window_id is an integer window identifier or a special window name (for example, Help).

x states the desired distance from the left of MapInfo Professional's workspace to the left edge of the window.

y states the desired distance from the top of MapInfo Professional's workspace to the top edge of the window.

paper_units is a string representing a paper unit name (for example, “cm” for centimeters).

The **Font clause** specifies a text style.

win_width is the desired width of the window.

win_height is the desired height of the window.

new_title is a string expression representing a new title for the window.

help_file is the name of a help file (for example, “FILENAME.HLP” on Windows).

context_ID is an integer help file context ID which identifies a specific help topic.

printer_name identifies a printer. The printer can be local or networked to the computer on which MapInfo Professional is running.

Method determines whether printing will go directly to the device driver or if MapInfo Professional will generate a Windows Enhanced Metafile first and then send the file to the printer or MapInfo will use an Offscreen bitmap to create the output first. EMF method enables the printing of maps with raster images that may not have printed at all in earlier versions, and that use substantially smaller spool files. Offscreen bitmap is invoked depending upon the type of translucent content in the map and enhanced rendering state of the window. However setting OSBM from this window means that printing will use Offscreen bitmaps regardless of the translucency and anti alias settings.

number is the number of copies of a print job that should be sent to the printer.

HWND is an integer window handle. The window specified by *HWND* will become the parent of the window specified by *window_id*; however, only Legend, Statistics, Info, Ruler, and Message windows may be re-parented in this manner.

table_name is the name of an open table to use with the Info window.

record_number is an integer: specify 1 or larger to display a record in the Info window, or specify 0 to display a “No Record” message.

*Enhanced s*ets the version of the rendering technology used to display and print graphics.

The On parameter enables the enhanced rendering technology and is set when the user selects the **Enable Enhanced Rendering** check box in the MapInfo Professional. The Off option disables the enhanced rendering technology and is set when the user does not select the **Enable Enhanced Rendering** check box in MapInfo Professional.

Smooth sets the new rendering technology enhancements for anti-aliasing vector, text and labels, and images.

> The Smooth options (*Vector, Text*, and *Image*) require that the *Enhanced* parameter be set to *On*. MapBasic will throw an error if you turn *Enhanced Off* and set any of the Smooth options to an option other than *None*.

Vector - sets the vector smoothing options for vectors.

None indicates that smoothing is turned off and the vector line and border objects are drawn without anti-aliasing.

Antialias indicates that smoothing is turned on and the vector objects. This option requires that the *Enhanced* parameter be set to *On*.

Text - sets the text smoothing options for n-n-curved labels and the non-curved labels and text objects.

The None parameter indicates that smoothing is turned off for rotated and horizontal labels and text objects.

The Antialias parameter indicates that smoothing is turned on for rotated and horizontal labels and text objects. This option requires that the Enhanced parameter be set to On.

Image - sets the raster image smoothing options.

None indicates that the smoothing is turned off for raster images.

Low indicates that the smoothing is turned on for raster images using a bilinear interpolation method. Using this method, the application displays better quality raster images than *None* but not as good as *High*. Using the *Low* option, the application displays raster images slower than when *None* is used but faster than when *High* is used. This option requires that the *Enhanced* parameter be set to *On*.

High indicates that the smoothing is turned on for raster images using a bicubic interpolation method. Using this method, the application displays better quality raster images than *Low* but results in slower display performance. This option requires that the *Enhanced* parameter be set to *On*.

Printer specifies window-specific overrides for printing.

Export specifies window-specific overrides for exporting.

Default will use the default values found in the output preferences corresponding to printing and/or exporting.

Name *printer_name* specifies the name of the printer to use.

Orientation Portrait prints the document using portrait orientation.

Orientation Landscape prints the document using landscape orientation.

Copies *number* specifies how many copies of the document to print.

Papersize *number* is the paper size information for the window. These numbers are universal for all printers under the Windows operating system. For example, 1 corresponds to Letter size, and 5 corresponds to Legal papersize. This number can be found in the MapBasic file, PAPERSIZE.DEF. Some printer drivers (for example big size plotters) can use their own numbering for identifying paper size. These numbers could be different from numbers that are provided in MapBasic definition

file “PaperSize.def”. Because of this, users with different printer drivers may not identify paper size information stored in a workspace correctly. In that case, paper size will be reset to the printer default value.

Border determines whether an additional black edged rectangle will be drawn around the extents of the window being printed or exported.

Truecolor determines whether to generate 24-bit true color output if it is possible to do so. If **Truecolor** is turned off, the output will be generated using 256 colors.

Dither determines which dithering method to use when it is necessary to convert a 24-bit image to 256 colors. This option is used when outputting raster and grid images. Dithering will occur if **Truecolor** is turned off or if the output device is not capable of supporting 24-bit color.

Method is a keyword that determines whether printing will go directly to the device driver or if MapInfo Professional will generate a Windows Enhanced Metafile first and then send that file to the printer. This method enables the printing of maps with raster images that may not have printed at all in earlier versions, and that use substantially smaller spool files.

Transparency Raster Internal has been removed; however, if present, the keyword will still be parsed without error to allow for compatibility with previous versions.

Transparency Raster determines how transparent pixels should be rendered. Select **Device** or **ROP** dependent upon your printer driver or export file format. You may need to determine your selection after trying each and determining which option produces the best output for you.

Transparency Raster ROP corresponds to the **Use ROP Method to Display Transparent Raster** option in the MapInfo Professional user interface (**Preferences > Output**, **File > Print > Advanced** button, and **File > Save Window As > Advanced** button). If **ROP** is selected, the transparent image is rendered using a raster operation (ROP) to handle the transparent pixels. This method is used to draw transparent (non-translucent) images onscreen; however, it does not always work well when printing. You will need to experiment to determine if your printer driver handles ROP correctly. If you are exporting an image using the **Save Window As** command, this option is beneficial if the output format is a metafile (EMF or WMF). Using the ROP method allows any underlying data to be rendered in the original form.

Transparency Raster Device prevents MapInfo Professional from performing any special handling when printing raster or grid images that contain transparency. The image will be generated using the same method that is used to display the image(s) on screen, but there may be some problems with the output.

Transparency Vector Internal causes MapInfo Professional to perform special handling when outputting transparent fill patterns or transparent bitmap symbols.

Transparency Vector Device prevents MapInfo Professional performing special handling when outputting transparent fill patterns or transparent bitmap symbols. This may cause problems with the output.

Margins User can set printer margins as floating point values in desired units. These values may be increased by the printer driver if the printer margins are smaller than physically possible on a particular printer.

Antialiasing determines whether anti-aliasing filter is used during image exporting. **Antialiasing** is ignored when images are exported to EMF or WMF formats, and when exporting the contents of a Browser window.

Threshold specifies a value that indicates which pixels to smooth. The application of the ant-aliasing filter on the image associates a value with each pixel. Only pixels with values above *threshold_value* are smoothed. If *threshold_value* is set to zero, than all pixels are smoothed. *threshold_value* should be in the range from 0 to 255.

MaskSize specifies a value that indicates the size of the anti-aliasing mask. For example, a value of three indicates an anti-aliasing mask of 3x3. If user sets mask *size_value* too high, then the resulting image can become too blurry.

Filter specifies which anti-aliasing filter to apply. Currently MapInfo Professional supports 6 different filters as listed in the table below.

Filter	ID	Description
FILTER_VERTICALLY_AND_HORIZONTALLY	0	Anti-alias image vertically and horizontally.
FILTER_ALL_DIRECTIONS_1	1	Anti-alias image in all directions
FILTER_ALL_DIRECTIONS_2	2	Anti-alias image in all directions. The filter used for this option is different than FILTER_ALL_DIRECTIONS_1 and gets better results for anti-aliasing text.
FILTER_DIAGONALLY	3	Anti-alias image diagonally.
FILTER_HORIZONTALLY	4	Anti-alias image horizontally
FILTER_VERTICALLY	5	Anti-alias image vertically

Description

The **Set Window** statement customizes an open window, setting such options as the window's size, position, status, font, or title.

The *window_id* parameter can be an integer window identifier, which you can obtain by calling the **FrontWindow( ) function** and the **WindowInfo( ) function**. Alternately, when you use the **Set Window** statement to affect a special MapInfo Professional window, such as the Statistics window, you can identify the window by its name (for example, Statistics) or by its code (for example, WIN_STATISTICS); codes are defined in MAPBASIC.DEF.

The table below lists the window names and window codes which you can use as the *window_id* parameter.

Window name	Window description
MapInfo	The frame window of the entire MapInfo Professional application. You can also refer to this window by its define: WIN_MAPINFO. (The MapInfo application window cannot be renamed).
MapBasic	The MapBasic window. You can also refer to this window by the Define code: WIN_MAPBASIC.
Help	The Help window. You can also refer to this window by the Define code: WIN_HELP.
Statistics	The Statistics window. You can also refer to this window by the Define code: WIN_STATISTICS.
Legend	The Theme Legend window. You can also refer to this window by the Define code: WIN_LEGEND.
Info	The Info Tool window (which appears when the user uses the Info tool). You also can refer to this window by the Define code: WIN_INFO.
Ruler	The window displayed when the user uses the Ruler tool. You can also refer to this window by the Define code: WIN_RULER.
Message	The Message window (which appears when you issue a **Print statement**). You can also refer to this window by the Define code: WIN_MESSAGE.

The optional **Position** clause controls the window's position in the MapInfo Professional workspace. The upper left corner of the workspace has the position 0, 0. The optional **Width** and **Height** clauses control the window's size. Window position and size values use paper units settings, such as “in” (inches) or “cm” (centimeters). MapBasic has a current paper units setting, which defaults to inches; a MapBasic program can change this setting through the **Set Paper Units statement**. A **Set Window** statement can override the current paper units by including the optional **Units** subclause within the **Position**, **Width**, and/or **Height** clauses.

If the statement includes the optional **Max** keyword, the window will be maximized (it will occupy all of MapInfo Professional's work space). If the statement includes the optional **Min** keyword, the window will be minimized (it will be reduced, appearing only as a small icon in the lower part of the screen). If a window is already minimized or maximized, and if the statement includes the optional **Restore** keyword, the window is restored to its previous size.

If the statement includes the optional **Front** keyword, MapBasic makes the window the active window; this is also known as setting the focus on the window. The window comes to the front, as if the user had clicked on the window's title bar.

The statement may always specify a **Position** clause or a **Front** clause, regardless of the type of window specified. However, some of the clauses in the **Set Window** statement apply only to certain types of windows. For example, the Ruler Tool window may not be re-sized, maximized or minimized.

To change the window's title, include the optional **Title** clause. The Application window title (the main "MapInfo" title bar) cannot be changed unless the user is running a runtime version of MapInfo Professional.

The **SysMenuClose** clause lets you disable the Close command in the window's system menu (the menu that appears when a user clicks the box in the upper-left corner of a window). Disabling the Close command only affects the user interface; MapBasic programs can still close the window by issuing **Close Window statement**. The following example disables the Close command of the active window:

```
Set Window FrontWindow( ) SysMenuClose Off
```

Before version 10.5, you could enable or disable the Close button regardless of the toolbar's floating or docking state. As of version 10.5, you cannot enable or disable the Close button when the toolbar is docked. You can only change the state when it is floating or floating and hidden.

Help Window Syntax

To control the online Help window, specify the **Help** keyword instead of the integer *window_id* argument. For example, the following statement displays topic 23 from a custom help file:

```
Set Window Help File "custom.hlp" ID 23
```

The **File** *help_file* clause sets which help file is active. On Windows, this action automatically displays the Help window (unless you also include the **Hide** keyword). Specifying **File Default** resets MapInfo Professional to use the standard MapInfo Professional help, but does not display the help file. MapInfo Professional has only one help file setting, which applies to all MapBasic applications that are running. If one application sets the current help file, other applications may be affected.

The **Off** clause turns off MapInfo Professional's help, so that pressing F1 on an MapInfo Professional dialog has no effect. Use the **Off** clause if you are integrating MapInfo Professional functionality into another application (for example, a Visual Basic program), if you want to prevent the user from seeing MapInfo Professional help. (MapInfo Professional help contains references to MapInfo Professional's menu names, which may not be available in your Visual Basic program.)

The **Permanent** clause sets MapInfo Professional to always use the help file specified by *help_file*, even when the user presses F1 on an MapInfo Professional dialog box. (On Windows, if you omit the **Permanent** keyword, MapInfo Professional resets the help system to use MAPINFOW.HLP whenever the user presses F1 on an MapInfo Professional dialog box.) The **Permanent** setting lasts for the remainder of the MapInfo Professional session, or until you specify a **Set Window Help File…** statement.

To control which help topic appears in the help window, include the **Contents** keyword (to display the Contents screen) or the **ID** clause (to display a specific topic).

MapBasic does not include a help compiler. For more information on working with online help, see the *MapBasic User Guide*.

What to do if MapInfo Professional Resets the Help System to use MapInfoW.hlp

The MapInfo Professional Help System is called MapInfoW.chm, but your Set Window Statement for MapBasic Help may be referencing an older file called MapInfoW.hlp. You may need to change the reference to Mapinfow.hlp to Mapinfow.chm.

On Windows, if you omit the Permanent keyword, MapInfo Professional resets the help system to use MapInfoW.hlp whenever the user presses F1 on an MapInfo Professional dialog box. The Permanent setting lasts for the remainder of the MapInfo Professional session, or until you specify a Set Window Help File statement.

Map or Layout Window Syntax

The **ScrollBars** clause only applies to Map windows. Use the **ScrollBars** clause to show or hide scroll-bars on a Map window.

The **Autoscroll** clause applies to Map and Layout windows. By default, the autoscroll feature is on for every Map and Layout window. In other words, users can scroll a Map or Layout by selecting a draggable tool (such as the Zoom In tool), clicking and dragging to the edge of the window. To prevent users from autoscrolling, specify **Autoscroll Off**. To determine whether a window has autoscroll turned on, call the **WindowInfo( ) function**.

Smart Pan changes the status of the window's panning. When **Smart Pan** is turned on for a Map window or a Layout window, panning and scrolling use off-screen bitmaps to reduce the number of white flashes. The default for **Smart Pan** is off.

When **Smart Pan** is activated for a Layout window, redraw is only affected when the Grabber tool is used.

When **Smart Pan** is activated for a Map window, there will be different effects depending on the method of moving the map. The Grabber tool automatically paints the exposed area as you grab and move the map. The map will move more slowly than when Smart Pan is off. A more complex map will move more slowly. Scrollbars and autoscrolling perform similarly to the Grabber tool, but the speed of the scrolling is not affected by smart panning. When the MapBasic command **Set Map** is used to center or pan with **Smart Redraw** on, the Map window changes without white flashes unless the map is repositioned in such a way that a complete redraw is required.

If off-screen bitmaps have been turned off, then **Smart Pan** in a Map window behaves like a Layout window.

Floating Window (Legend, Ruler, etc.) Syntax

The **Parent** clause allows you to specify a new parent window for a Legend, Statistics, Info, Ruler, or Message window; this clause is only supported on Windows. The window specified by *window_id* becomes a popup window, attached to the window specified by *HWND*.

Re-parenting a window in this manner changes the window's integer ID value. To return a window to its original parent (MapInfo Professional), specify zero as the *HWND*.

The **ReadOnly** / **Default Access** clause applies to the Info, Browser, and Legend windows. This clause controls whether the window is read-only. If you specify **ReadOnly**, the window does not allow editing. If you specify **Default Access**, the window reflects the read/write state of the table it's displaying. This works for the main legend and cartographic legends created with the **Create Legend statement** or the **Create Cartographic Legend statement**.

The **Table** clause allows you to display a specific row in the Info window; this clause is only valid when *window_id* refers to the Info window. Using the **Table** clause displays the Info window, if it was not already visible.

The **Show** or **Hide** clause allows you to show or hide any window that supports show/hide operations (for example, the Ruler window). It can also be used in the MapInfo Professional application window.

Controlling the Printer

By default, windows are printed using the global printer device. This is initialized to the default Windows printer or the MapInfo Professional preferred printer, depending on how the user has set preferences. Using the **Name** clause an application, workspace, or the MapBasic window can override the printer preferences for an individual document. Several settings for the printer can also be controlled by using additional command clauses. Also, when the printer settings are changed through the user interface, appropriate MapBasic commands are generated internally. These overrides are saved with the workspace commands for the affected windows, so they will be reapplied when the workspace is reopened. An override can be removed from a window by running a **Set Window Printer Default** command.

If **Scale Patterns** is set to **On**, fill patterns are scaled based on the ratio of the output device's resolution to the screen resolution.

Attribute codes, WIN_INFO_PRINTER_NAME (21), WIN_INFO_PRINTER_ORIENT (22) or WIN_INFO_PRINTER_COPIES (23), are also returned with **WindowInfo( ) function**.

Example

```
Set Window frontwindow( )
   Printer Name "\\Discovery\HP 2500CP"
      Orientation Portrait
      Copies 10
```

To find out the window's printer name, start MapInfo Professional, go to **File > Page Setup**. Click the **Printer** button. Use the printer name found in that dialog box.

Controlling Snap Tolerance

You can set snap to a particular pixel tolerance for a given window, set snap back to the default snap tolerance for a given window, or retrieve the current snap tolerance for a given window. You can also turn snap on/off for a given window, or retrieve information about whether snap is on/off for a window.

Snap mode settings for a particular window can be queried using new attribute parameters in the **WindowInfo( ) function**. Snap mode and tolerance can be set for each Map and Layout window. These settings are saved in the workspace for each window.

Example

```
Dim win_id As Integer
Open Table "world"
Map From world
win_id = FrontWindow( )
Set Window win_id Width 5 Height 3
```

Saving a .WOR that Can Be Opened in Localized/Unlocalized Versions

Before MapInfo Professional/MapBasic version 9.0.2, if you created a workspace file containing a layout and then sent it to another MapInfo Professional user working in a different locale, the workspace would error when the user tried to open it. This occurred because the map name would change due to the change in language.

We have created a registry entry workaround to prevent this error and allow users in different locales to open the workspaces without error. You must enter this registry entry manually.

To prevent map name errors due to the change in locale:

1. From the command line, type **regedit**. The Registry Editor window displays.
2. Go to My Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Mapinfo\Mapinfo\Common.
3. Right-click and select **New > DWORD value** to create a new DWORD registry entry.
4. Rename the entry **WriteWindowTitle** and press **Enter**. The Edit DWORD value dialog box displays.
5. Type **1** in the Value data field and click **OK** to save your entry.
6. Close MapInfo Professional and reopen it.

This corrects the problem by writing the name of the table explicitly in the Layout. This prevents the name change when the file changes locales.

See Also:

Browse statement, **Graph statement**, **Layout statement**, **Map statement**, **Set Paper Units statement**

Sgn( ) function

Purpose

Returns -1, 0, or 1, to indicate that a specified number is negative, zero, or positive (respectively). You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

Sgn (*num_expr*)

num_expr is a numeric expression.

Return Value

Float (-1, 0, or 1)

Description

The **Sgn( )** function returns a value of -1 if the *num_expr* is less than zero, a value of 0 (zero) if *num_expr* is equal to zero, or a value of 1 (one) if *num_expr* is greater than zero.

Example

```
Dim x As Integer
x = Sgn(-0.5)

' x now has a value of -1
```

See Also:

Abs( ) function

Shade statement

Purpose

Creates a thematic map layer and adds it to an existing Map window. You can issue this statement in the MapBasic window in MapInfo Professional.

Syntax

See the following sections:

- **Shading by Ranges of Values**
- **Shading by Individual Values**
- **Dot Density**
- **Graduated Symbols**
- **Pie Charts**
- **Bar Charts**

Description

The **Shade** statement creates a thematic map layer and adds the layer to an existing Map window. The **Shade** statement corresponds to MapInfo Professional's **Map > Create Thematic Map** menu item. For an introduction to thematic mapping and the **Create Thematic Map** menu item, see the MapInfo Professional documentation.

Between sessions, MapInfo Professional preserves thematic settings by storing a **Shade** statement in the workspace file. Thus, to see an example of the **Shade** statement, you could create a Map, choose the **Map > Create Thematic Map** command, save the workspace (for example, THEME.WOR), and examine the workspace in a MapBasic text edit window. You could then copy the **Shade** statement in your MapBasic program. Similarly, you can see examples of the **Shade** statement by opening MapInfo Professional's MapBasic Window before you choose **Map > Create Thematic Map**.

Shading by Ranges of Values

Syntax

```
Shade [ Window window_id ]
   { layer_id | layer_name }
   With Metadata
   With expr
   [ Ignore value_to_ignore ]
   Ranges
   [ Apply { Color | Size | All } ]
   [ Use { Color | Size | All } [ Line... ] [ Brush... ]
      [ Symbol... ]
   ]
   { [ From Variable float_array Style Variable style_array ] |
      minimum : maximum [ Pen... ] [ Line...] [ Brush... ]
```

```
    [ Symbol... ] [ , minimum : maximum [ Pen... ]
    [ Line...] [ Brush... ] [ Symbol... ] ... ]
}
[ Style Replace { On | Off } ]
[ Default [ Pen... ] [ Line... ] [ Brush... ] [ Symbol... ] ]
```

window_id is the integer window identifier of a Map window.

layer_id is the layer identifier of a layer in the Map (one or larger).

layer_name is the name of a layer in the Map.

expr is the expression by which the table will be shaded, such as a column name.

value_to_ignore is a value to be ignored; this is usually zero (when using numerical expressions) or a blank string (when using string expressions); no thematic object will be created for a row if the row's value matches the value to be ignored.

float_array is an array of float values initialized by a **Create Ranges statement**.

style_array is an array of **Pen**, **Brush** or **Symbol** values initialized by a **Create Styles statement**.

minimum is the minimum numeric value for a range.

maximum is the maximum numeric value for a range.

Description

The optional *window_id* clause identifies which Map is to be shaded; if no *window_id* is provided, MapBasic shades the topmost Map window.

The **Shade** statement must specify which layer to shade thematically, even if the Map window has only one layer. The layer may be identified by number (*layer_id*), where the topmost map layer has a *layer_id* value of one, the next layer has a *layer_id* value of two, etc. Alternately, the **Shade** statement can identify the map layer by name (for example, “world”).

Each **Shade** statement must specify an *expr* expression clause. MapInfo Professional evaluates this expression for each object in the table being shaded; following the **Shade** statement, MapInfo Professional chooses each object's display style based on that record's *expr* value. The expression typically includes the names of one or more columns from the table being shaded.

If you specify **With Metadata**, any theme metadata contained in the open table is used to create the Individual or Ranged Value theme.

The keywords following the *expr* clause dictate which type of shading MapInfo Professional will perform. The **Ranges** keyword results in a shaded map where each object falls into a range of values.

The **Pen clause** specifies a line style (for example, **MakePen**(*width*, *pattern*, *color*)) to use for the borders of filled objects (for example, regions).

The **Line** clause specifies a line style to use for lines, polylines, and arcs. The syntax of the **Line** clause is identical to the **Pen clause**, except for the keyword **Line** appearing in place of **Pen**.

The **Brush clause** specifies a fill style (for example, **MakeBrush**(*pattern*, *forecolor*, *backcolor*)).

The **Symbol clause** specifies a symbol style (for example, **MakeSymbol**(*shape*, *color*, *size*)).

For the specific syntax of a Ranges map, see **Syntax Shading by Ranges of Values**.

In a Ranges map, you can use the **From Variable** and **Style Variable** clauses to read pre-calculated sets of range information from array variables. The array variables must have been initialized using the **Create Ranges statement** and the **Create Styles statement**. For an example of using arrays in **Shade** statements, see **Create Ranges statement**.

If you specify either the **Ranges** or **Values** keyword, the statement can include the optional **Default** clause. This clause lets you specify the graphic styles used by the “all others” range. If a row does not fall into any of the specified ranges, MapInfo Professional assigns the row to the all-others range. If the **Shade** statement does not read range settings from array variables, then the **Ranges** keyword is followed by from one to sixteen explicit range descriptions. Each range description consists of a pair of numeric values (separated by a colon), followed by the graphic styles that MapInfo Professional should use to display objects belonging to that range. If a record's *expr* value is greater than or equal to the minimum value, and less than the maximum value, then that record belongs to that range. The range descriptions are separated by commas.

```
Open Table "states"
Map From states
Shade states With Pop_1990 Ranges
   4827000:29280000 Brush (2,0,201326591) ,
   1783000: 4827000 Brush (8,0,16777215) ,
   449000: 1783000 Brush (5,0,16777215)
```

If you are shading regions, specify **Brush clause**s to control the region fill styles. If you are shading points, specify **Symbol clause**s. If you are shading linear objects (lines, polylines, or arcs) specify **Line** clauses, not **Pen clause**s; the syntax is identical, except that you substitute the keyword **Line** instead of the keyword **Pen**. (In a **Shade** statement, the Pen clause controls the style for the borders of filled objects, such as regions.)

Style Replace On (default) specifies the layers under the theme are not drawn.

Style Replace Off specifies the layers under the theme are drawn, allowing for multi-variate transparent themes.

Style Replace On is the default and provides backwards compatibility with the existing behavior so that the underlying layers are not drawn.

You can use the **Apply** clause to control which display attributes MapInfo Professional applies to the shaded objects.

Apply clause	Effect
Apply Color	The shading only changes the colors of objects in the map. Point objects appear in their original shape and size, but the thematic shading controls the point colors. Line objects appear in their original pattern and thickness, but the thematic shading controls the line colors. Filled objects appear in their original fill pattern, but the thematic shading controls the foreground color.
Apply Size	The shading only changes the sizes of point objects and the thickness of linear objects. Point objects appear in their original shape and color, but the thematic shading controls the symbol sizes. Line objects appear in their original pattern and color, but the shading controls the line thickness.
Apply All	The shading controls all display attributes: symbol shape, symbol size, line pattern, line thickness, and color.

If you omit the **Apply** clause, **Apply All** is the default.

The **Use** clause lets you control whether MapInfo Professional applies all of the style elements from the range styles, or only some of the style elements. This is best illustrated by example. The following example shades the table WorldCap, which contains points. This example does not include a **Use** clause.

```
Shade WorldCap With Cap_Pop Ranges
   Apply All
   0 : 300000 Symbol(35,YELLOW,9) ,
   300000 : 900000 Symbol(35,GREEN,18) ,
   900000 : 20000000 Symbol(35,BLUE,27)
```

In this thematic map, each range appears exactly as its **Symbol clause** dictates: Points in the low range appear as 9-point, yellow stars (code 35 is a star shape); points in the medium range appear as 18-point, green stars; points in the high range appear as 27-point, blue stars.

The following example shows the same statement with the addition of a **Use Size** clause.

```
Shade WorldCap With Cap_Pop Ranges
   Apply All

   Use Size Symbol(34, RED, 24) ' <<<<< Note!

   0 : 300000 Symbol(35,YELLOW,9) ,
   300000 : 900000 Symbol(35,GREEN,18) ,
   900000 : 20000000 Symbol(35,BLUE,27)
```

ⓘ The **Use Size** clause provides its own Symbol style: Shape 34 (circle), in red.

Because of the **Use Size** clause, MapInfo Professional uses only the size values from the latter **Symbol clause**s (9, 18, 27 point); MapInfo Professional ignores the other display attributes (for example, YELLOW, GREEN, BLUE). The thematic map shows red circles, because the **Use Size**

Symbol clause specifies red circles. The end result: Points in the low range appear as 9-point, red circles; points in the medium range appear as 18-point, red circles; points in the high range appear as 27-point, red circles.

If you specify **Use Color** instead of **Use Size**, MapInfo Professional uses only the colors from the latter **Symbol clause**s. The map will show yellow, green, and blue circles, all at 24-point size.

Specifying **Use All** has the same effect as leaving out the **Use** clause.

The **Use** clause is only valid if you specify **Apply All** (or if you omit the **Apply** clause entirely).

Shading by Individual Values

Syntax

```
Shade [ Window window_id ]
   { layer_id | layer_name }
   With Metadata
   With expr
   [ Ignore value_to_ignore ]
   Values const [ Pen... ] [ Line...] [ Brush... ] [ Symbol... ]
      [ , const [ Pen... ] [ Line...] [ Brush... ] [ Symbol... ] ... ]
      [ Vary { Color | All } ]
   [ Style Replace { On | Off } ]
   [ Default [ Pen... ] [ Brush... ] [ Symbol... ] ]
```

window_id is the integer window identifier of a Map window.

layer_id is the layer identifier of a layer in the Map (one or larger).

layer_name is the name of a layer in the Map.

expr is the expression by which the table will be shaded, such as a column name.

value_to_ignore is a value to be ignored; this is usually zero (when using numerical expressions) or a blank string (when using string expressions); no thematic object will be created for a row if the row's value matches the value to be ignored.

const is a constant numeric expression or a constant string expression.

Description

The optional *window_id* clause identifies which Map is to be shaded; if no *window_id* is provided, MapBasic shades the topmost Map window.

The **Shade** statement must specify which layer to shade thematically, even if the Map window has only one layer. The layer may be identified by number (*layer_id*), where the topmost map layer has a *layer_id* value of one, the next layer has a *layer_id* value of two, etc. Alternately, the **Shade** statement can identify the map layer by name (for example, “world”).

Each **Shade** statement must specify an *expr* expression clause. MapInfo Professional evaluates this expression for each object in the table being shaded; following the **Shade** statement, MapInfo Professional chooses each object's display style based on that record's *expr* value. The expression typically includes the names of one or more columns from the table being shaded.

If you specify **With Metadata**, any theme metadata contained in the open table is used to create the Individual or Ranged Value theme.

The keywords following the *expr* clause dictate which type of shading MapInfo Professional will perform. The **Values** keyword creates a map where each unique value has its own display style.

The **Pen clause** specifies a line style (for example, **MakePen**(*width*, *pattern*, *color*)) to use for the borders of filled objects (for example, regions).

The **Line** clause specifies a line style to use for lines, polylines, and arcs. The syntax of the **Line** clause is identical to the **Pen clause**, except for the keyword **Line** appearing in place of **Pen**.

The **Brush clause** specifies a fill style (for example, **MakeBrush**(*pattern*, *forecolor*, *backcolor*)).

The **Symbol clause** specifies a symbol style (for example, **MakeSymbol**(*shape*, *color*, *size*)).

For the specific syntax of an Individual Values map, see Syntax **Shading by Individual Values**.

In a Values map, the keyword **Values** is followed by from one to 255 value descriptions. Each value description consists of a unique value (string or numeric), followed by the graphic styles that MapInfo Professional should use to display objects having that exact value. If a record's *expr* value is exactly equal to one of the **Shade** statement's value descriptions, then that record's object will be displayed with the appropriate graphic style. The value descriptions are separated by commas.

If the **Shade** statement specifies either the **Ranges** or **Values** keyword, the statement can include the optional **Default** clause. This clause lets you specify the graphic styles used by the “all others” range. If a row does not fall into any of the specified ranges, MapInfo Professional assigns the row to the all-others range. The **Vary** clause sets how the objects will vary in appearance. The default is **Vary All**. If **Vary** All is specified, all of the display tools for each range are applied in the theme. If **Vary Color** is specified, only the color for the specified range is applied.

Style Replace On (default) specifies the layers under the theme are not drawn.

Style Replace Off specifies the layers under the theme are drawn, allowing for multi-variate transparent themes. This enables transparent patterns to be displayed on the same layer.

Style Replace On is the default and provides backwards compatibility with the existing behavior so that the underlying layers are not drawn.

The following example assumes that the UK_Sales table has a column called Sales_Rep; this column contains the name of the sales representative who handles the accounts for a sales territory in the United Kingdom. The **Shade** statement will display each region in a shade which depends upon that region's salesperson. Thus, all regions assigned to Bob will appear in one color, while all regions assigned to Jan will appear in another color, etc.

```
Open Table "uk_sales"
Map From uk_sales

Shade 1 With Proper$(Sales_Rep)
   Ignore ""
   Values
      "Alan" ,
      "Amanda" ,
      "Bob" ,
      "Jan"
```

Dot Density

Syntax

```
Shade [ Window window_id ]
   { layer_id | layer_name }
   With expr
   Density dot_value { Circle | Square }
   Width dot_size
   [ Color color ]
```

window_id is the integer window identifier of a Map window.

layer_id is the layer identifier of a layer in the Map (one or larger).

layer_name is the name of a layer in the Map.

expr is the expression by which the table will be shaded, such as a column name.

dot_value is the numeric value associated with each dot in a dot density map.

dot_size is the size, in pixels, of each dot on a dot density map.

color is the RGB value for the color of the dots in a dot density map.

Description

The optional *window_id* clause identifies which Map is to be shaded; if no *window_id* is provided, MapBasic shades the topmost Map window.

The **Shade** statement must specify which layer to shade thematically, even if the Map window has only one layer. The layer may be identified by number (*layer_id*), where the topmost map layer has a *layer_id* value of one, the next layer has a *layer_id* value of two, etc. Alternately, the **Shade** statement can identify the map layer by name (for example, “world”).

Each **Shade** statement must specify an *expr* expression clause. MapInfo Professional evaluates this expression for each object in the table being shaded; following the **Shade** statement, MapInfo Professional chooses each object's display style based on that record's *expr* value. The expression typically includes the names of one or more columns from the table being shaded.

The keywords following the *expr* clause dictate which type of shading MapInfo Professional will perform. The **Density** keyword creates a dot density map.

For the specific syntax of a Dot Density map, see **Syntax Dot Density**.

In a Density map, the keyword **Density** is followed by a *dot_value* clause. You can specify either a Circle or Square thematic style. Note that a map layer must include regions in order to provide the basis for a meaningful dot density map; this is because the number of dots displayed in each region represent some sort of density value for that region. For example, each dot might represent one thousand households.

In a dot density map, a numeric *expr* value is calculated for each region; the *dot_value* represents a numeric value as well. MapInfo Professional decides how many dots to draw in a given region by dividing that region's *expr* value by the map's *dot_value* setting. Thus, if a region has an *expr* value of 100, and the **Shade** statement specifies a *dot_value* of 5, then MapInfo Professional draws 20 dots in that region, because each dot represents a quantity of 5.

The keyword **Width** is followed by *dot_size*. This specifies how large the dots should be, in terms of pixels. For Circle dot style, the *dot_size* can be 2 to 25 pixels in width. For Square dot style, the *dot_size* can be 1 to 25 pixels. The optional **Color** clause is used to set the color of the dots.

The following example creates a dot density map using the States table's Pop_1990 column, (which in this case indicates the number of households per state, circa 1990). The resultant dot density map will show many 4-pixel dots; each dot representing 60,000 households.

```
Open Table "states"
Map From states
shade window 176942288 7
with Pop_1990
density 600000 circle width 4
color 255
```

For backwards compatibility, the older MapBasic syntax (version 7.5 or earlier) is still supported.

Graduated Symbols

Syntax

```
Shade [ Window window_id ]
   { layer_id | layer_name }
   With expr
   Graduated min_value : symbol_size max_value : symbol_size
      Symbol...
      [ Inflect Symbol... ]
      [ Vary Size By { "LOG" | "SQRT" | "CONST" } ]
```

window_id is the integer window identifier of a Map window.

layer_id is the layer identifier of a layer in the Map (one or larger).

layer_name is the name of a layer in the Map.

expr is the expression by which the table will be shaded, such as a column name.

max_value is a number,

min_value is a number,

symbol_size is the point size to use for symbols having the appropriate value.

Description

The optional *window_id* clause identifies which Map is to be shaded; if no *window_id* is provided, MapBasic shades the topmost Map window.

The **Shade** statement must specify which layer to shade thematically, even if the Map window has only one layer. The layer may be identified by number (*layer_id*), where the topmost map layer has a *layer_id* value of one, the next layer has a *layer_id* value of two, etc. Alternately, the **Shade** statement can identify the map layer by name (for example, “world”).

Each **Shade** statement must specify an *expr* expression clause. MapInfo Professional evaluates this expression for each object in the table being shaded; following the **Shade** statement, MapInfo Professional chooses each object's display style based on that record's *expr* value. The expression typically includes the names of one or more columns from the table being shaded.

The keywords following the *expr* clause dictate which type of shading MapInfo Professional will perform. The **Graduated** keyword results in a graduated symbols map.

For the specific syntax of a Graduated map, see **Syntax Graduated Symbols**.

In a Graduated map, the keyword **Graduated** is followed by a pair of *value*:*symbol_size* clauses. The first of the *value*:*symbol_size* clauses specifies what size symbol corresponds to the minimum value, and the second of the *value*:*symbol_size* clauses specifies what size symbol corresponds to the maximum value. MapInfo Professional uses intermediate symbol sizes for rows having values between the extremes.

A **Symbol clause** dictates what type of symbol should appear (circle, star, etc.). If you include the optional **Inflect** clause, which specifies a second Symbol style, MapInfo Professional uses the secondary symbol style to draw symbols for rows having negative values.

The following example creates a graduated symbols map showing profits and losses. Stores showing a profit are represented as green triangles, pointing up. The **Shade** statement also includes an **Inflection** clause, so that stores showing a net loss appear as red triangles, pointing down.

```
Shade stores With Net_Profit
   Graduated
   0.0:0 15000:24
   Symbol(36, GREEN, 24)
   Inflect Symbol(37, RED, 24)
   Vary Size By "SQRT"
```

The optional **Vary Size By** clause controls how differences in numerical values correspond to differences in symbol sizes. If you omit the **Vary Size By** clause, MapInfo Professional varies the symbol size using the “SQRT” (square root) method, which assigns increasingly larger point sizes as the square roots of the values increase. When you vary by square root, each symbol's area is proportionate to the row's value; thus, if one row has a value twice as large as another row, the row with the larger value will have a symbol that occupies twice as much area on the map.

Having twice the area is not the same as having twice the point size. When you double an object's point size, its area quadruples, because you are increasing both height and width.

Pie Charts

Syntax

```
Shade [ Window window_id ]
    { layer_id | layer_name | Selection }
   With expr [ , expr... ]
   [ Half ] Pie [ Angle angle ] [ Counter ]
   [ Fixed ] [ Max Size chart_size [ Units unitname ]
      [ At Value max_value [ Vary Size By {"LOG" | "SQRT" | "CONST" } ]]]
```

```
[ Border Pen... ]
[ Position [ { Left | Right | Center } ] [ { Above | Below | Center }]]
[ Style Brush... [ , Brush... ] ]
```

window_id is the integer window identifier of a Map window.

layer_id is the layer identifier of a layer in the Map (one or larger).

layer_name is the name of a layer in the Map.

expr is the expression by which the table will be shaded, such as a column name.

angle is the starting angle, in degrees, of the first wedge in a pie chart.

chart_size is a float size, representing the maximum height of each pie or bar chart.

unitname is a paper unit name (for example, “in” for inches, “cm” for centimeters).

max_value is a number, used in the **At Value** clause to control the heights of Pie and Bar charts. For each record, if the sum of the column expressions equals the *max_value*, that record's Pie or Bar chart will be drawn at the *chart_size* height; the charts are smaller for rows with smaller sums.

Description

The optional *window_id* clause identifies which Map is to be shaded; if no *window_id* is provided, MapBasic shades the topmost Map window.

The **Shade** statement must specify which layer to shade thematically, even if the Map window has only one layer. The layer may be identified by number (*layer_id*), where the topmost map layer has a *layer_id* value of one, the next layer has a *layer_id* value of two, etc. Alternately, the **Shade** statement can identify the map layer by name (for example, “world”).

Each **Shade** statement must specify an *expr* expression clause. MapInfo Professional evaluates this expression for each object in the table being shaded; following the **Shade** statement, MapInfo Professional chooses each object's display style based on that record's *expr* value. The expression typically includes the names of one or more columns from the table being shaded.

The keywords following the *expr* clause dictate which type of shading MapInfo Professional will perform. The **Pie** keyword specify thematically constructed charts.

The **Pen clause** specifies a line style (for example, **MakePen**(*width*, *pattern*, *color*)) to use for the borders of filled objects (for example, regions).

The **Brush clause** specifies a fill style (for example, **MakeBrush**(*pattern*, *forecolor*, *backcolor*)).

For the specific syntax of a Pie map, see **Syntax Pie Charts**.

In a Pie map, MapInfo Professional creates a small pie chart for each map object to be shaded. The **With** clause specifies a comma-separated list of two or more expressions to comprise each thematic pie.

If you place the optional keyword **Half** before the keyword **Pie**, MapInfo Professional draws half-pies; otherwise, MapInfo Professional draws whole pies.

The optional **Angle** clause specifies the starting angle of the first pie wedge, specified in degrees. The default start angle is 180.

The optional **Counter** keyword specifies that wedges are drawn in counter-clockwise order, starting at the start angle.

The **Max Size** clause controls the sizes of the pie charts, in terms of paper units (for example, “in” for inches). If you include the **Fixed** keyword, all charts are the same size.

For example, the following statement produces pie charts, all of the same size:

```
Shade sales_95 With phone_sales, retail_sales
   Pie Fixed
   Max Size 0.25 Units "in"
```

To vary the sizes of Pie charts, omit the **Fixed** keyword and include the **At Value** clause. For example, the following statement produces a theme where the size of the Pie charts varies. If a record has a sum of 85,000 its Pie chart will be 0.25 inches tall; records having smaller values are shown as smaller Pie charts.

```
Shade sales_95 With phone_sales, retail_sales
   Pie
   Max Size 0.25 Units "in" At Value 85000
```

The optional **Vary Size By** clause controls how MapInfo Professional varies the Pie chart size. This clause is discussed above (see **Graduated Symbols**).

Each chart is placed on the original map object's centroid, unless a **Position** clause is used.

The **Style** clause specifies a comma-separated list of Brush styles; specify one Brush style for each expression specified in the **With** clause. Brush style settings are optional; if you omit these settings, MapInfo Professional uses any Brush preferences saved by the user.

The following example creates a thematic map layer which positions each pie chart directly above each map object's centroid.

```
Shade sales_95 With phone_sales, retail_sales
   Pie Angle 180
   Max Size 0.5 Units "in" At Value 85000
      Vary Size By "SQRT"
   Border Pen (1, 2, 0)
   Position Center Above
   Style Brush(2, RED, 0), Brush(2, BLUE, 0)
```

Bar Charts

Syntax

```
Shade [ Window window_id ]
   { layer_id | layer_name | Selection }
   With expr [ , expr... ]
   { Bar [ Normalized ] | Stacked Bar [ Fixed ] }
   [ Max Size chart_size [ Units unitname ]
      [ At Value max_value [ Vary Size By {"LOG" | "SQRT" | "CONST" } ]]
   [ Border Pen... ]
   [ Frame Brush... ]
   [ Width value [ Units unitname ] ]
```

```
[ Position [ { Left | Right | Center } ] [ { Above | Below | Center }]]
[ Style Brush... [ , Brush... ] ]
```

window_id is the integer window identifier of a Map window.

layer_id is the layer identifier of a layer in the Map (one or larger).

layer_name is the name of a layer in the Map.

expr is the expression by which the table will be shaded, such as a column name.

chart_size is a float size, representing the maximum height of each pie or bar chart.

max_value is a number, used in the **At Value** clause to control the heights of Pie and Bar charts. For each record, if the sum of the column expressions equals the *max_value*, that record's Pie or Bar chart will be drawn at the *chart_size* height; the charts are smaller for rows with smaller sums.

unitname is a paper unit name (for example, “in” for inches, “cm” for centimeters).

value

Description

The optional *window_id* clause identifies which Map is to be shaded; if no *window_id* is provided, MapBasic shades the topmost Map window.

The **Shade** statement must specify which layer to shade thematically, even if the Map window has only one layer. The layer may be identified by number (*layer_id*), where the topmost map layer has a *layer_id* value of one, the next layer has a *layer_id* value of two, etc. Alternately, the **Shade** statement can identify the map layer by name (for example, “world”).

Each **Shade** statement must specify an *expr* expression clause. MapInfo Professional evaluates this expression for each object in the table being shaded; following the **Shade** statement, MapInfo Professional chooses each object's display style based on that record's *expr* value. The expression typically includes the names of one or more columns from the table being shaded.

The keywords following the *expr* clause dictate which type of shading MapInfo Professional will perform. The **Bar** keyword specify thematically constructed charts.

The **Pen clause** specifies a line style (for example, **MakePen**(*width*, *pattern*, *color*)) to use for the borders of filled objects (for example, regions).

The **Brush clause** specifies a fill style (for example, **MakeBrush**(*pattern*, *forecolor*, *backcolor*)).

For the specific syntax of a Bar map, see **Syntax Bar Charts**.

In a Bar map, MapInfo Professional creates a small bar chart for each map object. The **With** clause specifies a comma-separated list of expressions to comprise each thematic chart.

If you place the optional keyword **Stacked** before the keyword **Bar**, MapInfo Professional draws a stacked bar chart; otherwise, MapInfo Professional draws bars side-by-side. If you omit the keyword **Stacked**, you can include the keyword **Normalized** to specify that the bars have independent scales.

When you create a Stacked bar chart map, you can include the optional **Fixed** keyword to specify that all bar charts in the thematic layer should appear in the same size (for example, half an inch tall) regardless of the numeric values for that map object. If you omit the **Fixed** keyword, MapInfo Professional sizes each object's bar chart according to the net sum of the values in the chart.

The **Frame** **Brush clause** specifies a fill style used for the background behind the bars.

The **Position** clause controls both the orientation of the bar charts (horizontal or vertical bars) and the position of the charts relative to object centroids. If the **Position** clause specifies **Left** or **Right**, the bars are horizontal, otherwise the bars are vertical.

The **Style** clause specifies a comma-separated list of Brush styles. Specify one Brush style for each expression specified in the **With** clause.

The following example creates a thematic map layer which positions each bar chart directly above each map object's centroid.

```
Shade sales_93
   With phone_sales, retail_sales
   Bar
   Max Size 0.4 Units "in" At Value 1245000
   Vary Size By "CONST"
   Border Pen (1, 2, 0)
   Position Center Above
   Style Brush(2, RED, 0), Brush(2, BLUE, 0)
```

See Also:

Create Ranges statement, **Create Styles statement**, **Map statement**, **Set Legend statement**, **Set Map statement**, **Set Shade statement**

Sin( ) function

Purpose

Returns the sine of a number. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

Sin(*num_expr*)

num_expr is a numeric expression representing an angle in radians.

Return Value

Float

Description

The **Sin( )** function returns the sine of the numeric *num_expr* value, which represents an angle in radians. The result returned from **Sin( )** will be between one and negative one. To convert a degree value to radians, multiply that value by DEG_2_RAD. To convert a radian value into degrees, multiply that value by RAD_2_DEG. The codes DEG_2_RAD and RAD_2_DEG are defined in MAPBASIC.DEF.

Example

```
Include "mapbasic.def"
Dim x, y As Float
x = 30 * DEG_2_RAD
y = Sin(x)
' y will now be equal to 0.5
' since the sine of 30 degrees is 0.5
```

See Also:

Acos( ) function, **Asin( ) function**, **Atn( ) function**, **Cos( ) function**, **Tan( ) function**

Space$( ) function

Purpose

Returns a string consisting only of spaces. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

Space$(*num_expr*)

num_expr is a SmallInt numeric expression.

Return Value

String

Description

The **Space$( )** function returns a string *num_expr* characters long, consisting entirely of space characters. If the *num_expr* value is less than or equal to zero, the **Space$( )** function returns a null string.

Example

```
Dim filler As String
filler = Space$(7)
' filler is now equal to the string "       "
' (7 spaces)
Note "Hello" + filler + "world!"
'this displays the message "Hello       world!"
```

See Also:

String$( ) function

SphericalArea( ) function

Purpose

Returns the area using as calculated in a Latitude/Longitude non-projected coordinate system using great circle based algorithms. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

SphericalArea(*obj_expr*, *unit_name*)

obj_expr is an object expression.

unit_name is a string representing the name of an area unit (for example, “sq km”).

Return Value

Float

Description

The **SphericalArea( )** function returns the area of the geographical object specified by *obj_expr*. The function returns the area measurement in the units specified by the *unit_name* parameter; for example, to obtain an area in acres, specify “acre” as the *unit_name* parameter. See **Set Area Units statement** for the list of available unit names.

The **SphericalArea( )** function will always return the area as calculated in a Latitude/Longitude non-projected coordinate system using spherical algorithms. A value of -1 will be returned for data that is in a NonEarth coordinate system since this data cannot be converted into a Latitude/longitude coordinate system.

Only regions, ellipses, rectangles, and rounded rectangles have any area. By definition, the **SphericalArea( )** of a point, arc, text, line, or polyline object is zero. The **SphericalArea( )** function returns approximate results when used on rounded rectangles. MapBasic calculates the area of a rounded rectangle as if the object were a conventional rectangle.

Examples

The following example shows how the **SphericalArea( )** function can calculate the area of a single geographic object. Note that the expression *tablename*.obj (as in states.obj) represents the geographical object of the current row in the specified table.

```
Dim f_sq_miles As Float
Open Table "states"
Fetch First From states
f_sq_miles = Area(states.obj, "sq mi")
```

You can also use the **SphericalArea( )** function within the **Select statement**, as shown in the following example.

```
Select state, SphericalArea(obj, "sq km")
   From states Into results
```

See Also:

CartesianArea( ) function, **SphericalArea( ) function**

SphericalConnectObjects( ) function

Purpose

Returns an object representing the shortest or longest distance between two objects. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

SphericalConnectObjects(*object1*, *object2*, *min*)

object1 and *object2* are object expressions.

min is a logical expression where TRUE calculates the minimum distance between the objects, and FALSE calculates the maximum distance between objects.

Return Value

This statement returns a single section, two-point Polyline object representing either the closest distance (`min == TRUE`) or farthest distance (`min == FALSE`) between *object1* and *object2*.

Description

One point of the resulting Polyline object is on *object1* and the other point is on *object2*. Note that the distance between the two input objects can be calculated using the **ObjectLen( ) function**. If there are multiple instances where the minimum or maximum distance exists (e.g., the two points returned are not uniquely the shortest distance and there are other points representing “ties”) then these functions return one of the instances. There is no way to determine if the object returned is uniquely the shortest distance.

SphericalConnectObjects( ) returns a Polyline object connecting *object1* and *object2* in the shortest (`min == TRUE`) or longest (`min == FALSE`) way using a spherical calculation method. If the calculation cannot be done using a spherical distance method (e.g., if the MapBasic coordinate system is NonEarth), then this function will produce an error.

SphericalDistance( ) function

Purpose

Returns the distance between two locations. You can call this function from the MapBasic Window in MapInfo Professional. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

SphericalDistance(*x1*, *y1*, *x2*, *y2*, *unit_name*)

x1 and *x2* are x-coordinates (for example, longitude).

y1 and *y2* are y-coordinates (for example, latitude).

unit_name is a string representing the name of a distance unit (for example,"km").

Return Value

Float

Description

The **SphericalDistance( )** function calculates the distance between two locations.

The function returns the distance measurement in the units specified by the unit_name parameter; for example, to obtain a distance in miles, specify "mi" as the *unit_name* parameter. See **Set Distance Units statement** for the list of available unit names.

The x- and y-coordinate parameters must use MapBasic's current coordinate system. By default, MapInfo Professional expects coordinates to use a Latitude/Longitude coordinate system. You can reset MapBasic's coordinate system through the **Set CoordSys statement**.

The **SphericalDistance( )** function always returns a value as calculated in a Latitude/Longitude non-projected coordinate system using great circle based algorithms. A value of -1 will be returned for data that is in a NonEarth coordinate system since this data cannot be converted into a Latitude/longitude coordinate system.

Example

```
Dim dist, start_x, start_y, end_x, end_y As Float
Open Table "cities"
Fetch First From cities
start_x = CentroidX(cities.obj)
start_y = CentroidY(cities.obj)
Fetch Next From cities
end_x = CentroidX(cities.obj)
end_y = CentroidY(cities.obj)
dist = SphericalDistance(start_x,start_y,end_x,end_y,"mi")
```

See Also:

CartesianDistance( ) function, **Distance( ) function**

SphericalObjectDistance( ) function

Purpose

Returns the distance between two objects. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

SphericalObjectDistance(*object1*, *object2*, *unit_name*)

object1 and *object2* are object expressions.

unit_name is a string representing the name of a distance unit.

Return Value

Float

Description

SphericalObjectDistance( ) returns the minimum distance between *object1* and *object2* using a spherical calculation method with the return value in *unit_name*. If the calculation cannot be done using a spherical distance method (e.g., if the MapBasic coordinate system is NonEarth), then this function will produce an error.

SphericalObjectLen( ) function

Purpose

Returns the geographic length of a line or polyline object. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

SphericalObjectLen(*obj_expr*, *unit_name*)

obj_expr is an object expression.

unit_name is a string representing the name of a distance unit (for example, “km”).

Return Value

Float

Description

The **SphericalObjectLen( )** function returns the length of an object expression. Note that only line and polyline objects have length values greater than zero; to measure the circumference of a rectangle, ellipse, or region, use the **Perimeter( ) function**.

The **SphericalObjectLen( )** function always returns a value as calculated in a Latitude/Longitude non-projected coordinate system using spherical algorithms. A value of -1 will be returned for data that is in a NonEarth coordinate system since this data cannot be converted into a Latitude/longitude coordinate system.

The **SphericalObjectLen( )** function returns a length measurement in the units specified by the *unit_name* parameter; for example, to obtain a length in miles, specify “mi” as the *unit_name* parameter. See **Set Distance Units statement** for the list of valid unit names.

Example

```
Dim geogr_length As Float
Open Table "streets"
Fetch First From streets
geogr_length = SphericalObjectLen(streets.obj, "mi")
' geogr_length now represents the length of the
' street segment, in miles
```

See Also:

CartesianObjectLen( ) function, **SphericalObjectLen( ) function**

SphericalOffset( ) function

Purpose

Returns a copy of the input object offset by the specified distance and angle using a spherical DistanceType. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

SphericalOffset(*object*, *angle*, *distance*, *units*)

object is the object being offset.

angle is the angle to offset the object.

distance is the distance to offset the object.

units is a string representing the unit in which to measure distance.

Return Value

Object

Description

This function produces a new object that is a copy of the input object offset by *distance* along *angle* (in degrees with horizontal in the positive X-axis being 0 and positive being counterclockwise). The unit string, similar to that used for the **ObjectLen( ) function** or the **Perimeter( ) function**, is the unit for the distance value. The DistanceType used is Spherical. If the coordinate system of the input object is NonEarth, an error will occur, since Spherical DistanceTypes are not valid for NonEarth. This is signified by returning a NULL object. The coordinate system used is the coordinate system of the input object.

There are some considerations for Spherical measurements that do not hold for Cartesian measurements. If you move an object that is in Latitude/Longitude, the shape of the object remains the same, but the area of the object will change. This is because you are picking one offset delta in degrees, and the actual measured distance for a degree is different at different locations.

For the Offset functions, the actual offset delta is calculated at some fixed point on the object (for example, the center of the bounding box), and then that value is converted from the input units into the coordinate system's units. If the coordinate system is Latitude/Longitude, the conversion to degrees uses the fixed point. The actual converted distance measurement could vary at different locations on the object. The distance from the input object and the new offset object is only guaranteed to be exact at the single fixed point used.

Example

```
SphericalOffset(Rect, 45, 100, "mi")
```

See Also:

SphericalOffsetXY( ) function

SphericalOffsetXY( ) function

Purpose

Returns a copy of the input object offset by the specified x- and -offset values using a Spherical DistanceType. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

SphericalOffsetXY(*object*, *xoffset*, *yoffset*, *units*)

object is the object being offset.

xoffset and *yoffset* are the distance along the x- and y-axes to offset the object.

units is a string representing the unit in which to measure distance.

Return Value

Object

Description

The **SphericalOffsetXY( )** function produces a new object that is a copy of the input object offset by *xoffset* along the x-axis and *yoffset* along the y-axis. The unit string, similar to that used for the **ObjectLen( ) function** or the **Perimeter( ) function**, is the unit for distance values. The DistanceType used is Spherical. If the coordinate system of the input object is NonEarth, an error will occur, since Spherical DistanceTypes are not valid for NonEarth. This is signified by returning a NULL object. The coordinate system used is the coordinate system of the input object.

There are some considerations for Spherical measurements that do not hold for Cartesian measurements. If you move an object that is in Latitude/Longitude, the shape of the object remains the same, but the area of the object will change. This is because you are picking one offset delta in degrees, and the actual measured distance for a degree is different at different locations.

For the Offset functions, the actual offset delta is calculated at some fixed point on the object (for example, the center of the bounding box), and then that value is converted from the input units into the coordinate system's units. If the coordinate system is Latitude/Longitude, the conversion to degrees uses the fixed point. The actual converted distance measurement could vary at different locations on the object. The distance from the input object and the new offset object is only guaranteed to be exact at the single fixed point used.

Example

```
SphericalOffsetXY(Rect, 92, -22, "mi")
```

See Also:

SphericalOffset( ) function

SphericalPerimeter( ) function

Purpose

Returns the perimeter of a graphical object. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

SphericalPerimeter(*obj_expr, unit_name*)

obj_expr is an object expression.

unit_name is a string representing the name of a distance unit (for example, “km”).

Return Value

Float

Description

The **SphericalPerimeter( )** function calculates the perimeter of the *obj_expr* object. The **SphericalPerimeter( )** function is defined for the following object types: ellipses, rectangles, rounded rectangles, and polygons. Other types of objects have perimeter measurements of zero. The **SphericalPerimeter( )** function returns a length measurement in the units specified by the *unit_name* parameter; for example, to obtain a length in miles, specify “mi” as the unit_name parameter. See **Set Distance Units statement** for the list of valid unit names.

The **SphericalPerimeter( )** function always returns a value as calculated in a Latitude/Longitude non-projected coordinate system using spherical algorithms. A value of -1 will be returned for data that is in a NonEarth coordinate system since this data cannot be converted into a Latitude/longitude coordinate system. The **SphericalPerimeter( )** function returns approximate results when used on rounded rectangles. MapBasic calculates the perimeter of a rounded rectangle as if the object were a conventional rectangle.

Example

The following example shows how you can use the **SphericalPerimeter( )** function to determine the perimeter of a particular geographic object.

```
Dim perim As Float
Open Table "world"
Fetch First From world
perim = SphericalPerimeter(world.obj, "km")
' The variable perim now contains
' the perimeter of the polygon that's attached to
' the first record in the World table.
```

You can also use the **SphericalPerimeter( )** function within the **Select statement**. The following **Select statement** extracts information from the States table, and stores the results in a temporary table called Results. Because the **Select statement** includes the **SphericalPerimeter( )** function, the Results table will include a column showing each state's perimeter.

```
Open Table "states"
Select state, Perimeter(obj, "mi")
   From states
   Into results
```

See Also:

CartesianPerimeter( ) function, **Perimeter( ) function**

Sqr( ) function

Purpose

Returns the square root of a number. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

Sqr(*num_expr*)

num_expr is a positive numeric expression.

Return Value

Float

Description

The **Sqr( )** function returns the square root of the numeric expression specified by *num_expr*. Since the square root operation is undefined for negative real numbers, *num_expr* should represent a value greater than or equal to zero.

Taking the square root of a number is equivalent to raising that number to the power 0.5. Accordingly, the expression **Sqr(***n***)** is equivalent to the expression *n* ^ 0.5; the **Sqr( )** function, however, provides the fastest calculation of square roots.

Example

```
Dim n As Float
n = Sqr(25)
```

See Also:

Cos( ) function, **Sin( ) function**, **Tan( ) function**

StatusBar statement

Purpose

Displays or hides the status bar, or displays a brief message on it. You can issue this statement in the MapBasic Window in MapInfo Professional.

Syntax

```
StatusBar { Show | Hide }
   [ Message message ]
   [ ViewDisplayPopup { On | Off } ]
   [ EditLayerPopup { On | Off } ]
```

message is a message to display on the status bar.

Description

Use the **StatusBar** statement to show or hide the status bar, or to display a brief message on the status bar.

To print a message to the status bar, use the optional **Message** clause.

```
StatusBar Message "Calculating coordinates..."
```

MapInfo Professional automatically updates the status bar as the user selects various buttons and menu items. Therefore, a message displayed on the status bar may disappear quickly. Therefore, you should not rely on status bar messages to display important prompts.

To display a message that does not disappear, use the **Print statement** to print a message to the Message window.

Use the **ViewDisplayPopup** parameter to allow the user to change view from the status bar. If this parameter is set to **On**, the user will be able to change the zoom level, scale, and cursor location settings from the status bar.

Use the **EditLayerPopup** parameter to allow the user to set the editable layer of a Map window from the status bar. If this parameter is set to **On**, the user will be able to select the editable layer from the status bar.

See Also:

Note statement, **Print statement**

Stop statement

Purpose

Suspends a running MapBasic application, for debugging purposes. You can issue this statement in the MapBasic Window in MapInfo Professional.

Syntax

```
Stop
```

Restrictions

You cannot issue a **Stop** statement from within a user-defined function or within a dialog box's handler procedure; therefore you cannot issue a **Stop** statement to debug a **Dialog statement** while the dialog box is still on the screen.

Description

The **Stop** statement is a debugging aid. It suspends the application which is running, and returns control to the user; presumably, the user in this case is a MapBasic programmer who is debugging a program.

When the **Stop** occurs, a message appears in the MapBasic window identifying the program line number of the **Stop**.

Following a **Stop**, you can use the MapBasic window to investigate the current status of the program. If you type:

```
? Dim
```

into the MapBasic window, MapInfo Professional displays a list of the local variables in use by the suspended program. Similarly, if you type:

```
? Global
```

into the MapBasic window, MapInfo Professional displays a list of the global variables in use.

To display the contents of a variable, type a question mark followed by the variable name. To modify the contents of the variable, type a statement of this form:

```
variable_name = new_value
```

where *variable_name* is the name of a local or global variable, and *new_value* is an expression representing the new value to assign to the variable.

To resume the execution of the application, choose **File > Continue**; note that, while a program is stopped, **Continue** appears on the File menu instead of **Run**. You can also restart a program by typing a **Continue statement** into the MapBasic window.

During a **Stop**, MapInfo Professional keeps the application file open. As long as this file remains open, the application cannot be recompiled. If you use a **Stop** statement, and you then wish to recompile your application, choose **File > Continue** before attempting to recompile.

See Also:

Continue statement

Str$( ) function

Purpose

Returns a string representing an expression (for example, a printout of a number). You can call this function from the MapBasic window in MapInfo Professional.

Syntax

```
Str$( expression )
```

expression is a numeric, Date, Pen, Brush, Symbol, Font, logical, or Object expression.

Return Value

String

Description

The **Str$( )** function returns a string which represents the value of the specified expression.

If *expression* is a negative number, the first character in the returned string is the minus sign (-). If *expression* is a positive number, the first character in the string is a space.

Depending on the number of digits of accuracy in the *expression* you specify, and depending on how many of the digits are to the left of the decimal point, the **Str$( )** function may return a string which represents a rounded value. If you need to control the number of digits of accuracy displayed in a string, use the **Format$( ) function**.

If *expression* is an Object expression, the **Str$( )** function returns a string, indicating the object type: Arc, Ellipse, Frame, Line, Point, Polyline, Rectangle, Region, Rounded Rectangle, or Text.

If *expression* is an Object expression of the form *tablename*.obj and if the current row from that table has no graphic object attached, **Str$( )** returns a null string.

Passing an uninitialized Object variable to the **Str$( )** function generates an error.

If *expression* is a Date, the output from **Str$( )** depends on how the user's computer is configured. For example, the following expression:

```
Str$( NumberToDate(19951231) )
```

might return “12/31/1995” or “1995/12/31” (etc.) depending on the date formatting in use on the user's computer. To control how **Str$( )** formats dates, use the **Set Format statement**.

If *expression* is a number, the **Str$( )** function uses a period as the decimal separator, even if the user's computer is set up to use another character as decimal separator. The **Str$( )** function never includes thousands separators in the return string. To produce a string that uses the thousands separator and decimal separator specified by the user, use the **FormatNumber$( ) function**.

Example

```
Dim s_spelled_out As String, f_profits As Float
f_profits = 123456
s_spelled_out = "Annual profits: $" + Str$(f_profits)
```

See Also:

Format$( ) function, **FormatNumber$( ) function**, **Set Format statement**, **Val( ) function**

String$( ) function

Purpose

Returns a string built by repeating a specified character some number of times. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

String$(*num_expr*, *string_expr*)

num_expr is a positive integer numeric expression.

string_expr is a string expression.

Return Value

String

Description

The **String$( )** function returns a string *num_expr* characters long; this result string consists of *num_expr* occurrences of the first character from the *string_expr* string. Thus, the *num_expr* expression should be a positive integer value, indicating the desired length of the result (in characters).

Example

```
Dim filler As String
filler = String$(5, "ABCDEFGH")
' at this point, filler contains the string "AAAAA"
' (5 copies of the 1st character from the string)
```

See Also:

Space$( ) function

StringCompare( ) function

Purpose

Performs case-sensitive string comparisons. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

StringCompare(*string1*, *string2*)

string1 and *string2* are string expressions.

Return Value

SmallInt: -1 if first string precedes second; 1 if first string follows second; zero if strings are equal.

Description

The **StringCompare( )** function performs case-sensitive string comparisons. MapBasic string comparisons which use the "=" operator are case-insensitive. Thus, a comparison expression such as the following:

```
If "ABC" = "abc" Then
```

evaluates as TRUE, because string comparisons are case-insensitive.

The **StringCompare( )** function performs a case-sensitive string comparison and returns an indication of how the strings compare.

Return value:	When:
-1	first string precedes the second string, alphabetically
0	the two strings are equal
1	first string follows the second string, alphabetically

Example

The function call `StringCompare("ABC", "abc")` returns a value of -1, since "A" precedes "a" in the set of character codes.

See Also:

Like( ) function, **StringCompareIntl( ) function**

StringCompareIntl( ) function

Purpose

Performs language-sensitive string comparisons. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

StringCompareIntl(*string1*, *string2*)

string1 and *string2* are the string expressions being compared.

Return Value

SmallInt: -1 if first string precedes second; 1 if first string follows second; zero if strings are equal.

Description

The **StringCompareIntl( )** function performs language-sensitive string comparisons. Call this function if you need to determine the alphabetical order of two strings, and the strings contain characters that are outside the ordinary U.S. character set (for example, umlauts).

The comparison uses whatever language settings are in use on the user's computer. For example, a Windows user can control language settings through the Control Panel.

Return value:	When:
-1	first string precedes the second string, using the current language setting
0	the two strings are equal
1	first string follows the second string, using the current language setting

See Also:

Like( ) function, **StringCompare( ) function**

StringToDate( ) function

Purpose

Returns a Date value, given a string. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

StringToDate(*datestring*)

datestring is a string expression representing a date.

Return Value

Date

Description

The **StringToDate( )** function returns a Date value, given a string that represents a date. MapBasic interprets the date string according to the date-formatting options that are set up on the user's computer. Computers within the U.S. are usually configured to format dates as Month/Day/Year, but computers in other countries are often configured with a different order (for example, Day/Month/Year) or a different separator character (for example, a period instead of a /). To force the **StringToDate( )** function to apply U.S. formatting conventions, use the **Set Format statement**.

To avoid the entire issue of how the user's computer is set up, call the **NumberToDate( ) function** instead of **StringToDate( )**. The **NumberToDate( ) function** is not affected by how the user's computer is set up.

The *datestring* argument must indicate the month (1 - 12, represented as one or two digits) and the day of the month (1 - 31, represented as one or two digits). You can specify the year as a four-digit number or as a two-digit number, or you can omit the year entirely. If you do not specify a year, MapInfo Professional uses the current year. If you specify the year as a two-digit number (for example, 96), MapInfo Professional uses the current century or the century as determined by the **Set Date Window statement**

Example

The following example specifies date strings with U.S. formatting: Month/Day/Year. Before calling **StringToDate( )**, this program calls the **Set Format statement** to guarantee that the U.S. date strings are interpreted correctly, regardless of how the system is configured.

```
Dim d_start, d_end As Date

Set Format Date "US"
d_start = StringToDate("12/17/92")
d_end = StringToDate("01/02/1995")
Set Format Date "Local"
```

In this example, the variable Date1 = 19890120, Date2 = 20101203 and MyYear = 1990.

```
DIM Date1, Date2 as Date
DIM MyYear As Integer
   Set Format Date "US"
   Set Date Window 75
   Date1 = StringToDate("1/20/89")
   Date2 = StringToDate("12/3/10")
   MyYear = Year("12/30/90")
```

These results are due to the **Set Date Window statement** which allows you to control the century value when given a two-digit year.

See Also:

NumberToDate( ) function, **Set Format statement**, **Str$( ) function**

StringToDateTime function

Purpose

Returns a DateTime value given a string that represents a date and time. MapBasic interprets the date/time string according to the date and time-formatting options that are set up on the user's computer. At least one space must be between the date and time. See **StringToDate( ) function** and **StringToTime function** for details. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

```
StringToDateTime (String)
```

Return Value

DateTime

Example

Copy this example into the MapBasic window for a demonstration of this function.

```
dim strX as string
dim Z as datetime
strX = "19990912041345789"
Z = StringtoDateTime(strX)
Print FormatDate$(Z)
Print FormatTime$(Z,"hh:mm:ss.fff tt")
```

StringToTime function

Purpose

Returns a Time value given a string that represents a time. MapBasic interprets the time string according to the time-formatting options that are set up on the user's computer. However, either 12 or 24-hour time representations are accepted. In addition, the less significant components of a time may be omitted. In other words, the hour must be specified, but the minutes, seconds, and milliseconds are optional. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

```
StringToTime (String)
```

Return Value

Time

Example

Copy this example into the MapBasic window for a demonstration of this function.

```
dim strY as string
dim X as time
strY = "010203000"
X = StringtoTime(strY)
Print FormatTime$ (X,"hh:mm:ss.fff tt")
```

StyleAttr( ) function

Purpose

Returns one attribute of a Pen, Brush, Font, or Symbol style. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

StyleAttr(*style*, *attribute*)

style is a Pen, Brush, Font, or Symbol style value.

attribute is an integer code specifying which component of the style should be returned.

Return Value

string or integer, depending on the attribute parameter.

Description

The **StyleAttr( )** function returns information about a Pen, Brush, Symbol, or Font style.

Each style type consists of several components. For example, a Brush style definition consists of three components: pattern, foreground color, and background color. When you call the **StyleAttr( )** function, the *attribute* parameter controls which style attribute is returned.

The *attribute* parameter must be one of the codes in the table below. Codes in the left column (for example, PEN_WIDTH) are defined in MAPBASIC.DEF.

attribute setting	ID	StyleAttr( ) returns:
BRUSH_PATTERN	1	Integer, indicating the Brush style's pattern.
BRUSH_FORECOLOR	2	Integer, indicating the Brush style's foreground color, as an RGB value.
BRUSH_BACKCOLOR	3	Integer, indicating the Brush style's background color as an RGB value, or -1 if the brush has a transparent background.
FONT_NAME	1	String, indicating the Font name.

attribute setting	ID	StyleAttr( ) returns:
FONT_STYLE	2	Integer value, indicating the Font style (0 = Plain, 1 = Bold, etc.); see **Font clause** for details.
FONT_POINTSIZE	3	Integer indicating the Font size, in points. ⓘ If the Text object is in a mappable table (as opposed to a Layout window), the point size is returned as zero, and the text height is dictated by the Map window's current zoom.
FONT_FORECOLOR	4	Integer value representing the RGB color of the font foreground.
FONT_BACKCOLOR	5	Integer value representing the RGB color of the font background, or -1 if the font has a transparent background. If the font style includes a halo, the RGB color represents the halo color.
PEN_WIDTH	1	Integer, indicating the Pen style's line width, in pixels or points.
PEN_PATTERN	2	Integer, indicating the Pen style's pattern.
PEN_COLOR	4	Integer, indicating the Pen style's RGB color value.
PEN_INDEX	5	Integer, representing the pen index number from the pen pattern.
PEN_INTERLEAVED	6	Logical, TRUE if line style is interleaved.
SYMBOL_KIND	7	Integer, indicating the type of symbol: 2 for TrueType symbols; 3 for bitmap file symbols.
SYMBOL_CODE	1	Integer, indicating the Symbol style's shape code. Applies to TrueType symbols.
SYMBOL_COLOR	2	Integer, indicating the Symbol style's color as an RGB value.
SYMBOL_POINTSIZE	3	Integer from 1 to 48, indicating the Symbol's size, in points.
SYMBOL_ANGLE	4	Float number, indicating the rotation angle of a TrueType symbol.
SYMBOL_FONT_NAME	5	String, indicating the name of the font used by a TrueType symbol.

attribute setting	ID	StyleAttr( ) returns:
SYMBOL_FONT_STYLE	6	Integer, indicating the style attributes of a TrueType symbol (0 = plain, 1 = Bold, etc.). See **Symbol clause** for a listing of possible values.
SYMBOL_CUSTOM_NAME	8	String, indicating the file name used by a bitmap file symbol.
SYMBOL_CUSTOM_STYLE	9	Integer, indicating the style attributes of a bitmap file symbol (0 = plain, 1 = show background, etc.). See **Symbol clause** for a listing of possible values.

Error Conditions

ERR_FCN_ARG_RANGE (644) error is generated if an argument is outside of the valid range.

Example

The following example uses the **CurrentPen( ) function** to determine the pen style currently in use by MapInfo Professional, then uses the **StyleAttr( )** function to determine the thickness of the pen, in pixels.

```
Include "mapbasic.def"
Dim cur_width As Integer
cur_width = StyleAttr(CurrentPen( ), PEN_WIDTH)
```

See Also:

Brush clause, **Font clause**, **Pen clause**, **Symbol clause**, **MakeBrush( ) function**, **MakeFont( ) function**, **MakePen( ) function**, **MakeSymbol( ) function**

StyleOverrideInfo( ) function

Returns information about a specific display style override.

Syntax

StyleOverrideInfo (*window_id*, *layer_number*, *override_index*, *attribute*)

window_id is the integer window identifier of a Map window.

layer_number is the number of a layer in the current Map window (for example, 1 for the top layer); to determine the number of layers in a Map window, call the **MapperInfo( ) function**.

override_index is an integer index (1-based) for the override definition within the layer.

attribute is a code indicating the type of information to return; see table below.

Return Value

Return value depends on attribute parameter.

Description

This function returns information about the specified display style override for one layer in an existing Map window. The *layer_number* must be a valid layer (1 is the topmost table layer, and so on). The *attribute* parameter must be one of the codes from the following table; codes are defined in MAPBASIC.DEF.

Attribute Code	ID	LayerInfo( ) Return Value
STYLE_OVR_INFO_NAME	1	Style override name.
STYLE_OVR_INFO_VISIBILITY	2	Smallint value, indicating whether the style override is visible; Return value will be one of the values: • STYLE_OVR_INFO_VIS_OFF (0) override is disabled/off; never visible • STYLE_OVR_INFO_VIS_ON (1) override is currently visible in the map • STYLE_OVR_INFO_VIS_ZOOM (2) override is currently not visible because it's outside the map zoom range
STYLE_OVR_INFO_ZOOM_MIN	3	Float value, indicating the minimum zoom value at which the style override displays.
STYLE_OVR_INFO_ZOOM_MAX	4	Float value, indicating the maximum zoom value at which the style override displays.
STYLE_OVR_INFO_ARROWS	5	Logical value; TRUE if override displays direction arrows on linear objects.
STYLE_OVR_INFO_NODES	6	Logical value; TRUE if override displays object nodes.
STYLE_OVR_INFO_CENTROIDS	7	Logical value; TRUE if override displays object centroids.
STYLE_OVR_INFO_ALPHA	8	SmallInt value, representing the alpha factor for the specified override. • 0=fully transparent. • 255=fully opaque.
STYLE_OVR_INFO_TRANSLUCENCY	9	SmallInt value, representing the translucency percentage for the specified override. • 100=fully transparent. • 0=fully opaque.

Attribute Code	ID	LayerInfo( ) Return Value
STYLE_OVR _INFO_LINE	10	Pen style used for displaying linear objects. If there are multiple styles, the bottom Pen style is returned.
STYLE_OVR _INFO_PEN	11	Pen style used for displaying the borders of filled objects. If there are multiple styles, the bottom Pen style is returned.
STYLE_OVR_INFO_BRUSH	12	Brush style used for displaying filled objects. If there are multiple styles, the bottom Brush style is returned.
STYLE_OVR_INFO_SYMBOL	13	Symbol style used for displaying point objects. If there are multiple styles, the bottom Symbol style is returned.
STYLE_OVR_INFO_FONT	14	Font style used for displaying text objects. If there are multiple styles, the bottom Font style is returned.
STYLE_OVR_INFO_SYMBOL_COUNT	15	SmallInt value, indicating the number of multiple SYMBOL styles.
STYLE_OVR_INFO_LINE_COUNT	16	SmallInt value, indicating the number of multiple LINE styles.
STYLE_OVR_INFO_PEN_COUNT	17	SmallInt value, indicating the number of multiple PEN styles.
STYLE_OVR_INFO_BRUSH_COUNT	18	SmallInt value, indicating the number of multiple BRUSH styles.
STYLE_OVR_INFO_FONT_COUNT	19	SmallInt value, indicating the number of multiple FONT styles.

Example

```
StyleOverrideInfo(nMID, nLayer, nOverride, STYLE_OVR_INFO_PEN_COUNT)
```

See Also:

LabelOverrideInfo( ) function, **LayerStyleInfo( ) function**, **Set Map statement**, **LayerInfo( ) function**

Sub…End Sub statement

Purpose

Defines a procedure, which can then be called through the **Call statement**.

Syntax

```
Sub proc_name [ ( [ ByVal ] parameter As var_type [ , ... ] ) ]
    statement_list
End Sub
```

proc_name is the name of the procedure.

parameter is the name of a procedure parameter.

var_type is a standard MapBasic variable type (for example, integer) or a custom variable Type.

statement_list is a list of zero or more statements comprising the body of the procedure.

Restrictions

You cannot issue a **Sub…End Sub** statement through the MapBasic window.

Description

The **Sub…End Sub** statement defines a sub procedure (often, simply called a procedure). Once a procedure is defined, other parts of the program can call the procedure through the **Call statement**.

Every **Sub…End Sub** definition must be preceded by a **Declare Sub statement**.

A procedure may have zero or more parameters. *parameter* is the name of the parameter; each of a procedure's parameters must be unique. If a sub procedure has two or more parameters, they must be separated by commas.

By default, each sub procedure parameter is defined “by reference.” When a sub procedure has a by-reference parameter, the caller must specify the name of a variable as the parameter. Subsequently, if the sub procedure alters the contents of the by-reference parameter, the caller's variable will reflect the change. This allows the caller to examine the results returned by the sub procedure. Alternately, any or all sub procedure parameters may be passed “by value” if the keyword **ByVal** appears before the parameter name in the **Sub** statement. When a parameter is passed by value, the sub procedure receives a copy of the value of the caller's parameter expression; thus, the caller can pass any expression, rather than having to pass the name of a variable. A sub procedure can alter the contents of a **ByVal** parameter without having any impact on the status of the caller's variables.

A procedure can take an array as a parameter. To declare a procedure parameter as an array, place parentheses after the parameter name in the **Sub…End Sub** statement (as well as in the **Declare Sub statement**). The following example defines a procedure which takes an array of Integers as a parameter.

```
Sub ListProcessor(items( ) As Integer)
```

When a sub procedure expects an array as a parameter, the procedure's caller must specify the name of an array variable, without the parentheses.

If a sub procedure's local variable has the same name as an existing global variable, all of the sub procedure's references to that variable name will access the local variable.

A sub procedure terminates if it encounters an **Exit Sub statement**.

You cannot pass arrays, custom Type variables, or Alias variables as **ByVal** (by-value) parameters to sub procedures. However, you can pass any of those data types as by-reference parameters.

Example

In the following example, the sub procedure Cube cubes a number (raises the number to the power of three), and returns the result. The sub procedure takes two parameters; the first parameter contains the number to be cubed, and the second parameter passes the results back to the caller.

```
Declare Sub Main
Declare Sub Cube(ByVal original As Float, cubed As Float)

Sub Main
   Dim x, result As Float
   Call Cube(2, result)
   ' result now contains the value: 8 (2 x 2 x 2)
   x = 1
   Call Cube(x + 2, result)
   ' result now contains the value: 27 (3 x 3 x 3)
End Sub

Sub Cube (ByVal original As Float, cubed As Float)
   ' Cube the "original" parameter value, and store
   ' the result in the "cubed" parameter.
   cubed = original ^ 3
End Sub
```

See Also:

Call statement, **Declare Sub statement**, **Dim statement**, **Exit Sub statement**, **Function…End Function statement**, **Global statement**

Symbol clause

Purpose

Specifies a symbol style for point objects. You can use this clause in the MapBasic Window in MapInfo Professional.

MapInfo 3.0 Symbols

Syntax

Symbol (*shape*, *color*, *size*)

shape is an integer, 31 or larger, specifying which character to use from MapInfo Professional's standard symbol set. To create an invisible symbol, use 31; see table below. The standard set of symbols includes symbols 31 through 67, but the user can customize the symbol set by using the Symbol application.

color is an integer RGB color value; see **RGB( ) function**.

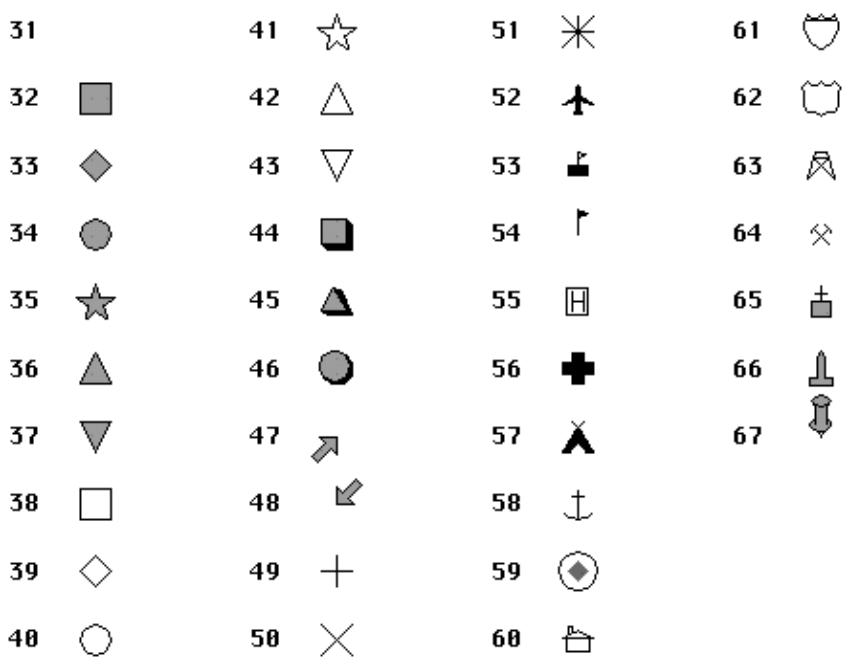
size is an integer point size, from 1 to 48.

Description

The Symbol clause specifies the settings that dictate the appearance of a point object. Note that Symbol is a clause, not a complete MapBasic statement. Various object-related statements, such as Create Point, allow you to specify a Symbol clause; this lets you specify the symbol style of the new object.

Some MapBasic statements (for example, **Alter Object…Info OBJ_INFO_SYMBOL**) take a **Symbol** expression as a parameter (for example, the name of a Symbol variable), rather than a full Symbol clause (the keyword **Symbol** followed by the name of a Symbol variable).

The following table lists the standard symbol shapes that are available when you use MapInfo 3.0 symbols:



Example

The following example shows how a **Set Map statement** can incorporate a **Symbol** clause. **Set Map statement** below specifies that symbol objects in the mapper's first layer should be displayed using symbol 34 (a filled circle), filled in red, at a size of eighteen points.

```
Include "mapbasic.def"

Set Map
   Layer 1 Display Global
   Global Symbol MakeSymbol(34,RED,18)
```

True Type Font

Syntax

Symbol (*shape*, *color*, *size*, *fontname*, *fontstyle*, *rotation*)

shape is an integer, 32 or larger, specifying which character to use from a TrueType font. To create an invisible symbol, use 32.

color is an integer RGB color value; see **RGB( ) function**.

size is an integer point size, from 1 to 48.

fontname is a string representing a TrueType font name (for example, “WingDings”).

fontstyle is an integer code controlling attributes such as bold; see table below.

rotation is a floating-point number representing a rotation angle, in degrees.

Description

When you specify a TrueType font symbol, the fontstyle argument controls attributes such as Bold. The following table lists the fontstyle values you can specify:

fontstyle value	Symbol Style
0	Plain
1	Bold
16	Border (black outline)
32	Drop Shadow
256	Halo (white outline)

To specify two or more style attributes, add the values from the left column. For example, to specify both the Bold and the Drop Shadow attributes, use a fontstyle value of 33. Styles 16 and 256 are mutually exclusive.

Custom Bitmap File

Syntax

Symbol (*filename*, *color*, *size*, *customstyle*)

filename is a string up to 31 characters long, representing the name of a bitmap file. The file must be in the CustSymb directory.

color is an integer RGB color value; see **RGB( ) function**.

size is an integer point size, from 1 to 48.

customstyle is an integer code controlling color and background attributes. See table below.

Description

When you specify a custom symbol, the *customstyle* argument controls background, color, and display size settings, as described in the following table.

customstyle value	Symbol Style
0	The Show Background, the Apply Color, and the Display at Actual Size settings are off; the symbol appears in its default state at the point size specified by the size parameter. White pixels in the bitmap are displayed as transparent, allowing whatever is behind the symbol to show through.
1	The Show Background setting is on; white pixels in the bitmap are opaque.
2	The Apply Color setting is on; non-white pixels in the bitmap are replaced with the symbol's color setting.
3	Both Show Background and Apply Color are on.
4	The Display at Actual Size setting is on; the bitmap image is rendered at its native width and height in pixels.
5	The Show Background and Display at Actual Size settings are on.
7	The Show Background, the Apply Color, and the Display at Actual Size settings are on.

Symbol Expression

Syntax

Symbol *symbol_expr*

symbol_expr is a Symbol expression, which can either be the name of a Symbol variable, or a function call that returns a Symbol value, for example, MakeSymbol(shape, color, size).

See Also:

MakeCustomSymbol( ) function, **MakeFontSymbol( ) function**, **MakeSymbol( ) function**, **StyleAttr( ) function**

SystemInfo( ) function

Purpose

Returns information about the operating system or software version. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

```
SystemInfo(attribute)
```

attribute is an integer code indicating which system attribute to query.

> The MAPBASIC.DEF constant for this function is 18.

Return Value

SmallInt, logical, or string

Description

The **SystemInfo( )** function returns information about MapInfo Professional's system status. The attribute can be any of the codes listed in the table below. The codes are defined in MAPBASIC.DEF

attribute code	ID	SystemInfo( ) Return Value
SYS_INFO_APPLICATIONWND	7	Integer, representing the Windows *HWND* specified by the **Set Application Window statement** (or zero if no such *HWND* has been set).
SYS_INFO_APPVERSION	2	Integer value: the version number with which the application was compiled, multiplied by 100.
SYS_INFO_CHARSET	5	String value: the name of the native character set.
SYS_INFO_COPYPROTECTED	6	Logical value: TRUE means the user is running a copy-protected version of MapInfo Professional.
SYS_INFO_DATE_FORMAT	11	String: “US” or “Local” depending on the date formatting in effect; for details, see **Set Format statement**.

attribute code	ID	SystemInfo( ) Return Value
SYS_INFO_DDESTATUS	8	Integer value, representing the number of elements in the DDE execute queue. If the queue is empty, **SystemInfo( )** returns zero (if an incoming execute would be enqueued) or -1 (if an execute would be executed immediately).
SYS_INFO_DIG_INSTALLED	12	Logical value: TRUE if a digitizer is installed, along with a compatible driver.
SYS_INFO_DIG_MODE	13	Logical value: TRUE if Digitizer Mode is on.
SYS_INFO_MAPINFOWND	9	Integer, representing a Windows *HWND* of the MapInfo Professional frame window, or zero on non-Windows platforms.
SYS_INFO_MDICLIENTWND	15	Integer, representing a Windows *HWND* of the MapInfo Professional MDICLIENT window, or 0 on non-Windows platforms.
SYS_INFO_MIPLATFORM	14	Integer value, indicating the type of MapInfo Professional software that is running.
SYS_INFO_MIVERSION	3	Integer value, indicating the version of MapInfo Professional that is currently running, multiplied by 100.
SYS_INFO_NUMBER_FORMAT	10	String: “9,999.9” or “Local” depending on the number formatting in effect; for details, see **Set Format statement**.
SYS_INFO_PLATFORM	1	Integer value, indicating the hardware platform on which the application is running. The return value will be PLATFORM_WIN.
SYS_INFO_PRODUCTLEVEL	16	Integer value, indicating the product level of the version of MapInfo Professional that is running (for example, 1100 for MapInfo Professional 11.0).
SYS_INFO_RUNTIME	4	Logical value: TRUE if invoked within a run-time version of MapInfo Professional, FALSE otherwise.

attribute code	ID	SystemInfo( ) Return Value
SYS_INFO_APPIDISPATCH (value=17)	17	Integer, representing the IDispatch OLE Automation pointer for the MapInfo Application.
SYS_INFO_MIBUILD_NUMBER)	18	This function has an attribute to return the current build number so you can distinguish between MapInfo Professional point versions in MapBasic. For example, SystemInfo(SYS_INFO_MIVERSION) returns 850 for MapInfo Professional 8.5 and 8.5.2. You can further distinguish between 8.5 and 8.5.2 with SystemInfo(SYS_INFO_MIBUILD_NUMBER), which returns 32 for 8.5, and 60 for 8.5.2.

Error Conditions

ERR_FCN_ARG_RANGE (644) error is generated if an argument is outside of the valid range.

Example

The following example uses the **SystemInfo( )** function to determine what type of MapInfo software is running. The program only calls a DDE-related procedure if the program is running some version of MapInfo Professional.

```
Declare Sub DDE_Setup

If SystemInfo(SYS_INFO_PLATFORM) = PLATFORM_WIN Then
   Call DDE_Setup
End If
```

TableInfo( ) function

Purpose

Returns information about an open table. Has a define for FME (Universal Data) tables. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

TableInfo(*table_id*, *attribute*)

table_id is a string representing a table name, a positive integer table number, or 0 (zero).

attribute is an integer code indicating which aspect of the table to return (see table of attributes below). The following example returns the coordsys clause with bounds:

```
TableInfo(table_id, TAB_INFO_COORDSYS_CLAUSE)
TableInfo(table_id, 29)
```

Return Value

String, SmallInt, or logical, depending on the attribute parameter specified.

Description

The **TableInfo( )** function returns one piece of information about an open table.

The *table_id* can be a string representing the name of the open table. Alternately, *table_id* can be a table number. If *table_id* is 0 (zero), the **TableInfo( )** function returns information about the most recently opened, most recently created table; or a table that has just been renamed. This allows a MapBasic program to determine the working name of a table in cases where the **Open Table statement** did not include an **As** clause. If there are no open tables, or if the most recently-opened table has already been closed, the **TableInfo( )** function generates an error.

The *attribute* parameter can be any value from the table below. Codes in the left column (for example, TAB_INFO_NAME) are defined in MAPBASIC.DEF.

attribute code	ID	TableInfo( ) returns
TAB_INFO_BROWSER_LIST	35	String result: indicates which columns will be displayed in a browser. This information is stored in table metadata. Return empty string if this information is absent.
TAB_INFO_COORDSYS_CLAUSE	29	String result, indicating the table's **CoordSys clause**, such as "`CoordSys Earth Projection 1, 0`". Returns empty string if table is not mappable.

TAB_INFO_COORDSYS_CLAUSE_WITHOUT_BOUNDS	37	String result, representing the table's CoordSys clause without bounds.
		`TableInfo(`*`table_id`*`, TAB_INFO_COORDSYS_CLAUSE)`
		returns the coordsys clause **without** bounds
		or
		`TableInfo(`*`table_id`*`, 29)`
		returns the coordsys clause **with** bounds

TAB_INFO_COORDSYS_MINX, TAB_INFO_COORDSYS_MINY, TAB_INFO_COORDSYS_MAXX, TAB_INFO_COORDSYS_MAXY	25 26 27 28	Float results, indicating the minimum or maximum x or y map coordinates that the table is able to store; if table is not mappable, returns zero.

attribute code	ID	TableInfo( ) returns
TAB_INFO_COORDSYS_NAME	30	String result, representing the name of the coordinate system as listed in MAPINFOW.PRJ (but without the optional "\p…" suffix that appears in MAPINFOW.PRJ). Returns empty string if table is not mappable, or if coordinate system is not found in MAPINFOW.PRJ.
TAB_INFO_DESCRIPTION	38	String result: returns a table description string that can be specified in a TAB file. If there is no description in a TAB file, then it returns an empty string.
TAB_INFO_EDITED	9	Logical result; TRUE if table has unsaved edits.
TAB_INFO_FASTEDIT	10	Logical result; TRUE if the table has **FastEdit** mode turned on, FALSE otherwise. (See **Set Table statement** for information on FastEdit mode.)
TAB_INFO_ISMANAGED	41	Logical result: TRUE if table is managed in a library service.
TAB_INFO_MAPPABLE	5	Logical result; TRUE if the table is mappable.
TAB_INFO_MAPPABLE_TABLE	12	String result indicating the name of the table containing graphical objects. Use this code when you are working with a table that is actually a relational join of two other tables, and you need to know the name of the base table that contains the graphical objects.
TAB_INFO_MINX, TAB_INFO_MINY, TAB_INFO_MAXX, TAB_INFO_MAXY	20 21 22 23	Float results, indicating the minimum and maximum x- and y-coordinates of all objects in the table.
TAB_INFO_NAME	1	String result, indicating the name of the table.
TAB_INFO_NCOLS	4	SmallInt, indicating the number of columns.
TAB_INFO_NREFS	31	SmallInt, indicating the number of other base tables that reference this table. (Returns zero for most tables, or non-zero in cases where a table is defined as a join of two other tables, such as a StreetInfo table.) May only be used with base tables (TAB_TYPE_BASE).
TAB_INFO_NROWS	8	SmallInt, indicating the number of rows.

attribute code	ID	TableInfo( ) returns
TAB_INFO_NUM	2	SmallInt result, indicating the number of the table.
TAB_INFO_PARENTTABLEID	40	String result: returns the table ID from which this TAB file was copied. If this was not created from another TAB file, then it returns an empty string.
TAB_INFO_READONLY	6	Logical result; TRUE if the table is read-only.
TAB_INFO_SEAMLESS	24	Logical result; TRUE if seamless behavior is on for this table.
TAB_INFO_SUPPORT_MZ	32	Logical result: TRUE if table supports m and z-values.
TAB_INFO_TABFILE	19	String result, representing the table's full directory path. Returns an empty string if the table is a query table.
TAB_INFO_TABLEID	39	String result: returns the unique table ID for a TAB file. If there is no Table ID in a TAB file, then it returns an empty string.
TAB_INFO_TEMP	7	Logical result; TRUE if the table is temporary (for example, QUERY1).
TAB_INFO_THEME_METADATA	36	Logical result; TRUE if the table has default theme metadata.

attribute code	ID	TableInfo( ) returns
TAB_INFO_TYPE	3	SmallInt result, indicating the type of table. The returned value will match one of these: • TAB_TYPE_BASE (1) if a normal or seamless table • TAB_TYPE_RESULT (2) if results of a query • TAB_TYPE_VIEW (3) if table is actually a view; for example, StreetInfo tables are actually views • TAB_TYPE_IMAGE (4) if table is a raster image • TAB_TYPE_LINKED (5) if this table is linked • TAB_TYPE_WMS (6) if table is from a Web Map Service • TAB_TYPE_WFS (7) if table is from a Web Feature Service • TAB_TYPE_FME (8) if table is opened through FME • TAB_TYPE_TILESERVER (9) if table is a raster image from a Tile Server
TAB_INFO_UNDO	11	Logical result; TRUE if the undo system is being used with the specified table, or FALSE if the undo system has been turned off for the table through the **Set Table statement**.
TAB_INFO_USERBROWSE	14	Logical result: FALSE if a **Set Table statement** has set the **UserBrowse** option to **Off**.
TAB_INFO_USERCLOSE	15	Logical result: FALSE if a **Set Table statement** has set the **UserClose** option to **Off**.
TAB_INFO_USERDISPLAYMAP	18	Logical result: FALSE if a **Set Table statement** has set the **UserDisplayMap** option to **Off**.
TAB_INFO_USEREDITABLE	16	Logical result: FALSE if a **Set Table statement** has set the **UserEdit** option to **Off**.
TAB_INFO_USERMAP	13	Logical result: FALSE if a **Set Table statement** has set the **UserMap** option to **Off**.
TAB_INFO_USERREMOVEMAP	17	Logical result: FALSE if a **Set Table statement** has set the **UserRemoveMap** option to **Off**.

attribute code	ID	TableInfo( ) returns
TAB_INFO_Z_UNIT	34	String result: indicates distance units used for z-values. Return empty string if units are not specified.
TAB_INFO_Z_UNIT_SET	33	Logical result: TRUE if unit is set for z-values.

Error Conditions

ERR_TABLE_NOT_FOUND (405) error is generated if the specified table was not available.

ERR_FCN_ARG_RANGE (644) error is generated if an argument is outside of the valid range.

Example

```
Include "mapbasic.def"
Dim i_numcols As SmallInt, L_mappable As Logical
Open Table "world"
i_numcols = TableInfo("world", TAB_INFO_NCOLS)
L_mappable = TableInfo("world", TAB_INFO_MAPPABLE)

TableInfo(table_id, TAB_INFO_COORDSYS_CLAUSE)
TableInfo(table_id, 29)  -  Returns the coordsys clause with bounds
```

See Also:

Open Table statement

TableListInfo( ) function

Purpose

Returns information about the Table List window.

Syntax

TableListInfo(*attribute*)

attribute is a code indicating the type of information to return; see table below.

Description

The **TableListInfo( )** function returns one piece of information about the Table List window.

The *attribute* parameter is a value from the table below. Codes in the left column are defined in MAPBASIC.DEF.

attribute code	ID	TableInfo( ) returns
TL_INFO_SEL_COUNT	1	Smallint result, indicating the number of selected items.

Examples

```
TableListInfo(TL_INFO_SEL_COUNT)
```

The following example uses this function in conjunction with a custom item on the Table List shortcut menu.

```
include "mapbasic.def"
include "menu.def"
declare sub main
declare sub ShowTABPaths
'================================================================
sub main
' Add new item to Table List context menu
   alter menu ID M_SHORTCUT_TLV_TABLES add
       "Show TAB path..." calling ShowTABPaths
end sub
'================================================================
sub ShowTABPaths()
' Get the number of selected items
   dim selCount as integer
   selCount = TableListInfo(TL_INFO_SEL_COUNT)
' Print the table name and TAB file location fo all selected items
   dim index as integer
   for index = 1 to selCount
' Get the table id
       dim tableId as integer
       tableId = TableListSelectionInfo(index, TL_SEL_INFO_ID)
' Use the table in call to get the TAB path
       dim tablePath as string
       tablePath = TableInfo(tableId, TAB_INFO_TABFILE)
' Print the info
       print tableName + " - " + tablePath
   next
end sub
```

See Also:

TableListSelectionInfo( ) function

TableListSelectionInfo( ) function

Purpose

Returns information about a selected item in the Table List window.

Syntax

TableListSelectionInfo (*selection_index*, *attribute*)

selection_id is the index of a selected item in Table List.

attribute is a code indicating the type of information to return; see table below.

Description

The *attribute* parameter can be any value from the table below. Codes in the left column are defined in MAPBASIC.DEF.

attribute code	ID	TableInfo( ) returns
TL_SEL_INFO_NAME	1	String result, representing the name of the selected.
TL_SEL_INFO_ID	2	Smallint value, indicating the id of the table associated with the selected item. This value can be used in calls to TableInfo().

Example

```
TableListSelectionInfo(index, TL_SEL_INFO_ID)
```

See Also:

TableListInfo( ) function

Tan( ) function

Purpose

Returns the tangent of a number. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

Tan(*num_expr*)

num_expr is a numeric expression representing an angle in radians.

Return Value

Float

Description

The **Tan( )** function returns the tangent of the numeric *num_expr* value, which represents an angle in radians.

To convert a degree value to radians, multiply that value by DEG_2_RAD (0.01745329252). To convert a radian value into degrees, multiply that value by RAD_2_DEG (57.29577951). (Note that your program will need to Include “MAPBASIC.DEF” in order to reference DEG_2_RAD or RAD_2_DEG).

Example

```
Include "mapbasic.def"

Dim x, y As Float

x = 45 * DEG_2_RAD
y = Tan(x)
' y will now be equal to 1,
' since the tangent of 45 degrees is 1
```

See Also:

Acos( ) function, **Asin( ) function**, **Atn( ) function**, **Cos( ) function**, **Sin( ) function**

TempFileName$( ) function

Purpose

Returns a name that can be used when creating a temporary file. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

TempFileName$(*dir*)

dir is the string that specifies the directory that will store the file; "" specifies the system temporary storage directory.

Return Value

Returns a string that specifies a unique file name, including its path.

Description

Use the **TempFileName$( )** function when you need to create a temporary file, but you do not know what file name to use.

When you call **TempFileName$( )**, MapBasic returns a string representing a file name. The **TempFileName$( )** function does not actually create the file. To create the file, issue an **Open File statement**.

If the *dir* parameter is an empty string (""), the returned file name will represent a file in the system's temporary storage directory, such as “G:\TEMP\~MAP0023.TMP”.

In a networked environment, it is possible that two users could attempt to create the same file at the same time. If you try to create a file using a filename returned by **TempFileName$( )**, and an error occurs because that file already exists, it is likely that another network user created the file moments after your program called **TempFileName$( )**. To reduce the likelihood of such file conflicts, issue the **Open File statement** immediately after calling **TempFileName$( )**. To eliminate all chances of file sharing conflicts, create an error handler, and enable the error handler (by issuing an **OnError statement**) before issuing the **Open File statement**.

See Also:

FileExists( ) function

Terminate Application statement

Purpose

Halts execution of a running or sleeping MapBasic application. You can issue this statement from the MapBasic Window in MapInfo Professional.

Syntax

`Terminate Application` *`app_name`*

app_name is a string representing the name of the running application (for example, “SCALEBAR.MBX”).

Description

If a MapBasic program creates custom menu items or ButtonPad buttons, that MapBasic program can remain in memory, “sleeping,” until the user exits MapInfo Professional. To force a sleeping application to halt, issue a **Terminate Application** statement. For example, if you need to halt an application for debugging purposes, you can issue the **Terminate Application** statement from the MapBasic Window.

If your application launches another MapBasic application (using the **Run Application statement**), you can use the **Terminate Application** statement to halt the other MapBasic application.

Terminate Application allows one program to halt another program. The easiest way for a program to halt itself is to issue an **End Program statement**.

See Also:

End Program statement, **Run Application statement**

TextSize( ) function

Purpose

Returns the point size of a text object in a window. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

TextSize(*window_id*, *text_obj*)

window_id is the integer window identifier of a Map or Layout window. Call the **FrontWindow( ) function** or the **WindowID( ) function** to obtain window identifiers.

text_obj is a text object.

> If the text object is from a Map window, the window ID must be the ID of a Map window. If the text object is from a Layout, the window ID must be the ID of a Layout window.

Return Value

Float

Description

The **TextSize( )** function will return the point size of a text object in a window at its current zoom level. This function correlates to selecting a text object and selecting **Edit > Get Info** or pressing F7.

Example

If the active window is a map and a text object is selected:

```
print TextSize(FrontWindow( ), selection.obj)
```

See Also:

Font clause

Time( ) function

Purpose

The time function returns the current system time in string format. The time may be returned in 12- or 24-hour time format. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

StringVar = **Time**(*Format*)

Description

StringVar is a string variable which will be given the system time in HH:MM:SS format. Format is an integer value indicating the format of the string to return. The time will be returned in 24-hour format if Format is 24. Any other value will return the time in 12-hour format.

Timer( ) function

Purpose

Returns the number of elapsed seconds. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

Timer( )

Return Value

Integer

Description

The **Timer( )** function returns the number of seconds that have elapsed since Midnight, January 1, 1970. By calling the **Timer( )** function before and after a particular operation, you can time how long the operation took (in seconds).

Example

```
Declare Sub Ubi

Dim start, elapsed As Integer

start = Timer( )
Call Ubi
elapsed = Timer( ) - start

'
' elapsed now contains the number of seconds
' that it took to execute the procedure Ubi
'
```

ToolHandler procedure

Purpose

A reserved procedure name; works in conjunction with a special ToolButton (the MapBasic tool).

Syntax

```
Declare Sub ToolHandler
Sub ToolHandler
    statement_list
End Sub
```

statement_list is a list of statements to execute when the user clicks with the MapBasic tool.

Description

ToolHandler is a special-purpose MapBasic procedure name, which operates in conjunction with the MapBasic tool.

Defining a **ToolHandler** procedure is a simple way to add a custom button to MapInfo Professional's Main ButtonPad. However, the button associated with a **ToolHandler** procedure is restricted; you cannot use custom icons or drawing modes with the ToolHandler's button. To create a custom button which has no restrictions, use the **Alter ButtonPad statement** and **Create ButtonPad statement** statements.

If the user runs an application which contains a procedure named **ToolHandler**, a plus-shaped tool (the MapBasic tool) appears on the Main ButtonPad. The MapBasic tool is enabled whenever a Browser, Map, or Layout window is the active window. If the user selects the MapBasic tool and clicks in the Browser, Map, or Layout window, MapBasic automatically calls the **ToolHandler** procedure.

A **ToolHandler** procedure can use the **CommandInfo( ) function** to determine where the user clicked. If the user clicked in a Browser, the CommandInfo( ) function returns the row and column where the user clicked. If the user clicked in a Map, the CommandInfo( ) function returns the map coordinates of the location where the user clicked; these coordinates are in MapBasic's current coordinate system (see **Set CoordSys statement**).

If the user clicked in a Layout window, the CommandInfo( ) function returns the layout coordinates (for example, distance from the upper left corner of the page) where the user clicked; these coordinates are in MapBasic's current paper units (see **Set Paper Units statement**).

By calling the CommandInfo( ) function, you can also detect whether the user held down the shift key and/or the Control key while clicking. This allows you to write applications which react differently to click events than to shift-click events.

To make the MapBasic tool the active tool, issue the statement:

```
Run Menu Command M_TOOLS_MAPBASIC
```

For a **ToolHandler** procedure to take effect, the user must run the application. If an application contains a special procedure name—such as **ToolHandler**—the application “goes to sleep” when the Main procedure runs out of statements to execute.

The Main procedure may be explicit or implied. The application is said to be “sleeping” because the **ToolHandler** procedure is still in memory, although it may be inactive. If the user selects the MapBasic tool and clicks with it, MapBasic automatically calls the **ToolHandler** procedure, so that the procedure may react to the click event.

When any procedure in an application executes the **End Program statement**, the application is completely removed from memory. That is, a program which executes an **End Program statement** is no longer sleeping—it is terminated altogether. So, you can use the **End Program statement** to terminate a **ToolHandler** procedure once it is no longer wanted. Conversely, you should be careful not to issue an **End Program statement** while the **ToolHandler** procedure is still needed.

Depending on the circumstances, a **ToolHandler** procedure may need to issue a **Set CoordSys statement** before determining the coordinates of where the user clicked. If the **ToolHandler** procedure is called because the user clicked in a Browser, no **Set CoordSys statement** is necessary. If the user clicks in a Layout window, the **ToolHandler** procedure may need to issue a **Set CoordSys Layout statement** before determining where the user clicked in the layout. If the user clicks in a Map window, and the application's current coordinate system does not match the coordinate system of the Map (because the application has issued a **Set CoordSys statement**), the **ToolHandler** procedure may need to issue a **Set CoordSys statement** before determining where the user clicked in the map.

Example

The following program sets up a **ToolHandler** procedure that will be called if the user selects the MapBasic tool, then clicks on a Map, Browser, or Layout window. In this example, the **ToolHandler** simply displays the location where the user clicked.

```
Include "mapbasic.def"
Declare Sub ToolHandler
Note "Ready to test the MapBasic tool."

Sub ToolHandler
   Note "x:" + Round(CommandInfo(CMD_INFO_X), 0.1) + Chr$(10) +
      " y:" + Round(CommandInfo(CMD_INFO_Y), 0.1)
End Sub
```

See Also:

CommandInfo( ) function

TriggerControl( ) function

Purpose

Returns the ID of the last dialog control chosen by the user. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

TriggerControl( )

Return Value

Integer

Description

Within a **Dialog statement**'s handler procedure, the **TriggerControl( )** function returns the control ID of the last control which the user operated.

Each control in a dialog box can have its own dedicated handler procedure; alternately, one procedure can act as the handler for two or more controls. A procedure which handles multiple controls can use the **TriggerControl( )** function to detect which control the user clicked.

Error Conditions

ERR_INVALID_TRIG_CONTROL (843) error is generated if the **TriggerControl( )** function is called when no dialog box is active.

See Also:

Alter Control statement, **Dialog statement**, **Dialog Preserve statement**, **Dialog Remove statement**, **ReadControlValue( ) function**

TrueFileName$( ) function

Purpose

Returns a full file specification, given a partial specification. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

TrueFileName$(*file_spec*)

file_spec is a string representing a partial file specification (for example, "C:PARCELS.TAB")

Description

This function returns a full file specification (including full drive name and full directory name), given a partial specification.

In some circumstances, you may need to process a partial file specification. For example, on a DOS system, the following file specification is partial (it includes a drive letter, C:, but it omits the current directory name):

```
"C:parcels.tab"
```

If the current directory on drive C: is "\mapinfo\data" then the following function call:

```
TrueFileName$("C:parcels.tab")
```

returns the string:

```
"C:\mapinfo\data\parcels.tab"
```

If your application prompts the user to type in the name of a hard drive or file path, you may want to use **TrueFileName$( )** to expand the path entered by the user into a full path.

The **TrueFileName$( )** function does not verify the existence of the named file; it merely expands the partial drive letter and directory path. To determine whether a file exists, use the **FileExists( ) function**.

See Also:

ProgramDirectory$( ) function

Type statement

Purpose

Defines a custom variable type which can be used in later **Dim statement**s and **Global statement**s.

Syntax

```
Type type_name
   element_name As var_type
   [ ... ]
End Type
```

type_name is the name you define for the data type.

element_name is the name you define for each element of the type.

var_type is the data type of that element.

Restrictions

Any **Type** statements must appear at the "global" level in a program file (for example, outside of any sub procedure). You cannot issue a **Type** statement through the MapBasic window. You cannot pass a **Type** variable as a by-value parameter to a procedure or function. You cannot write a **Type** variable to a file using a **Put statement**.

Description

The **Type** statement creates a new data type composed of elements of existing data types. You can address each element of a variable of a custom type using an expression structured as *variable_name.element_name*. A **Type** can contain elements of other custom types and elements which are arrays. You can also declare arrays of variables of a custom Type. You cannot copy the entire contents of a **Type** variable to another **Type** variable using an assignment of the form *var_name = var_name*.

Example

```
Type Person
   fullname As String
   age As Integer
   dateofbirth As Date
End Type

Dim sales_mgr, sales_people(10) As Person
```

```
sales_mgr.fullname = "Otto Carto"
sales_people(1).fullname = "Melinda Robertson"
```

See Also:

Dim statement, **Global statement**, **ReDim statement**

UBound( ) function

Purpose

Returns the current size of an array. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

UBound(*array*)

array is the name of an array variable.

Return Value

Integer

Description

The **UBound( )** function returns an integer value indicating the current size (or “upper bound”) of an array variable.

Every array variable has an initial size, which can be zero or larger. This initial size is specified in the variable's **Dim statement** or **Global statement**. However, an array's size can be reset through the **ReDim statement**. The **UBound( )** function returns an array's current size, as an integer value indicating how many elements can currently be stored in the array. A MapBasic array can have up to 32,767 items.

Example

```
Dim matrix(10) As Float
Dim depth As Integer

depth = UBound(matrix)
' depth now has a value of 10

ReDim matrix(20)
depth = UBound(matrix)
' depth now has a value of 20
```

See Also:

Dim statement, **Global statement**, **ReDim statement**

UCase$( ) function

Purpose

Returns a string, converted to upper-case. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

UCase$(*string_expr*)

string_expr is a string expression.

Return Value

String

Description

The **UCase$( )** function returns the string which is the upper-case equivalent of the string expression *string_expr*.

Conversion from lower to upper case only affects alphabetic characters (A through Z); numeric digits and punctuation marks are not affected. Thus, the function call `UCase$("A#12a")` returns the string value “A#12A”.

Example

```
Dim regular, upper_case As String

regular = "Los Angeles"
upper_case = UCase$(regular)
' upper_case now contains the value "LOS ANGELES"
```

See Also:

LCase$( ) function, **Proper$( ) function**

UnDim statement

Purpose

Undefines a variable. You can issue this statement in the MapBasic Window in MapInfo Professional.

Syntax

UnDim *variable_name*

variable_name is the name of a variable that was declared through the MapBasic window or through a workspace.

Restrictions

The **UnDim** statement cannot be used in a compiled MapBasic program; it may only be used within a workspace or entered through the MapBasic window.

Description

After you use the **Dim statement** to create a variable, you can use the **UnDim** statement to destroy that variable definition. For example, suppose you type a **Dim statement** into the MapBasic window to declare the variable X:

```
Dim X As Integer
```

Now suppose you want to redefine X to be a Float. The following statements redefine X:

```
UnDim X
Dim X As Float
```

See Also:

Dim statement, **ReDim statement**

UnitAbbr$( ) function

Purpose

Returns a string representing the abbreviated version of a standard MapInfo Professional unit name. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

UnitAbbr$(*unit_name*)

unit_name is a string representing a standard MapInfo Professional unit name (for example, “km”).

Return Value

String expression, representing an abbreviated unit name (for example, “km”)

Description

The *unit_name* parameter must be one of MapInfo Professional's standard, English-language unit names, such as “km” (for kilometers) or “sq km” (for square kilometers).

The **UnitAbbr$( )** function returns an abbreviated version of the unit name. The exact string returned depends on whether the user is running the English-language version of MapInfo Professional or a translated version. For example, if a user is running the German-language version of MapInfo Professional, the following function call returns the German translation of “sq km”:

```
UnitAbbr$("sq km")
```

The **UnitAbbr$( )** function can operate on units of distance, area, paper, and time. For a listing of MapInfo Professional's standard distance unit names (for example, “km”), see **Set Distance Units statement**. For a listing of area unit names (for example, “sq km”), see **Set Area Units statement**. For a listing of paper unit names (for example, “in” for inches on a page layout), see **Set Paper Units statement**. Time unit names include seconds (“sec”), minutes (“min”), and hours (“hr”).

The *unit_name* parameter can also be “degree” (in which case, **UnitAbbr$( )** returns “deg”).

See Also:

Set Area Units statement, **Set Distance Units statement**, **Set Paper Units statement**, **UnitName$( ) function**

UnitName$( ) function

Purpose

Returns a string representing the full version of a standard MapInfo Professional unit name. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

UnitName$(*unit_name*)

unit_name is a string representing a standard MapInfo Professional unit name (for example, “km”)

Return Value

String expression, representing a full unit name (for example, “kilometers”)

Description

The *unit_name* parameter must be one of MapInfo Professional's standard, English-language unit names, such as “km” (for kilometers) or “sq km” (for square kilometers).

The **UnitName$( )** function returns a string representing the full version of the unit name. The exact string returned depends on whether the user is running the English-language version of MapInfo Professional or a translated version. For example, if a user is running the French-language version of MapInfo Professional, the following function call returns the French translation of “square kilometers”:

```
UnitName$("sq km")
```

The **UnitName$( )** function can operate on units of distance, area, paper, and time. For a listing of MapInfo Professional's standard distance unit names (for example, “km”), see **Set Distance Units statement**. For a listing of area unit names (for example, “sq km”), see **Set Area Units statement**. For a listing of paper unit names (for example, “in” for inches on a page layout), see **Set Paper Units statement**. Time unit names include seconds (“sec”), minutes (“min”), and hours (“hr”).

The *unit_name* parameter can also be “degree” (in which case, **UnitName$( )** returns “degrees”).

See Also:

Set Area Units statement, **Set Distance Units statement**, **Set Paper Units statement**, **UnitAbbr$( ) function**

Unlink statement

Purpose

Unlinks a table which was downloaded and linked from a remote database with the **Server Link Table statement**. You can issue this statement in the MapBasic Window in MapInfo Professional.

Syntax

```
Unlink TableName
```

TableName is the name of an open MapInfo linked table.

Description

Unlinking a table removes the link to the remote database. This statement doesn't work if edits are pending (in other words, the user must first commit or rollback). All metadata associated with the table linkage is removed. Fields that were marked non-editable are now editable. The end product is a normal MapInfo base table.

Example

```
Unlink "City_1k"
```

See Also:

Commit Table statement, **Server Link Table statement**

Update statement

Purpose

Modifies one or more rows in a table. You can issue this statement in the MapBasic Window in MapInfo Professional.

Syntax

```
Update table Set column = expr [, column = expr, ...]
   [ Where RowID = idnum ]
```

table is the name of an open table.

column is the name of a column.

expr is an expression to assign to a column.

idnum is the number of a row in the table.

Description

The **Update** statement modifies one or more columns in a table. By default, the **Update** statement will affect all rows in the specified table. However, if the statement includes a **Where Rowid** clause, only one particular row will be updated. The **Set** clause specifies what sort of changes should be made to the affected row or rows.

To update the map object that is attached to a row, specify the column name Obj in the **Set** clause; see example below.

Examples

In the following example, we have a table of employee data; each record states the employee's department and salary. Let's say we wish to give a seven percent raise to all employees of the marketing department currently earning less than $20,000. The example below uses a **Select statement** to select the appropriate employee records, and then uses an **Update** statement to modify the salary column accordingly.

```
Select * From employees
   Where department ="marketing" And salary < 20000
Update Selection
   Set salary = salary * 1.07
```

By using a **Where RowID** clause, you can tell MapBasic to only apply the **Set** operation to one particular row of the table. The following example updates the salary column of the tenth record in the employees table:

```
Update employees
   Set salary = salary * 1.07
   Where Rowid = 10
```

The next example stores a point object in the first row of a table:

```
Update sites
   Set Obj = CreatePoint(x, y)
   Where Rowid = 1
```

See Also:

Insert statement

Update Window statement

Purpose

Forces MapInfo Professional to process all pending changes to a window. You can issue this statement in the MapBasic Window in MapInfo Professional.

Syntax

`Update Window` *`window_id`*

window_id is an integer window identifier.

Description

The **Update Window** statement forces MapInfo Professional to process any pending window display changes.

Under some circumstances, window operations performed by a MapBasic application do not appear immediately. For example, if an application issues a **Dialog statement** immediately after modifying a Map window, the changes to the Map window may not appear until after the user dismisses the dialog box. To force MapInfo Professional to process pending display changes, use the **Update Window** statement.

See Also:

Set Event Processing statement

URL clause

Purpose

Specifies the library service URL. You can use this clause in the MapBasic Window in MapInfo Professional.

Syntax

URL *url*

url is a valid Library Service URL.

Description

The **URL** clause specifies the default Library Service URL to use. It checks that the input is a valid Library Service URL, and displays an error message if it is not valid.

The default Library service URL is set to an empty string “” to indicate that the Library Service is not currently set. Once set to a valid URL, you can reset the Library Service URL to an empty string to reset it.

Example

```
Include "MAPBASIC.DEF"
Set LibraryServiceInfo URL
http://localhost:8080/LibraryService/LibraryService
```

See Also:

Set LibraryServiceInfo statement, **LibraryServiceInfo( ) function**

USNGToPoint(string)

Purpose

Converts a string representing an USNG (United States National Grid) coordinate into a point object in the current MapBasic coordinate system. You can call this function from the MapBasic Window in MapInfo Professional.

Syntax

```
USNGToPoint(string)
```

string is a string expression representing a USNG grid reference.

Return Value

Object.

Description

The returned point will be in the current MapBasic coordinate system, which by default is Long/Lat (no datum). For the most accurate results when saving the resulting points to a table, set the MapBasic coordinate system to match the destination table's coordinate system before calling **USNGToPoint( )**. This will prevent MapInfo Professional from doing an intermediate conversion to the datumless Long/Lat coordinate system, which can cause a significant loss of precision.

Example 1

```
dim obj1 as Object
dim s_USNG As String
dim obj2 as Object
obj1 = CreatePoint(-74.669, 43.263)
s_USNG = PointToUSNG$(obj1)
obj2 = USNGToPoint(s_USNG)
```

Example 2

```
Open Table "C:\Temp\MyTable.TAB" as USNGfile
' When using the PointToUSNG$( ) or USNGToPoint( ) functions,
' it is very important to make sure that the current MapBasic
' coordsys matches the coordsys of the table where the
' point object is being stored.
'Set the MapBasic coordsys to that of the table used
Set CoordSys Table USNGfile
'Update a Character column (for example COL2) with USNG strings from
'a table of points
Update USNGfile
   Set Col2 = PointToUSNG$(obj)
'Update two float columns (Col3 & Col4) with
'CentroidX & CentroidY information
'from a character column (Col2) that contains USNG strings.
```

```
Update USNGfile
   Set Col3 = CentroidX(USNGToPoint(Col2))
Update USNGtestfile ' USNGfile
   Set Col4 = CentroidY(USNGToPoint(Col2))
Commit Table USNGfile
Close Table USNGfile
```

See Also:

PointToUSNG$(obj, datumid)

Val( ) function

Val( ) function

Purpose

Returns the numeric value represented by a string. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

Val(*string_expr*)

string_expr is a string expression.

Return Value

Float

Description

The **Val( )** function returns a number based on the *string_expr* string expression. **Val( )** ignores any white spaces (tabs, spaces, line feeds) at the start of the *string_expr* string, then tries to interpret the first character(s) as a numeric value. The **Val( )** function then stops processing the string as soon as it finds a character that is not part of the number. If the first non-white-space character in the string is not a period, a digit, a minus sign, or an ampersand character (&), **Val( )** returns zero. (The ampersand is used in hexadecimal notation; see example below.)

If the string includes a decimal separator, it must be a period, regardless of whether the user's computer is set up to use some other character as the decimal separator. Also, the string cannot contain thousands separators. To remove thousands separators from a numeric string, call the **DeformatNumber$( ) function**.

Example

```
Dim f_num As Float
f_num = Val("12 thousand")
' f_num is now equal to 12
```

```
f_num = Val("12,345")
' f_num is now equal to 12

f_num = Val(" 52 - 62 Brunswick Ave")
' f_num is now equal to 52

f_num = Val("Eighteen")
' f_num is now equal to 0 (zero)

f_num = Val("&H1A")
' f_num is now equal to 26 (which equals hexadecimal 1A)
```

See Also:

DeformatNumber$( ) function, **Format$( ) function**, **Set Format statement**, **Str$( ) function**

Weekday( ) function

Purpose

Returns an integer from 1 to 7, indicating the weekday of a specified date. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

Weekday(*date_expr*)

date_expr is a date expression.

Return Value

SmallInt value from 1 to 7, inclusive; 1 represents Sunday.

Description

The **Weekday( )** function returns an integer representing the day-of-the-week component (one to seven) of the specified date.

The **Weekday( )** function only works for dates on or after January 1, in the year 100. If *date_expr* specifies a date before the year 100, the **Weekday( )** function returns a value of zero.

Example

```
If Weekday( CurDate( ) ) = 6 Then
   '
   ' then the date is a Friday
   '
End If
```

See Also:

CurDate( ) function, **Day( ) function**, **Month( ) function**, **Year( ) function**

WFS Refresh Table statement

Purpose

Refreshes a WFS table from the server. You can issue this statement in the MapBasic Window in MapInfo Professional.

Syntax

`WFS Refresh Table` *`alias`* `[ using map [ window id ] ]`

alias is the an alias for an open registered WFS table.

If the table was created with a row filter where the geometry of the wfs table is within the current mapper (the row filter operation will be ogc:BBOX and the value is CURRENT_MAPPER), then the only data in the table will be what is inside the mapper's bounds. Refreshing the table will use the old mapper bounds and ignore any zoom or pan changes made since unless the optional using map is used. In this case, the bounds of the current mapper will be used and any zoom and pan operations that have occurred will be taken into account. If the window is not provided, then the topmost map window is used for the bounds. Otherwise the bounds of the map window specified by the window id will be used.

Specifying a row filter using the mapper bounds can speed up the initial display of the WFS table, since it restricts the amount of data being transferred from the server.

Example

The following example refreshes the local table named watershed.

```
WFS Refresh Table watershed
```

If the WFS table was created with a row filter of the bounds of the mapper, and the mapper has been panned, then the parts of the wfs table may not be displayed. To update the table so that it displays everything in the current mapper, the following can be used.

```
WFS Refresh Table watershed Using Map
```

See Also:

Register Table statement, **TableInfo( ) function**

While…Wend statement

Purpose

Defines a loop which executes as long as a specified condition evaluates as TRUE.

Syntax

```
While condition
    statement_list
Wend
```

condition is a conditional expression which controls when the loop should stop.

statement_list is the group of statements to execute with each iteration of the loop.

Restrictions

You cannot issue a **While…Wend** statement through the MapBasic window.

Description

The **While…Wend** statement provides loop control. MapBasic evaluates the condition; if it is TRUE, MapBasic will execute the *statement_list* (and then evaluate the condition again, etc.).

As long as the condition remains TRUE, MapBasic will repeatedly execute the *statement_list*. When and if the condition becomes FALSE, MapBasic will skip the *statement_list*, and continue execution with the first statement following the **Wend** keyword.

Note that a statement of this form:

```
While condition
    statement_list
Wend
```

is functionally identical to a statement of this form:

```
Do While condition
    statement_list
Loop
```

The **While…Wend** syntax is provided for stylistic reasons (for example, for the sake of those programmers who prefer the **While…Wend** syntax over the **Do…Loop statement** syntax).

Example

```
Dim psum As Float, i As Integer
Open Table "world"
Fetch First From world
i = 1
While i <= 10
   psum = psum + world.population
   Fetch Next From world
   i = i + 1
Wend
```

See Also:

Do…Loop statement, **For…Next statement**

WinChangedHandler procedure

Purpose

A reserved procedure, called automatically when a Map window is panned or zoomed, or whenever a map layer is added or removed.

Syntax

```
Declare Sub WinChangedHandler
Sub WinChangedHandler
    statement_list
End Sub
```

statement_list is a list of statements to execute when the map is panned or zoomed.

Description

`WinChangedHandler` is a special-purpose MapBasic procedure name. If the user runs an application containing a procedure named `WinChangedHandler`, the application “goes to sleep” when the Main procedure runs out of statements to execute. As long as the sleeping application remains in memory, MapBasic calls `WinChangedHandler` whenever a Map window's extents are modified (for example, the Map is scrolled, zoomed or re-sized). Within the `WinChangedHandler` procedure, call the **CommandInfo( ) function** to determine the integer window ID of the affected window.

Multiple MapBasic applications can be “sleeping” at the same time. When a Map window changes, MapBasic automatically calls all sleeping **WinChangedHandler** procedures, one after another.

Under some circumstances, MapBasic may call a **WinChangedHandler** procedure as a result of an event which did not affect the map extents. For example, drawing a new object may trigger the **WinChangedHandler** procedure. To halt a sleeping application and remove it from memory, use the **End Program statement**.

Auto-scrolling Map Windows

MapInfo Professional automatically scrolls the Map window if the user clicks with the mouse and then drags to the edge of the window. If the user auto-scrolls a Map window, MapInfo Professional calls **WinChangedHandler** after the tool action is completed or canceled.

For example, if you use MapInfo Professional's Ruler tool and you autoscroll the window during each segment, MapInfo Professional calls **WinChangedHandler** once, after you double-click to complete the measurement (or after you press Esc to cancel the Ruler tool). If the user auto-scrolls while using a custom MapBasic tool, MapInfo Professional calls the tool's handler procedure, and then calls **WinChangedHandler**.

MapInfo Professional will not call **WinChangedHandler** if the user auto-scrolls but then returns to the original location before completing the operation or pressing Esc.

To disable the autoscroll feature, use the **Set Window statement**.

Example

For an example of using a **WinChangedHandler** procedure, see the OverView sample program.

See Also:

CommandInfo( ) function, **WinClosedHandler procedure**

WinClosedHandler procedure

Purpose

A reserved procedure, called automatically when a Map, Browse, Graph, Layout, Redistricting, or MapBasic window is closed.

Syntax

```
Declare Sub WinClosedHandler

Sub WinClosedHandler
    statement_list
End Sub
```

statement_list is a list of statements to execute when a window is closed.

Description

WinClosedHandler is a special-purpose MapBasic sub procedure name. If the user runs an application containing a procedure named **WinClosedHandler**, the application "goes to sleep" when the Main procedure runs out of statements to execute. As long as the sleeping application remains in memory, MapBasic automatically calls the **WinClosedHandler** procedure whenever a window is closed.

Within the **WinClosedHandler** procedure, you can use issue the function call:

```
CommandInfo( CMD_INFO_WIN )
```

to determine the window identifier of the closed window.

When any procedure in an application executes the **End Program statement**, the application is completely removed from memory. Thus, you can use the **End Program statement** to terminate a **WinClosedHandler** procedure once it is no longer wanted. Conversely, you should be careful not to issue an **End Program statement** while the **WinClosedHandler** procedure is still needed.

Multiple MapBasic applications can be "sleeping" at the same time. When a window is closed, MapBasic automatically calls all sleeping **WinClosedHandler** procedures, one after another.

See Also:

CommandInfo( ) function, **EndHandler procedure**, **RemoteMsgHandler procedure**, **SelChangedHandler procedure**, **ToolHandler procedure**, **WinChangedHandler procedure**

WindowID( ) function

Purpose

Returns a MapInfo Professional window identifier. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

WindowID(*window_num*)

window_num is a number or a numeric code; see table below.

Return Value

Integer

Description

A window identifier is an integer value which uniquely identifies an existing window. Several MapBasic statements (for example, the **Set Map statement**) take window identifiers as parameters.

The following table lists the various ways that you can specify the *window_num* parameter:

Value of window_num	Result
Positive SmallInt value (1, 2, … *n*)	MapInfo Professional returns the window ID of a document window, such as a Map or Browse window. For example, if you specify 1, MapInfo Professional returns the integer ID of the first document window. Note that *n* is the number of open document windows; call the **NumWindows( ) function** to determine *n*.
Negative SmallInt value (-1,-2, …-*m*)	MapInfo Professional returns the window ID of a window, which may be a document window or a floating window such as the Info window. Note that *m* is the total number of windows owned by MapInfo Professional; call the **NumAllWindows( ) function** to determine *m*. Using this syntax, you could call **WindowID( )** within a loop to build a list of the ID numbers of all open windows.

Value of window_num	Result
Zero (0)	MapInfo Professional returns the window ID of the most recently opened document window, custom Legend window, or ButtonPad; returns zero if no windows are open.
Window code (for example, WIN_RULER)	If you specify a window code with a value from 1001 to 1013, MapInfo Professional returns the ID of a special window. Window codes are defined in MAPBASIC.DEF. For example, the code WIN_RULER (with a value of 1007) represents the window used by MapInfo Professional's Ruler tool.

Error Conditions

ERR_BAD_WINDOW_NUM (648) error is generated if the *window_num* parameter is invalid.

See Also:

FrontWindow( ) function, **NumWindows( ) function**

WindowInfo( ) function

Purpose

Returns information about a window. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

WindowInfo(*window_spec*, *attribute*)

window_spec is a number or a code that specifies which window you want to query.

attribute is an integer code indicating which information about the window to return.

Return Value

Depends on the *attribute* parameter.

Description

The **WindowInfo( )** function returns one piece of information about an existing window.

Many of the values that you pass as the parameters to **WindowInfo( )** are defined in the standard MapBasic definitions file, MAPBASIC.DEF. Your program should Include “MAPBASIC.DEF” if you are going to call **WindowInfo( )**.

The following table lists the various ways that you can specify the *window_spec* parameter:

Value of window_spec	Description
Integer window ID	You can use an integer window ID (which you can obtain by calling the **WindowID( ) function** or the **FrontWindow( ) function**) to specify which window you want to query.
Positive SmallInt value (1, 2, … *n*)	The function queries a document window, such as a Map or Browser window. For example, specify 1 to retrieve information on the first document window. Note that *n* is the number of open document windows; call the **NumWindows( ) function** to determine *n*.
Negative SmallInt value (-1,-2, …-*m*)	The function queries a window, which may be a document window or a floating window such as the Info window. Note that *m* is the total number of windows owned by MapInfo Professional; call the **NumAllWindows( ) function** to determine *m*. Using this syntax, you could call **WindowInfo( )** within a loop to query every open window.
Zero (0)	The function queries the most recently-opened window. If no windows are open, an error occurs.
Window code (for example, WIN_RULER)	If you specify a window code with a value from 1001 to 1013, the function queries a special system window. Window codes are defined in MAPBASIC.DEF. For example, MAPBASIC.DEF contains the code WIN_RULER (with a value of 1007), which represents the window used by MapInfo Professional's Ruler tool.

The attribute parameter dictates which window attribute the function should return. The attribute parameter must be one of the codes from the table below:

attribute code	ID	WindowInfo(attribute) returns:
WIN_INFO_AUTOSCROLL	17	Logical value: TRUE if the autoscroll feature is on for this window, allowing the user to scroll the window by dragging to the window's edge. To turn autoscroll on or off, use the **Set Window statement**.
WIN_INFO_CLONEWINDOW	15	String value: a string of MapBasic statements that can be used in a **Run Command statement** to duplicate a window.
WIN_INFO_HEIGHT	5	Float value: window height (in paper units).
WIN_INFO_LEGENDS_MAP	10	Integer value: when you query a Legend window created using the **Create Legend statement**, this code returns the integer window ID of the Map or Graph window that owns the legend. When you query the standard Legend window, returns 0.
WIN_INFO_ADORNMENTS_MAP	—	Overloaded with the same value as WIN_INFO_LEGENDS_MAP (10). If the WindowID is a Adornment, this will return the WindowID of the Mapper used to create the Adornment.
WIN_INFO_NAME	1	String value: the name of the window.
WIN_INFO_OPEN	11	Logical value: TRUE if the window is open (used with special windows such as the Info window).
WIN_INFO_SMARTPAN	18	Logical value; TRUE if **Smart Pan** has been set on.
WIN_INFO_STATE	9	SmallInt value: WIN_STATE_NORMAL if at normal size, WIN_STATE_MINIMIZED (1) if minimized, WIN_STATE_MAXIMIZED (2) if maximized.
WIN_INFO_SYSMENUCLOSE	16	Logical value: FALSE indicates that a **Set Window statement** has disabled the **Close** command on the window's system menu.

attribute code	ID	WindowInfo(attribute) returns:
WIN_INFO_TABLE	10	String value: For Map windows, the name of the window's “CosmeticN” table. For Layout windows, the name of the window's “LayoutN” table. For Browser or Graph windows, the name of the table displayed in the window.
WIN_INFO_TOPMOST	8	Logical value: TRUE if this is the active window.
WIN_INFO_TYPE	3	SmallInt value: window type, such as WIN_LAYOUT (3). See table below.
WIN_INFO_WIDTH	4	Float value: window width (in paper units).
WIN_INFO_WINDOWID	13	Integer value, representing the window's ID; identical to the value returned by the **WindowID( ) function**. This is useful if you pass zero as the *window_spec*.
WIN_INFO_WND	12	Integer value. On Windows, the value represents a Windows *HWND* for the window you are querying.
WIN_INFO_WORKSPACE	14	String value: the string of MapBasic statements that a Save Workspace operation would write to a workspace to record the settings for this map. Differs from WIN_INFO_CLONEWINDOW (15) in that the results include **Open Table statement**, etc.
WIN_INFO_X	6	Float value: the window's distance from the left edge of the MapInfo Professional work area (in paper units).
WIN_INFO_Y	7	Float value: the window's distance from the top edge of the MapInfo Professional work area (in paper units).
WIN_INFO_PRINTER_NAME	21	Returns string value with printer identifier (for example, \\DISCOVERY\HP4_DEVEL)
WIN_INFO_PRINTER_ORIENT	22	Returns WIN_PRINTER_PORTRAIT (1) or WIN_PRINTER_LANDSCAPE (2)
WIN_INFO_PRINTER_COPIES	23	Returns integer number of copies.
WIN_INFO_SNAPMODE	19	Returns a logical value. TRUE if snap mode is on. FALSE if snap mode is off.

attribute code	ID	WindowInfo(attribute) returns:
WIN_INFO_SNAPTHRESHOLD	20	Returns a SmallInt value representing the pixel tolerance.
WIN_INFO_PRINTER_PAPERSIZE	24	Integer value. Refer to the PAPERSIZE.DEF file (In the \MapInfo\MapBasic directory) for the meaning of the return value.
WIN_INFO_PRINTER_LEFTMARGIN	25	Float value: left printer margin value in current units.
WIN_INFO_PRINTER_RIGHTMARGIN	26	Float value: right printer margin value in current units.
WIN_INFO_PRINTER_TOPMARGIN	27	Float value: top margin value in current units.
WIN_INFO_PRINTER_BOTTOMMARGIN	28	Float value: bottom printer margin value in current units.
WIN_INFO_PRINTER_BORDER (29)	29	String value: ON if a black border will be on the printer output, OFF otherwise.
WIN_INFO_PRINTER_TRUECOLOR	30	String value: ON if use 24-bit true color to print raster and grid images. This is possible when the image is 24 bit and the printer supports more than 256 colors, OFF otherwise.
WIN_INFO_PRINTER_DITHER	31	String value: return dithering method, which is used when it is necessary to convert a 24-bit image to 256 colors. Possible return values are HALFTONE and ERRORDIFFUSION. This option is used when printing raster and grid images. Dithering will occur if WIN_INFO_PRINTER_TRUECOLOR (30) is disabled or if the printer color depth is 256 colors or less.
WIN_INFO_PRINTER_METHOD	32	String value: possible return values are DEVICE, EMF, or PRINTOSBM.
WIN_INFO_PRINTER_TRANSPRASTER	33	String value: possible return values are DEVICE and INTERNAL.
WIN_INFO_PRINTER_TRANSPVECTOR	34	String value: possible return values are DEVICE and INTERNAL.
WIN_INFO_EXPORT_BORDER	35	String value: possible return values are ON and OFF.

attribute code	ID	WindowInfo(attribute) returns:
WIN_INFO_EXPORT_TRUECOLOR	36	String value: possible return values are ON and OFF.
WIN_INFO_EXPORT_DITHER	37	String value: possible return values are HALFTONE and ERRORDIFFUSION.
WIN_INFO_EXPORT_TRANSPRASTER	38	String value: possible return values are DEVICE and INTERNAL.
WIN_INFO_EXPORT_TRANSPVECTOR	39	String value: possible return values are DEVICE and INTERNAL.
WIN_INFO_PRINTER_SCALE_PATTERNS	40	Logical value. TRUE if window is scaled on printer output. FALSE if not scaled.
WIN_INFO_EXPORT_ANTIALIASING	41	String value: ON if a anti-aliasing filter will be used for exporting, OFF otherwise.
WIN_INFO_EXPORT_THRESHOLD	42	Integer value between 0 and 255 that specifies anti-aliasing threshold.
WIN_INFO_EXPORT_MASKSIZE	43	Integer value between 0 and 100
WIN_INFO_EXPORT_FILTER	44	Integer value that return one of possible anti-aliasing filters: • FILTER_VERTICALLY_AND_HORIZONTALLY (0) • FILTER_ALL_DIRECTIONS_1 (1) • FILTER_ALL_DIRECTIONS_2 (2) • FILTER_DIAGONALLY (3) • FILTER_HORIZONTALLY (4) • FILTER_VERTICALLY (5)
WIN_INFO_ENHANCED_RENDERING	45	Logical value: TRUE if enhanced rendering is on for this window. To turn enhanced rendering on or off, use the Set Window statement.
WIN_INFO_SMOOTH_TEXT	46	String value: The string representation of the current smooth text mode for the window. To change the smooth mode, use the Set Window statement.

attribute code	ID	WindowInfo(attribute) returns:
WIN_INFO_SMOOTH_IMAGE	47	String value: The string representation of the current smooth image mode for the window. To change the smooth mode, use the Set Window statement.
WIN_INFO_SMOOTH_VECTOR	48	String value: The string representation of the current smooth vector mode for the window. To change the smooth mode, use the Set Window statement.

If you specify WIN_INFO_TYPE as the *attribute*, **WindowInfo( )** returns one of these values:

Window type	ID	Window description
WIN_MAPPER	1	Map window
WIN_BROWSER	2	Browse window
WIN_LAYOUT	3	Layout window
WIN_GRAPH	4	Graph window
WIN_BUTTONPAD	19	A ButtonPad window
WIN_TOOLBAR	25	The Toolbar window
WIN_CART_LEGEND	27	The Cartographic Legend window
WIN_3DMAP	28	The 3D Map window
WIN_ADORNMENT	32	A Adornment window
WIN_HELP	1001	The Help window
WIN_MAPBASIC	1002	The MapBasic window
WIN_MESSAGE	1003	The Message window (used with the **Print statement**)
WIN_RULER	1007	The Ruler window (displays the distances measured by the Ruler tool)
WIN_INFO	1008	The Info window (displays data when the user clicks with the Info tool)
WIN_LEGEND	1009	The Theme Legend window
WIN_STATISTICS	1010	The Statistics window
WIN_MAPINFO	1011	The MapInfo Professional application window

Each Map window has a special, temporary table, which represents the “cosmetic layer” for that map. These tables (which have names like “Cosmetic1”, “Cosmetic2”, etc.) are invisible to the MapInfo Professional user. To obtain the name of a Cosmetic table, specify WIN_INFO_TABLE (10). Similarly, you can obtain the name of a Layout window's temporary table (for example, “Layout1”) by calling **WindowInfo( )** with the WIN_INFO_TABLE (10) attribute.

Error Conditions

ERR_BAD_WINDOW (590) error is generated if the *window_id* parameter is invalid.

ERR_FCN_ARG_RANGE (644) error is generated if an argument is outside of the valid range.

Example

The following example opens the Statistics window if it isn't open already.

```
If Not WindowInfo(WIN_STATISTICS,WIN_INFO_OPEN) Then
   Open Window WIN_STATISTICS
End If
```

See Also:

Browse statement, **Graph statement**, **Map statement**

WinFocusChangedHandler procedure

Purpose

A reserved procedure name, called automatically when the window focus changes.

Syntax

```
Declare Sub WinFocusChangedHandler

Sub WinFocusChangedHandler
    statement_list
End Sub
```

Description

If a MapBasic application contains a sub procedure called **WinFocusChangedHandler**, MapInfo Professional calls the sub procedure automatically, whenever the window focus changes. This behavior applies to all MapInfo Professional window types (Browsers, Maps, etc.). Within the **WinFocusChangedHandler** procedure, you can obtain the integer window ID of the current window by calling CommandInfo(CMD_INFO_WIN).

The **WinFocusChangedHandler** procedure should not use the **Note statement** and should not open or close any windows. These restrictions are similar to those for other handlers, such as the **SelChangedHandler procedure**.

The **WinFocusChangedHandler** procedure should be as short as possible, to avoid slowing system performance.

Example

The following example shows how to enable or disable a menu item, depending on whether the active window is a Map window.

```
Include "mapbasic.def"
Include "menu.def"
Declare Sub Main
Declare sub WinFocusChangedHandler
Sub Main
   ' At this point, we could create a custom menu item
   ' which should only be enabled if the current window
   ' is a Map window...
End Sub

Sub WinFocusChangedHandler
   Dim i_win_type As SmallInt

   i_win_type=WindowInfo(CommandInfo(CMD_INFO_WIN),WIN_INFO_TYPE)

   If i_win_type = WIN_MAPPER Then
      ' here, we could enable a map-related menu item
   Else
      ' here, we could disable a map-related menu item
   End If
End Sub
```

See Also:

WinChangedHandler procedure

Write # statement

Purpose

Writes data to an open file.

Syntax

Write # *file_num* [, *expr* ...]

file_num is the number of an open file.

expr is an expression to write to the file.

Description

The **Write #** statement writes data to an open file. The file must have been opened in a sequential mode which allows modification of the file (Output or Append).

The *file_num* parameter corresponds to the number specified in the **As** clause of the **Open File statement**.

If the statement includes a comma-separated list of expressions, MapInfo Professional automatically inserts commas into the file to separate the items. If the statement does not include any expressions, MapInfo Professional writes a blank line to the file.

The **Write #** statement automatically encloses string expressions in quotation marks within the file. To write text to a file without quotation marks, use the **Print # statement**.

Use the **Input # statement** to read files that were created using **Write #**.

See Also:

Input # statement, **Open File statement**, **Print # statement**

Year( ) function

Purpose

Returns the year component of a date value. You can call this function from the MapBasic window in MapInfo Professional.

Syntax

Year(*date_expr*)

date_expr is a date expression.

Return Value

SmallInt

Description

If the **Set Date Window statement** is off, then the year also depends on your system clock.

Examples

The following example shows how you can use the **Year( )** function to extract only the year component of a particular date value.

```
Dim sampleDate as Date
Set Date Window Off
sampleDate=StringToDate("10/1/98")
Print Year(sampleDate)
' 2098 (or 1998 if the computer's system date is set in the 1900's)
' because with date windowing off MapInfo uses the current century
Set Date Window 50
' now assume that two-digit dates fall in the period 1950-2049
print Year(sampleDate)
' still 2098, because date variable has already been assigned!
sampleDate=StringToDate("10/1/98")
' re-assign variable now that the date window has changed
print Year(sampleDate) ' 1998
Undim sampleDate
```

The **Year( )** function can also take a string, rather than a Date variable. In that case, implicit conversion to date format occurs. The following example illustrates this:

```
Set Date Window Off
Print Year("10/1/99") ' prints 2099
Set Date Window 50
Print Year("10/1/99") ' prints 1999
```

You can also use the **Year( )** function within the SQL Select statement. The following Select statement selects only particular rows from the Orders table. This example assumes that the Orders table has a Date column, called OrderDate. The Select statement's Where clause tells MapInfo Professional to only select the orders from December of 1993.

```
Open Table "orders"
Select * From orders
   Where Month(orderdate) = 12 And Year(orderdate) = 1993
```

See Also:

CurDate( ) function, **Day( ) function**, **DateWindow( ) function**, **Month( ) function**, **Weekday( ) function**

A

HTTP and FTP Libraries

This appendix details the HTTP and FTP libraries that enable MapBasic programmers to use web-based technology. These libraries allow access to RSS feeds and other web-based location information such as weather information, traffic feeds, vehicle locations, etc., as well as the ability to set up FTP connections and search, receive, and send files through a MapBasic program. This library uses the common DEF files: HTTPLib.DEF, HTTPType.DEF, and HTTPUtil.DEF, which are installed in <*Your MapBasic Installation Directory*>\Samples\MapBasic\INC. Make sure you include these files as header files into your programs. All the functionality described in this appendix is also dependent on the presence of GmlXlat.dll which is installed with MapInfo Professional.

We have provided sample applications that demonstrate the use of these libraries. See *Your MapBasic Installation Directory*\Samples\MapBasic\HTTPLib and *Your MapBasic Installation Directory*\Samples\MapBasic\FTPLib for the specific samples.

All of the functions and procedures listed in this appendix are wrappers of the corresponding methods of Microsoft MFC Classes. The wrapped classes include CInternetSession, CHttpConnection, CFtpConnection, CHttpFile, and CFtpFileFind. For more detailed information about the usage of the related classes refer to the MSDN reference for MFC classes (**http://msdn2.microsoft.com/en-us/library/bk77x1wx(en-US,vs.80).aspx**).

As this is a library, the functions and procedures listed in this appendix do not execute from a MapBasic window in MapInfo Professional.

MICloseContent( ) procedure

Purpose

Closes and disposes of the CString handle and frees its memory.

Syntax

```
MICloseContent( ByVal hContent As CString )
```

hContent is the CString object handle to be disposed of.

Description

Use the **MICloseContent( )** procedure to close and free the CString object handle, obtained by calling the **MIGetContent( ) function**, when the handle is no longer in use.

See Also:

MIGetContent( ) function

MICloseFtpConnection( ) procedure

Purpose

Closes and disposes of the CFtpConnection handle and frees its memory.

Syntax

```
MICloseFtpConnection( ByVal hConnection As CFtpConnection )
```

hConnection is the CFtpConnection object handle to be disposed of.

Description

Use the **MICloseFtpConnection( )** procedure to close and free the CFtpConnection handle, obtained by calling the **MIGetFtpConnection( ) function**, when the handle is no longer in use.

See Also:

MIGetFtpConnection( ) function

MICloseFtpFileFind( ) procedure

Purpose

Closes and disposes of the CFtpFileFind handle and frees its memory.

Syntax

MICloseFtpFileFind(ByVal *hFTPFind* As CFtpFileFind)

hFTPFind is the handle to a CFtpFileFind object to be disposed of.

Description

Use the **MICloseFtpFileFind( )** procedure to close and free the CFtpFileFind handle, obtained by calling the **MIGetFtpFileFind( ) function**, when the handle is no longer in use.

See Also:

MIGetFtpFileFind( ) function

MICloseHttpConnection( ) procedure

Purpose

Closes and disposes of the CHttpConnection handle and frees its memory.

Syntax

MICloseHttpConnection(ByVal *hConnection* As CHttpConnection)

hConnection is the CHttpConnection object handle to be disposed of.

Description

Use the **MICloseHttpConnection( )** to close and free the CHttpConnection handle, obtained by calling the **MIGetHttpConnection( ) function**, when the handle is no longer in use.

See Also:

MIGetHttpConnection( ) function

MICloseHttpFile( ) procedure

Purpose

Closes and disposes of the CHttpFile handle and frees its memory.

Syntax

MICloseHttpFile(ByVal *hFile* As CHttpFile)

hFile is the CHttpFile object handle to be disposed of.

Description

Use the **MICloseHttpFile( )** procedure to close and free the CHttpFile object handle, obtained by calling the **MIOpenRequest( ) function** or the **MIOpenRequestFull( ) function**, when the handle is no longer in use.

See Also:

MIOpenRequest( ) function, **MIOpenRequestFull( ) function**

MICloseSession( ) procedure

Purpose

Closes and disposes of the CInternetSession handle and frees its memory.

Syntax

```
MICloseSession( ByVal hSession As CInternetSession )
```

hSession is the CInternetSession object handle to be disposed of.

Description

Use the **MICloseSession( )** procedure to close and free the CInternetSession handle, obtained by calling the **MICreateSession( ) function** or the **MICreateSessionFull( ) function**, when the handle is no longer in use.

See Also:

MICreateSession( ) function, **MICreateSessionFull( ) function**

MICreateSession( ) function

Purpose

Creates a CInternetSession object and returns the handle to it.

Syntax

```
MICreateSession( ByVal strAgent As String ) As CInternetSession
```

strAgent is a string that identifies the name of the application or entity calling the Internet functions (for example, “MapInfo Professional”). If the string is empty, the application name will be used.

Return Value

A handle to a CInternetSession object. If the call fails, Null is returned. To determine the cause of the failure, call the **MIGetErrorMessage( ) function**.

Description

MICreateSession( ) is the first Internet function called by an application. Use CInternetSession to create and initialize a single, or several simultaneous Internet sessions, and, if necessary, to describe your connection to a proxy server. If you want to perform service-specific (for example, HTTP, FTP) actions on files located on a server, you must establish the appropriate connection with that server. To open a particular kind of connection directly to a particular service, use the proper functions, such as the **MIGetHttpConnection( ) function** or the **MIGetFtpConnection( ) function**. For detailed information, refer to the Microsoft MSDN library.

The caller has to dispose of the handle by calling the **MICloseSession( ) procedure** when the handle is no longer in use.

See Also:

MICloseSession( ) procedure

MICreateSessionFull( ) function

Purpose

Creates a CInternetSession object and returns the handle to it.

Syntax

```
MICreateSessionFull( ByVal strAgent As String, ByVal dwContext As Integer,
   ByVal dwAccessType As Integer, ByVal strProxyName As String,
   ByVal strProxyBypass As String, ByVal dwFlags As Integer
   As CInternetSession
```

strAgent is a string that identifies the name of the application or entity calling the Internet functions (for example, “MapInfo Professional”). If the string is empty, the application name is used.

dwContext is the context identifier for the operation.

dwAccessType is The type of access required. The following are the valid values, exactly one of which may be supplied:

dwAccessType value	Definition
INTERNET_OPEN_TYPE_PRECONFIG	Connect using preconfigured settings in the registry. This access type is set as default. To connect through a TIS proxy, set *dwAccessType* to this value; you then set the registry appropriately.
INTERNET_OPEN_TYPE_DIRECT	Connect directly to Internet.
INTERNET_OPEN_TYPE_PROXY	Connect through a CERN proxy.

strProxyName is the name of the preferred CERN proxy if *dwAccessType* is set as INTERNET_OPEN_TYPE_PROXY. Default is an empty string.

strProxyBypass is a string that contains an option list of server addresses. These addresses may be bypassed when using proxy access. If an empty string is supplied, the bypass list will be read from the registry. This parameter is meaningful only if *dwAccessType* is set to INTERNET_OPEN_TYPE_PROXY.

dwFlags indicates various caching options. The default is set to 0. The possible values include:

dwFlag value	Definition
INTERNET_FLAG_DONT_CACHE	Do not cache the data, either locally or in any gateway servers.
INTERNET_FLAG_OFFLINE	Download operations are satisfied through the persistent cache only. If the item does not exist in the cache, an appropriate error code is returned.

Return Value

A handle to a CInternetSession object. If the call fails, Null is returned. To determine the cause of the failure, call the **MIGetErrorMessage( ) function**.

Description

MICreateSessionFull( ) is the first Internet function called by an application. Use CInternetSession to create and initialize a single or several simultaneous Internet sessions and, if necessary, to describe your connection to a proxy server. If you want to perform service-specific (for example, HTTP, FTP) actions on files located on a server, you must establish the appropriate connection with that server. To open a particular kind of connection directly to a particular service, use the proper functions, such as the **MIGetHttpConnection( ) function** or the **MIGetFtpConnection( ) function**. For detailed information, refer to the Microsoft MSDN library.

The caller has to dispose of the handle by calling the **MICloseSession( ) procedure** when the handle is no longer in use.

See Also:

MICreateSession( ) function, **MICloseSession( ) procedure**

MIErrorDlg( ) function

Purpose

Displays an error message dialog box.

Displays a dialog box for the error that is passed to MIErrorDlg, if an appropriate dialog box exists. The function also checks the headers of the specified CHttpFile for any hidden errors and displays a dialog box if needed.

Syntax

MIErrorDlg(ByVal *hFile* As CHttpFile, ByVal *dwError* As Integer) As Integer

hFile is a CHttpFile handle.

dwError is the error code which is used to get the error message.

Return Value

Returns one of the following values, otherwise returns an error value.

Error Code	Description
ERROR_SUCCESS	The function completed successfully. In the case of authentication this indicates that the user clicked the Cancel button.
ERROR_CANCELLED	The function was canceled by the user.
ERROR_INTERNET_FORCE_RETRY	This indicates that the function needs to redo its request. In the case of authentication this indicates that the user clicked the OK button.

Description

Use this function to get the error message in the form of a dialog box if an appropriate dialog box exists. It also checks the headers for any hidden errors and displays a dialog box if needed. For more information, refer to the Microsoft MSDN library. Allowable error codes are as follows:

Error Code	Description
ERROR_INTERNET_HTTP_TO_HTTPS_ON_REDIR	Notifies the user of the zero crossing to and from a secure site.
ERROR_INTERNET_INCORRECT_PASSWORD	Displays a dialog box requesting the user's name and password.
ERROR_INTERNET_INVALID_CA	Notifies the user that the function does not recognize the certificate authority that generated the certificate for this Secure Socket Layer (SSL) site.
ERROR_INTERNET_POST_IS_NON_SECURE	Displays a warning about posting data to the server through a nonsecure connection.

Error Code	Description
ERROR_INTERNET_SEC_CERT_CN_INVALID	Indicates that the SSL certificate Common Name (host name field) is incorrect. Displays an Invalid SSL Common Name dialog box and lets the user view the incorrect certificate. Also allows the user to select a certificate in response to a server request.
ERROR_INTERNET_SEC_CERT_DATE_INVALID	Notifies the user that the SSL certificate has expired.

For more information, refer to the Microsoft MSDN Library.

See Also:

MIOpenRequest( ) function, **MISendRequest( ) function**

MIFindFtpFile( ) function

Purpose

Finds an FTP file with the given CFtpFileFind handle.

Syntax

```
MIFindFtpFile( ByVal hFTPFind As CFtpFileFind, ByVal strName As String )
   As SmallInt
```

hFTPFind is a CFtpFileFind handle.

strDirName is a string that contains the name of the file to find. If it is empty, the call performs a wildcard search (*).

Return Value

Nonzero if successful; otherwise 0. To determine the cause of the failure, call the **MIGetErrorMessage( ) function**.

Description

After calling **MIFindFtpFile( )** to retrieve the first FTP file, you can call the **MIFindNextFtpFile( ) function** to retrieve subsequent FTP files.

See Also:

MIGetFtpFileFind( ) function, **MIFindNextFtpFile( ) function**, **MIGetFtpFileName( ) procedure**, **MIIsFtpDirectory( ) function**, **MIIsFtpDots( ) function**

MIFindNextFtpFile( ) function

Purpose

Continues a file search begun with a call to the **MIFindFtpFile( ) function** with the given CFtpFileFind handle.

Syntax

```
MIFindNextFtpFile( ByVal hFTPFind As CFtpFileFind ) As SmallInt
```

hFTPFind is a CFtpFileFind handle.

Return Value

Nonzero if there are more files; zero if the file found is the last one in the directory or if an error occurred.

Description

You must call **MIfindNextFtpFile( )** at least once before calling any attribute function, such as **MIGetFtpFileName( ) procedure**, **MIIsFtpDirectory( ) function**, and **MIIsFtpDots( ) function**.

See Also:

MIGetFtpFileFind( ) function, **MIFindFtpFile( ) function**, **MIGetFtpFileName( ) procedure**, **MIIsFtpDirectory( ) function**, **MIIsFtpDots( ) function**

MIGetContent( ) function

Purpose

Gets the content of the file.

Syntax

```
MIGetContent( ByVal hFile As CHttpFile )  As CString
```

hFile is a CHttpFile handle

Return Value

A handle to a CString object that contains the content of the file. If the call fails, Null is returned. To determine the cause of the failure, call **MIGetErrorMessage( ) function**.

Description

MIGetContent( ) gets the content of a file and stores it in a CString object, which is an alternative to the **MIGetContentToFile( ) function**. It has to remember that MIGetContentToFile and MIGetContent are exclusive, which means you can only use one of them during the life time of a CHttpFile object.

The caller has to dispose of the handle by calling the **MICloseContent( ) procedure** when the handle is no longer in use.

See Also:

MIOpenRequest( ) function, **MISendRequest( ) function**, **MICloseContent( ) procedure**, **MIGetContentToFile( ) function**

MIGetContentBuffer( ) function

Purpose

Gets the content in the format of a string with the given size.

Syntax

```
MIGetContentBuffer( ByVal hContent As CString, pBuffer As String,
   ByVal nLen As Integer As SmallInt
```

hContent is a CString handle.

pBuffer is a reference to a string that receives the content of the file.

nLen is the size of *pBuffer* in number of characters or bytes.

Return Value

Nonzero if successful; otherwise 0. To determine the cause of the failure, call the **MIGetErrorMessage( ) function**.

Description

Use **MIGetContentBuffer( )** to get the content of the file in string format with a given size. You have to allocate memory for *pBuffer* before calling this function.

See Also:

MIGetContent( ) function, **MIGetContentLen( ) function**, **MIGetContentString( ) function**.

MIGetContentLen( ) function

Purpose

Gets the content length.

Syntax

```
MIGetContentLen( ByVal hContent As CString ) As Integer
```

hContent is a CString handle.

Return Value

The content length in number of characters or bytes. Zero could be either that the call fails or that content is empty. To determine the cause of the failure, call the **MIGetErrorMessage( ) function**.

Description

Use **MIGetContentLen( )** to get the content length of the file in number of characters or bytes.

See Also:

MIGetErrorMessage( ) function

MIGetContentString( ) function

Purpose

Gets the content in the format of string.

Syntax

```
MIGetContentString( ByVal hContent As CString ) As String
```

hContent s a CString handle.

Return Value

A string that contains the content of the file. To determine the cause of the failure if an empty string is returned, call the **MIGetErrorMessage( ) function**.

Description

Use this function to get the content of the file in string format.

See Also:

MIGetContent( ) function, **MIGetContentLen( ) function**

MIGetContentToFile( ) function

Purpose

Saves the contents to a given file.

Syntax

```
MIGetContentToFile( ByVal hFile As CHttpFile,
   ByVal strFileName As String As SmallInt
```

hFile is a CHttpFile handle

strFileName is a string that identifies the local file name that receives the content of *hFile*.

Return Value

Nonzero if successful; otherwise 0. To determine the cause of the failure, call the **MIGetErrorMessage( ) function**.

Description

This function is used to save the content to a CHttpFile to a local file, which is an alternative to MIGetContent. It will create a new file with given file name. If the file exists already, it is truncated to 0 length. It has to remember that MIGetContentToFile and MIGetContent are exclusive, which means you can only use one of them during the life time of a CHttpFile object.

See Also:

MIOpenRequest( ) function, **MISendRequest( ) function**, **MIGetContent( ) function**.

MIGetContentType( ) function

Purpose

Gets the content type of the file.

Syntax

```
MIGetContentType( ByVal hFile As CHttpFile, pBuffer As String,
   pBufferLength As Integer As SmallInt
```

hFile is a CHttpFile handle.

pBuffer is a reference to a string that receives the content type.

pBufferLength is a reference to an integer that contains the length of *pBuffer* in number of characters or bytes on entry. When the function succeeds (a string is written to *pBuffer*), it contains the length of the string in characters, minus 1 for the terminating NULL character.

Return Value

Nonzero if successful; otherwise 0. To determine the cause of the failure, call the **MIGetErrorMessage( ) function**.

Description

Use **MIGetContentType( )** to get the content type of the file in string format. For example, written value in *pBuffer* could be “image/jpeg” or “text/html”.

See Also:

MIQueryInfo( ) function

MIGetCurrentFtpDirectory( ) function

Purpose

Gets the name of the current directory on the FTP server with the given CFtpConnection handle.

Syntax

```
MIGetCurrentFtpDirectory( Byval hConnection As CFtpConnection,
   pDirName As String, pLen As Integer As SmallInt
```

hConnection is a CFtpConnection handle.

pDirName is a reference to a string that receives the name of the directory.

pLen is a reference to an integer that contains the size of the buffer referenced by *pDirName* as input; the number of characters stored to *pDirName* as output.

Return Value

Nonzero if successful; otherwise 0. To determine the cause of the failure, call the **MIGetErrorMessage( ) function**.

Description

The parameters *pDirName* can be either fully qualified, or partially qualified, file names relative to the current directory. A backslash (\) or forward slash (/) can be used as the directory separator for either name. **MIGetCurrentFtpDirectory( )** translates the directory name separators to the appropriate characters before they are used.

See Also:

MIGetFtpConnection( ) function, **MISetCurrentFtpDirectory( ) function**

MIGetErrorCode( ) function

Purpose

Retrieves the last error that was set as a result of a call to a function in this library.

Syntax

```
MIGetErrorCode() As Integer
```

Return Value

Error code of the last error set.

Description

MIGetErrorCode( ) is called if the function's return value indicates its failure and you want to know the error code. To obtain an error message as a String, call the **MIGetErrorMessage( ) function**.

The error value retrieved is only set when certain errors occur during calls to other functions in the HTTP and FTP API. It is primarily useful for determining which error occurred as a result of a call to the **MISendRequest( ) function** or the **MISendSimpleRequest( ) function** so the correct value can be passed to the **MIErrorDlg( ) function**.

See Also:

MIGetErrorMessage( ) function, **MISendRequest( ) function**, **MISendSimpleRequest( ) function**, **MIErrorDlg( ) function**

MIGetErrorMessage( ) function

Purpose

Retrieves the last error message.

Syntax

```
MIGetErrorMessage( ) As String
```

Return Value

A string that contains the error message.

Description

MIGetErrorMessage( ) should be called immediately to get useful data when a function's return value indicates its failure and you want to know the cause. Many of the returned errors are system-set errors.

MIGetFileURL( ) function

Purpose

Gets the name of the HTTP file as a URL.

Syntax

```
MIGetFileURL( ByVal hFile As CHttpFile, pURL As String,
   ByVal lURLLen As Integer ) As SmallInt
```

hFile is a CHttpFile handle.

pURL is a reference to a string that receives the name of the HTTP file as a URL.

pURLLen is the size of *pURL* in number of characters or bytes.

Return Value

Nonzero if successful; otherwise 0. To determine the cause of the failure, call the **MIGetErrorMessage( ) function**.

Description

Use **MIGetFileURL( )** to get the name of the HTTP file as a URL.

See Also:

MISendRequest( ) function, **MIOpenRequest( ) function**

MIGetFtpConnection( ) function

Purpose

Establishes an FTP connection and gets a handle to a CFtpConnection object.

Syntax

```
MIGetFtpConnection( ByVal hSession As CInternetSession,
   ByVal strServer As String, ByVal strUserName As String,
   ByVal strPassword As String, ByVal nPort As INTERNET_PORT )
   As CFtpConnection
```

hSession is a CinternetSession handle.

strServer is a string that contains the FTP server name.

strUserName is a string that specifies the name of the user to log in.

strPassword is a string that specifies the password to use to log in.

nPort is a number that identifies the TCP/IP port to use on the server.

Return Value

A handle to a CFtpConnection object. If the call fails, Null is returned. To determine the cause of the failure, call the **MIGetErrorMessage( ) function**.

Description

MIGetFtpConnection( ) connects to an FTP server, creates and returns a handle to a CFtpConnection object. It does not perform any specific operation on the server. If you intend to get or put files, for example, you must perform those operations as separate steps.

The caller has to dispose of the handle by calling the **MICloseFtpConnection( ) procedure** when the handle is no longer in use.

See Also:

MICloseFtpConnection( ) procedure, **MIGetHttpConnection( ) function**, **MICreateSession( ) function**, **MICreateSessionFull( ) function**, **MIParseURL( ) function**.

MIGetFtpFile( ) function

Purpose

Gets a file from an FTP server with the given CFtpConnection handle and stores it on the local. machine.

Syntax

```
MIGetFtpFile( ByVal hConnection As CFtpConnection,
   ByVal strRemoteFile As String, ByVal strLocalFile As String,
   ByVal bFailIfExists As SmallInt, ByVal dwAttributes As Integer,
   ByVal dwFlags As Integer ) As SmallInt
```

hConnection is a CFtpConnection handle.

strRemoteFile is a string that contains the name of a file to retrieve from the FTP server.

strLocalFile is a string that contains the name of the file to create on the local system.

bFailIfExists indicates whether the file name may already be used by an existing file. If the local file name already exists, and this parameter is TRUE, **MIGetFtpFile( )** fails. Otherwise, **MIGetFtpFile( )** erases the existing copy of the file.

dwAttributes indicates the attributes of the file. This can be any combination of the following FILE_ATTRIBUTE_* flags.

dwAttirbute value	Definition
FILE_ATTRIBUTE_ARCHIVE	The file is an archive file. Applications use this attribute to mark files for backup or removal.
FILE_ATTRIBUTE_COMPRESSED	The file or directory is compressed. For a file, compression means that all of the data in the file is compressed. For a directory, compression is the default for newly created files and subdirectories.
FILE_ATTRIBUTE_DIRECTORY	The file is a directory.
FILE_ATTRIBUTE_NORMAL	The file has no other attributes set. This attribute is valid only if used alone. All other file attributes override FILE_ATTRIBUTE_NORMAL:
FILE_ATTRIBUTE_HIDDEN	The file is hidden. It is not to be included in an ordinary directory listing.
FILE_ATTRIBUTE_READONLY	The file is read only. Applications can read the file but cannot write to it or delete it.

dwAttirbute value	Definition
FILE_ATTRIBUTE_SYSTEM	The file is part of, or is used exclusively by, the operating system.
FILE_ATTRIBUTE_TEMPORARY	The file is being used for temporary storage. Applications should write to the file only if absolutely necessary. Most of the file's data remains in memory without being flushed to the media because the file will soon be deleted.

dwFlags specifies the conditions under which the transfer occurs. This parameter can be any of the following values:

dwFlags value	Definition
FTP_TRANSFER_TYPE_ASCII	Transfers the file using FTP's ASCII (Type A) transfer method. Control and formatting information is converted to local equivalents.
FTP_TRANSFER_TYPE_BINARY	Transfers the file using FTP's Image (Type I) transfer method. The file is transferred exactly as it exists with no changes. This is the default transfer method.
FTP_TRANSFER_TYPE_UNKNOWN	Defaults to FTP_TRANSFER_TYPE_BINARY.

Return Value

Nonzero if successful; otherwise 0. To determine the cause of the failure, call the **MIGetErrorMessage( ) function**.

Description

Both *strRemoteFile* and *strLocalFile* can be either partially qualified file names relative to the current directory, or fully qualified. A backslash (`\`) or forward slash (`/`) can be used as the directory separator for either name.

See Also:

MIGetFtpConnection( ) function, **MIPutFtpFile( ) function**

MIGetFtpFileFind( ) function

Purpose

Gets a handle to a CFtpFileFind object.

Syntax

```
MIGetFtpFileFind( ByVal hConnection As CFtpConnection ) As CFtpFileFind
```

hConnection is a CFtpConnection handle.

Return Value

A handle to a CFtpFileFind object. If the call fails, Null is returned. To determine the cause of the failure, call the **MIGetErrorMessage( ) function**.

Description

The CFtpFileFind class aids in Internet file searches of FTP servers.

The caller has to dispose of the handle by calling **MICloseFtpFileFind( ) procedure** when the handle is no longer in use.

See Also:

MIGetFtpConnection( ) function, **MICloseFtpFileFind( ) procedure**

MIGetFtpFileName( ) procedure

Purpose

Gets the name of the found file with the given CFtpFileFind handle.

Syntax

```
MIGetFtpFileName( ByVal hFTPFind As CFtpFileFind, pFileName As String,
   ByVal bufferlen As Integer )
```

hFTPFind is a CFtpFileFind handle.

pFileName is a reference to a string that will receive the name of the found file.

bufferlen is the size of the buffer referenced by *pFileName*.

Description

You must call **MIFindNextFtpFile( ) function** at least once before calling **MIGetFtpFileName( )**.

See Also:

MIGetFtpFileFind( ) function, **MIFindNextFtpFile( ) function**, **MIFindFtpFile( ) function**

MIGetHttpConnection( ) function

Purpose

Establishes an HTTP connection and gets a handle to a CHttpConnection object.

Syntax

```
MIGetHttpConnection( ByVal hSession As CInternetSession,
   ByVal strServer As String, ByVal nPort As INTERNET_PORT
   As CHttpConnection
```

hSession is a CinternetSession handle.

strServer is a string that contains the HTTP server name.

nPort is a number that identifies the TCP/IP port to use on the server.

Return Value

A handle to a CHttpConnection object. If the call fails, Null is returned. To determine the cause of the failure, call the **MIGetErrorMessage( ) function**.

Description

MIGetHttpConnection( ) connects to an HTTP server, creates and returns a handle to a CHttpConnection object. It does not perform any specific operation on the server. If you intend to query an HTTP header, for example, you must perform this operation in separate steps.

The caller has to dispose of the handle by calling the **MICloseHttpConnection( ) procedure** when the handle is no longer in use.

See Also:

MICloseHttpConnection( ) procedure, **MIGetFtpConnection( ) function**, **MICreateSession( ) function**, **MICreateSessionFull( ) function**, **MIParseURL( ) function**

MIIsFtpDirectory( ) function

Purpose

Determines if the found file is a directory with the given CFtpFileFind handle.

Syntax

```
MIIsFtpDirectory( ByVal hFTPFind As CFtpFileFind ) As SmallInt
```

hFTPFind is a CFtpFileFind handle.

Return Value

Nonzero if the found file is a directory; otherwise 0.

Description

You must call **MIFindNextFtpFile( ) function** at least once before calling **MIIsFtpDirectory( )**.

See Also:

MIGetFtpFileFind( ) function, **MIFindNextFtpFile( ) function**, **MIFindFtpFile( ) function**, **MIGetFtpFileName( ) procedure**

MIIsFtpDots( ) function

Purpose

Tests for the current directory and parent directory markers while iterating through files with the given CFtpFileFind handle.

Syntax

```
MIIsFtpDots( ByVal hFTPFind As CFtpFileFind ) As SmallInt
```

hFTPFind is a CFtpFileFind handle.

Return Value

Nonzero if the found file has the name "`.`" or "`..`", which indicates that the found file is actually a directory. Otherwise 0.

Description

You must call the **MIFindNextFtpFile( ) function** at least once before calling **MIIsFtpDots( )**.

See Also:

MIGetFtpFileFind( ) function, **MIFindNextFtpFile( ) function**, **MIFindFtpFile( ) function**, **MIGetFtpFileName( ) procedure**

MIOpenRequest( ) function

Purpose

Opens an HTTP connection.

Syntax

```
MIOpenRequest( ByVal hConnection As CHttpConnection,
   ByVal nVerb As Integer, ByVal strObjectName As String
) As CHttpFile
```

hConnection is a CHttpConnection handle.

nVerb is a number associated with the HTTP request type. Can be one of the following:

```
HTTP_VERB_POST
HTTP_VERB_GET
HTTP_VERB_HEAD
HTTP_VERB_PUT
```

```
HTTP_VERB_LINK
HTTP_VERB_DELETE
HTTP_VERB_UNLINK
```

strObjectName is a string containing the target object of the specified verb. This is generally a file name, an executable module, or a search specifier.

Return Value

A handle to a CHttpFile object requested. If the call fails, Null is returned. To determine the cause of the failure, call the **MIGetErrorMessage( ) function**.

Description

This function opens an HTTP connection and returns a handle to a CHttpFile object, which provides services requesting and reading files on an HTTP server. If your Internet session reads data from an HTTP server, you must get a handle to CHttpFile object.

The caller has to dispose of the handle by calling **MICloseHttpFile( ) procedure** when the handle is no longer in use.

See Also:

MIGetHttpConnection( ) function, **MIOpenRequestFull( ) function**, **MIParseURL( ) function**

MIOpenRequestFull( ) function

Purpose

Opens an HTTP connection.

Syntax

```
MIOpenRequestFull( ByVal hConnection As CHttpConnection,
   ByVal nVerb As Integer, ByVal strObjectName As String,
   ByVal strReferer As String, ByVal dwContext As Integer,
   ByVal strVersion As String, ByVal dwFlags As Integer )
   As CHttpFile
```

hConnection is a CHttpConnection handle.

nVerb is a number associated with the HTTP request type. Can be one of the following:

- HTTP_VERB_POST
- HTTP_VERB_GET
- HTTP_VERB_HEAD
- HTTP_VERB_PUT
- HTTP_VERB_LINK
- HTTP_VERB_DELETE
- HTTP_VERB_UNLINK

strObjectName is a string containing the target object of the specified verb. This is generally a file name, an executable module, or a search specifier.

strReferer is a string that specifies the address (URL) of the document from which the URL in the request (*strObjectName*) was obtained. If the string is empty, no HTTP header is specified.

dwContext is the context identifier for the **MIOpenRequestFull( )** operation. For detailed information, refer to the Microsoft MSDN library.

strVersion is a string defining the HTTP version. If the string is empty, “HTTP/1.0” is used.

dwFlags is any combination of the following INTERNET_ FLAG_* flags:

dwFlag value	Definition
INTERNET_FLAG_RELOAD	Forces a download of the requested file, object, or directory listing from the origin server, not from the cache.
INTERNET_ FLAG_DONT_CACHE	Does not add the returned entry to the cache.
INTERNET_FLAG_MAKE_PERSISTENT	Adds the returned entity to the cache as a persistent entity. This means that standard cache cleanup, consistency checking, or garbage collection cannot remove this item from the cache.
INTERNET_ FLAG_SECURE	Use secure transaction semantics. This translates to using SSL/PCT and is only meaningful in HTTP requests.
INTERNET_ FLAG_NO_AUTO_REDIRECT	Used only with HTTP, specifies that redirection should not be automatically handled in the **MISendRequest( ) function**.

Return Value

A handle to a CHttpFile object requested. If the call fails, Null is returned. To determine the cause of the failure, call the **MIGetErrorMessage( ) function**.

Description

This function opens an HTTP connection and returns a handle to a CHttpFile object, which provides services requesting and reading files on an HTTP server. If your Internet session reads data from an HTTP server, you must get a handle to CHttpFile object. This function wraps the MFC function OpenRequest which has an additional parameter to indicate accepted types. In this version, the wrapper function always sets this parameter to NULL.

The caller has to dispose of the handle by calling the **MICloseHttpFile( ) procedure** when the handle is no longer in use.

See Also:

MIGetHttpConnection( ) function, MIOpenRequest( ) function, MIParseURL( ) function

MIParseURL( ) function

Purpose

Parses a URL string and returns the type of service and its components.

Syntax

```
MIParseURL( ByVal strURL As String, pServiceType As Integer,
   pServer As String, ByVal nServerLen As Integer,
   pObject As String, ByVal nObjectLen As Integer,
   pPort As INTERNET_PORT )
   As SmallInt
```

strURL is a string that contains the URL to be parsed.

pServiceType is a reference to a integer that receives the type of Internet service. The possible values are one of the following:

- INTERNET_SERVICE_FTP
- INTERNET_SERVICE_GOPHER
- INTERNET_SERVICE_HTTP
- AFX_INET_SERVICE_UNK
- AFX_INET_SERVICE_FILE
- AFX_INET_SERVICE_MAILTO
- AFX_INET_SERVICE_MID
- AFX_INET_SERVICE_CID
- AFX_INET_SERVICE_NEWS
- AFX_INET_SERVICE_NNTP
- AFX_INET_SERVICE_PROSPERO
- AFX_INET_SERVICE_TELNET
- AFX_INET_SERVICE_WAIS
- AFX_INET_SERVICE_AFS
- AFX_INET_SERVICE_HTTPS

strsServer is a reference to a string that specifies the first segment of the URL following the service type.

nServerLen is the size of the buffer referenced by *strsServer*.

pObject is a reference to an object that the URL refers to (may be empty).

nObjectLen is the size of the buffer referenced by *pObject*.

pPort is a reference to an integer that contains the determined port number from either the Server or Object portions of the URL, if either exists. The port number is used to identify the TCP/IP port to use on the server.

Return Value

Nonzero if the URL was successfully parsed; otherwise, 0 if it is empty or does not contain a known Internet service type. To determine the cause of the failure, call the **MIGetErrorMessage( ) function**.

Description

MIParseURL( ) parses a URL string and returns the type of service and its components. For example, it parses URLs of the form: `ftp://ftp.mysite.org/` and returns its components stored as follows:

```
pServer == "ftp.mysite.org"
pObject == "/"
nPort == #port
pServiceType == INTERNET_SERVICE_FTP
```

MIPutFtpFile( ) function

Purpose

Stores a file on an FTP server with the given CFtpConnection handle.

Syntax

```
MIPutFtpFile( ByVal hConnection As CFtpConnection,
   ByVal strLocalFile As String, ByVal strRemoteFile As String,
   ByVal dwFlags As Integer )
   As SmallInt
```

hConnection is a CFtpConnection handle.

strLocalFile is a string that contains the name of the file to send from the local system.

strRemoteFile is a string that contains the name of the file to create on the FTP server.

dwFlags specifies the conditions under which the transfer occurs. This parameter can be any of the following values:

dwFlag value	Definition
FTP_TRANSFER_TYPE_ASCII	The file transfers using FTP ASCII (Type A) transfer method. Converts control and formatting information to local equivalents.
FTP_TRANSFER_TYPE_BINARY	The file transfers data using FTP's Image (Type I) transfer method. The file transfers data exactly as it exists, with no changes. This is the default transfer method.

Return Value

Nonzero if successful; otherwise 0. To determine the cause of the failure, call the **MIGetErrorMessage( ) function**.

Description

Both *strRemoteFile* and *strLocalFile* can be either partially qualified file names relative to the current directory, or fully qualified. A backslash (`\`) or forward slash (`/`) can be used as the directory separator for either name.

See Also:

MIGetFtpConnection( ) function, **MIGetFtpFile( ) function**

MIQueryInfo( ) function

Purpose

Returns response or request headers from an HTTP request.

Syntax

```
MIQueryInfo( ByVal hFile As CHttpFile, ByVal dwInfoLevel As Integer,
   pBuffer As String, pBufferLength As Integer ) As SmallInt
```

hFile is a CHttpFile handle.

dwInfoLevel is a combination of the attribute to query, and a modifier flag that specifies the type of information requested: For a list of the modifier flags, refer to the Microsoft MSDN library.

pBuffer is a reference to a string that receives the information. For the attribute HTTP_QUERY_CUSTOM, *pBuffer* is also an input indicating which header name to query.

pBufferLength is a reference to an integer that contains the length of *pBuffer* in number of characters or bytes on entry. When the function succeeds (a string is written to *pBuffer*), it contains the length of the string in characters minus 1 for the terminating NULL character.

Return Value

Nonzero if successful; otherwise 0. To determine the cause of the failure, call the **MIGetErrorMessage( ) function**.

Description

Use this function to get response or request headers from an HTTP request. For a description of attribute values, refer to the Microsoft MSDN library for information on Query Info flags (**http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wininet/wininet/query_info_flags.asp**).

See Also:

MIOpenRequest( ) function, **MISendRequest( ) function**

MIQueryInfoStatusCode( ) function

Purpose

Gets the status code associated with an HTTP request.

Syntax

```
MIQueryInfoStatusCode( ByVal hFile As CHttpFile, pStatusCode As Integer
   s SmallInt
```

hFile is a CHttpFile handle.

pStatusCode is a reference to an integer that receives the status code. Status codes indicate the success or failure of the requested event. HTTP status codes fall into groups indicating the success or failure of the request. The following tables outline the status code groups and the most common HTTP status codes.

HTTP Status Code Groups

Group	Meaning
200-299	Success
300-399	Information
400-499	Request error
500-599	Server error

Common HTTP Status Codes

Status code	Meaning
200	URL located, transmission follows.
400	Unintelligible request.
404	Requested URL not found.
405	Server does not support requested method.
500	Unknown server error.
503	Server capacity reached.

Return Value

Nonzero if successful; otherwise 0. To determine the cause of the failure, call the **MIGetErrorMessage( ) function**.

Description

Use this function to get the status code associated with an HTTP request. For information, refer to the Microsoft MSDN library.

See Also:

MIOpenRequest( ) function, **MISendRequest( ) function**

MISaveContent( ) function

Purpose

Saves the content to a given file.

Syntax

```
MISaveContent( ByVal hContent As CString, ByVal strFileName As String
   As SmallInt
```

hContent is a CString handle.

strFileName is a string that identifies the file name that receives the content.

Return Value

Nonzero if successful; otherwise 0. To determine the cause of the failure, call the **MIGetErrorMessage( ) function**.

Description

This function is used to save the content to a file. It will create a new file with the given file name. If the file exists already, it is truncated to 0 length.

See Also:

MIGetContent( ) function

MISendRequest( ) function

Purpose

Sends a request to an HTTP server.

Syntax

```
MISendRequest( ByVal hFile As CHttpFile, ByVal strHeaders As String,
   ByVal dwHeadersLen As Integer, ByVal strOptional As String,
   ByVal dwOptionalLen As Integer, ByVal bAuthenticate As SmallInt
   As SmallInt
```

hFile is a CHttpFile handle.

strHeaders is a string containing the name of the headers to send.

dwHeadersLen is the length of the headers identified by *strHeaders*.

strOptional is any optional data to send immediately after the request headers. This is generally used for POST and PUT operations. This can be empty if there is no optional data to send.

dwOptionalLen is the length of *strOptional*.

bAuthenticate indicates whether to check authentication or not.

Return Value

Nonzero if successful; otherwise 0. To determine the cause of the failure, call the **MIGetErrorMessage( ) function**.

Description

This function sends a request to an HTTP server.

See Also:

MIOpenRequest( ) function, **MIOpenRequestFull( ) function**

MISendSimpleRequest( ) function

Purpose

Sends a request to an HTTP server.

Syntax

```
MISendSimpleRequest( ByVal hFile As CHttpFile,
   ByVal bAuthenticate As SmallInt ) As SmallInt
```

hFile is a CHttpFile handle.

bAuthenticate indicates whether to check authentication or not.

Return Value

Nonzero if successful; otherwise 0. To determine the cause of the failure, call the **MIGetErrorMessage( ) function**.

Description

This function sends a request to an HTTP server.

See Also:

MIOpenRequest( ) function, **MIOpenRequestFull( ) function**, **MISendRequest( ) function**

MISetCurrentFtpDirectory( ) function

Purpose

Changes to a different directory on the FTP server with the given CFtpConnection handle.

Syntax

```
MISetCurrentFtpDirectory( Byval hConnection As CFtpConnection,
   Byval strDirName As String ) As SmallInt
```

hConnection is a CFtpConnection handle.

strDirName is a string that contains the name of the directory.

Return Value

Nonzero if successful; otherwise 0. To determine the cause of the failure, call the **MIGetErrorMessage( ) function**.

Description

The *pDirName* parameter can be either a partially or fully qualified file name relative to the current directory. A backslash (\) or forward slash (/) can be used as the directory separator for either name.

MISetCurrentFtpDirectory( ) translates the directory name separators to the appropriate characters before they are used.

See Also:

MIGetFtpConnection( ) function, **MIGetCurrentFtpDirectory( ) function**

MISetSessionTimeout( ) function

Purpose

Sets the time-out options for the Internet session.

Syntax

```
MISetSessionTimeout( ByVal hSession As CInternetSession,
   ByVal Connect As Integer, ByVal Send As Integer,
   ByVal Receive As Integer )  As SmallInt
```

hSession is the CInternetSession object handle.

Connect is an integer that contains time-out value in millisecond to use for the Internet connection request.

Send is an integer that contains the time-out value in milliseconds to use for sending a request.

Receive is an integer that contains the time-out value in milliseconds to use for receiving a request.

Return Value

Nonzero if successful; otherwise 0. To determine the cause of the failure, call the **MIGetErrorMessage( ) function**.

Description

Use this function to set time-out values for the Internet session. The default value of each setting (*Connect*, *Send*, *Receive*) is 0. For detailed information, refer to the Microsoft MSDN library.

See Also:

MICreateSession( ) function, **MICreateSessionFull( ) function**

B

XML Library

This appendix details the XML document library that enables MapBasic programmers to create and parse XML documents and other web-based technology. This library uses common DEF files: XMLLib.DEF and XMLTypes.DEF, which are installed in <*Your MapBasic Installation Directory*>\Samples\MapBasic\INC. Make sure you include these files as header files into your programs. All the functionality described in this appendix is also dependent on the presence of GmlXlat.dll which is installed with MapInfo Professional.

All of the functions and procedures listed in this appendix are wrappers of the corresponding methods of Microsoft XML Interfaces and Classes. Wrapped classes include: IXMLDOMNode, IXMLDOMNodeList, IXMLDOMNamedNodeMap, IXMLDOMSchemaCollection2, and IXMLDOMDocument2. For more detailed information about the usage of the related classes and interfaces refer to the MSDN reference **http://msdn.microsoft.com/library/default.asp?url=/library/en-us/xmlsdk/html/39b17b9c-04c7-4fa8-bcee-1f7d57eefd74.asp**

As this is a library, the functions and procedures listed in this appendix do not execute from a MapBasic window in MapInfo Professional.

MIXmlAttributeListDestroy( ) procedure

Purpose

Disposes of the MIXmlNamedNodeMap object and frees its memory.

Syntax

```
MIXmlNodeListDestroy( ByVal hXMLNodeList As MIXmlNodeList )
```

hXMLAttributeList is The MIXmlNamedNodeMap object handle to be disposed of.

Description

The caller has to call this function to free the MIXmlNamedNodeMap handle obtained by calling the **MIXmlGetAttributeList( ) function**, when the handle is no longer in use.

See Also:

MIXmlGetAttributeList( ) function

MIXmlDocumentCreate( ) function

Purpose

Creates an MIXmlDocument object and gets a handle to the object.

Syntax

```
MIXmlDocumentCreate( ) As MIXmlDocument
```

Return Value

A handle to the MIXmlDocument object. If the call fails, Null is returned. To determine the cause of the failure, call the **MIGetErrorMessage( ) function**.

Description

MIXmlDocumentCreate( ) creates and returns a handle to an MIXmlDocument object. It represents the top level of the XML source.

The caller has to dispose of the handle by calling the **MIXmlDocumentDestroy( ) procedure** when the handle is no longer in use.

See Also:

MIXmlDocumentDestroy( ) procedure

MIXmlDocumentDestroy( ) procedure

Purpose

Disposes of the MIXmlDocument and frees its memory.

Syntax

```
MIXmlDocumentDestroy( ByVal hXMLDocument As MIXmlDocument )
```

hXMLDocument is the MIXmlDocument object handle to be disposed of.

Description

The caller has to call this function to close and free the MIXmlDocument handle obtained by calling the **MIXmlDocumentCreate( ) function** when the handle is no longer in use.

See Also:

MIXmlDocumentCreate( ) function

MIXmlDocumentGetNamespaces( ) function

Purpose

Creates an MIXMLSchemaCollection object and gets a handle to the object.

Syntax

```
MIXmlDocumentGetNamespaces( ByVal hXMLDocument As MIXmlDocument )
   As MIXMLSchemaCollection
```

hXMLDocument is the MIXmlDocument object handle.

Return Value

A handle to an MIXMLSchemaCollection object if successful; otherwise NULL.

Description

This method creates an MIXMLSchemaCollection object.

The caller has to dispose of the handle by calling the **MIXmlSCDestroy( ) procedure** when the handle is no longer in use.

See Also:

MIXmlDocumentCreate( ) function, **MIXmlSCDestroy( ) procedure**

MIXmlDocumentGetRootNode( ) function

Purpose

Retrieves the root element of the document.

Syntax

```
MIXmlDocumentGetRootNode( ByVal hXMLDocument As MIXmlDocument )
   As MIXmlNode
```

hXMLDocument is the MIXmlDocument object handle.

Return Value

A handle to an MIXmlNode object representing the root element of the document if successful; otherwise NULL.

Description

MIXmlDocumentGetRootNode( ) retrieves a handle to an MIXmlNode object that represents the root of the XML document tree. It returns NULL if no root exists.

The caller has to dispose of the handle by calling the **MIXmlNodeDestroy( ) procedure** when the handle is no longer in use.

See Also:

MIXmlDocumentCreate( ) function, **MIXmlNodeDestroy( ) procedure**

MIXmlDocumentLoad( ) function

Purpose

Loads an XML document from the specified location.

Syntax

```
MIXmlDocumentLoad( ByVal hXMLDocument As MIXmlDocument,
   ByVal strPath As String, pbParsingError As SmallInt,
   ByVal bValidate As SmallInt, ByVal bResolveExternals As SmallInt )
   As SmallInt
```

hXMLDocument is the MIXmlDocument object handle.

strPath is a string containing the path/URL that specifies the location of the XML file.

pbParsingError is a reference to a SmallInt that indicates TRUE if the load succeeded; FALSE if the load failed.

bValidate is a SmallInt that indicates whether the parser should validate this document. If TRUE (1), it validates during parsing. If FALSE (0), it parses only for well-formed XML.

bResolveExternals is a SmallInt that indicates whether external definitions, resolvable namespaces, document type definition (DTD) external subsets, and external entity references, are to be resolved at parse time, independent of validation. When the *bResolveExternals* parameter is TRUE (1), external definitions are resolved at parse time. This allows default attributes and data types to be defined on elements from the schema and allows use of the DTD as a file inclusion mechanism. This setting is independent of whether validation is to be performed, as indicated by the value of the *bValidate* property. If externals cannot be resolved during validation, a validation error occurs. When the value of *bResolveExternals* is FALSE (0), externals are not resolved and validation is not performed.

Return Value

Nonzero if successful, otherwise 0. To determine the cause of the failure, call the **MIGetErrorMessage( ) function**.

Description

If the URL cannot be resolved or accessed or does not reference an XML document, this method returns FALSE. Calling **MIXmlDocumentLoad( )** on an existing document immediately discards the content of the document. If loading an XML document from a resource, the load must be performed asynchronously or the load will fail.

See Also:

MIXmlDocumentCreate( ) function, **MIXmlDocumentLoadXML( ) function**, **MIXmlDocumentLoadXMLString( ) function**

MIXmlDocumentLoadXML( ) function

Purpose

Loads an XML document using the supplied string.

Syntax

```
MIXmlDocumentLoadXML( ByVal hXMLDocument As MIXmlDocument,
   ByVal hContent As CString, pbParsingError As SmallInt,
   ByVal bValidate As SmallInt, ByVal bResolveExternals As SmallInt )
   As SmallInt
```

hXMLDocument is The MIXmlDocument object handle.

hContent is a CString handle to the string containing the XML string to load into this XML document object. This string can contain an entire XML document or a well-formed fragment.

pbParsingError is a reference to a SmallInt that indicates TRUE (nonzero) if the load succeeded; FALSE (0) if the load failed.

bValidate is a SmallInt that Indicates whether the parser should validate this document. If TRUE (1), it validates during parsing. If FALSE (0), it parses only for well-formed XML.

bResolveExternals is a SmallInt that indicates whether external definitions, resolvable namespaces, document type definition (DTD) external subsets, and external entity references, are to be resolved at parse time, independent of validation. When the *bResolveExternals* parameter is TRUE (1), external definitions are resolved at parse time. This allows default attributes and data types to be defined on elements from the schema and allows use of the DTD as a file inclusion mechanism. This setting is independent of whether validation is to be performed, as indicated by the value of the *bValidate* property. If externals cannot be resolved during validation, a validation error occurs. When the value of *bResolveExternals* is FALSE (0), externals are not resolved and validation is not performed.

Return Value

Nonzero if successful, otherwise 0. To determine the cause of the failure, call the **MIGetErrorMessage( ) function**.

Description

Calling **MIXmlDocumentLoadXML( )** on an existing document immediately discards the content of the document. It will work only with UTF-16 or UCS-2 encodings.

See Also:

MIXmlDocumentCreate( ) function, **MIXmlDocumentLoad( ) function**, **MIXmlDocumentLoadXMLString( ) function**

MIXmlDocumentLoadXMLString( ) function

Purpose

Loads an XML document using a supplied string.

Syntax

```
MIXmlDocumentLoadXMLString( ByVal hXMLDocument As MIXmlDocument,
   ByVal strXML As String, pbParsingError As SmallInt,
   ByVal bValidate As SmallInt, ByVal bResolveExternals As SmallInt )
   As SmallInt
```

hXMLDocument is the MIXmlDocument object handle.

strXML is a string containing the XML string to load into this XML document object. This string can contain an entire XML document or a well-formed fragment.

pbParsingError is a reference to a SmallInt that indicates TRUE (nonzero) if the load succeeded; FALSE (0) if the load failed.

bValidate is a SmallInt that Indicates whether the parser should validate this document. If TRUE (1), it validates during parsing. If FALSE (0), it parses only for well-formed XML.

bResolveExternals is a SmallInt that indicates whether external definitions, resolvable namespaces, document type definition (DTD) external subsets, and external entity references, are to be resolved at parse time, independent of validation. When the *bResolveExternals* parameter is TRUE (1),

external definitions are resolved at parse time. This allows default attributes and data types to be defined on elements from the schema and allows use of the DTD as a file inclusion mechanism. This setting is independent of whether validation is to be performed, as indicated by the value of the *bValidate* property. If externals cannot be resolved during validation, a validation error occurs. When the value of *bResolveExternals* is FALSE (0), externals are not resolved and validation is not performed.

Return Value

Nonzero if successful, otherwise 0. To determine the cause of the failure, call the **MIGetErrorMessage( ) function**.

Description

Calling **MIXmlDocumentLoadXMLString( )** on an existing document immediately discards the content of the document. It will work only with UTF-16 or UCS-2 encodings.

See Also:

MIXmlDocumentCreate( ) function, **MIXmlDocumentLoad( ) function**, **MIXmlDocumentLoadXML( ) function**.

MIXmlDocumentSetProperty( ) function

Purpose

Sets the properties for the MIXmlDocument object.

Syntax

```
MIXmlDocumentSetProperty( ByVal hXMLDocument As MIXmlDocument,
   ByVal strPropertyName As String, ByVal strPropertyValue As String )
   As SmallInt
```

hXMLDocument is the MIXmlDocument object handle.

strPropertyName is a string that contains the name of the property to be set. For a list of properties that can be set using this method, refer to the Microsoft MSDN library.

strPropertyValue is a string that contains the value of the specified property. For a list of property values that can be set using this method, refer to the Microsoft MSDN library.

Return Value

Nonzero if successful; otherwise 0.

Description

This method sets the property on the MIXmlDocument object. There are some limitation on which properties can be set using this method. For details, refer to the Microsoft MSDN library.

See Also:

MIXmlDocumentCreate( ) function, MIXmlDocumentLoad( ) function

MIXmlGetAttributeList( ) function

Purpose

Retrieves the MIXmlNamedNodeMap object with the given node.

Syntax

```
MIXmlGetAttributeList( ByVal hXMLNode As MIXmlNode ) As MIXmlNamedNodeMap
```

hXMLNode is the MIXmlNode object handle.

Return Value

A handle to the MIXmlNamedNodeMap object that contains the nodes which can return attributes. Returns NULL for all other node types.

Description

MIXmlGetAttributeList( ) creates an MIXmlNamedNodeMap object and returns the handle to the object. This object only contains the nodes which can return attributes (Element, Entity, and Notation nodes). Null is returned for all other node types. For the valid node types, a handle to an MIXmlNamedNodeMap object is always returned; when there are no attributes on the element, the list length is set to zero. For detailed information and the list of valid node types, refer to the Microsoft MSDN library.

The caller has to dispose of the handle by calling the MIXmlAttributeListDestroy( ) procedure when the returned handle is no longer in use.

See Also:

MIXmlDocumentGetRootNode( ) function, MIXmlAttributeListDestroy( ) procedure

MIXmlGetChildList( ) function

Purpose

Gets an MIXmlNodeList object that contains the children nodes of the given node instance.

Syntax

```
MIXmlGetChildList( ByVal hXMLNode As MIXmlNode) As MIXmlNodeList
```

hXMLNode is the MIXmlNode object handle.

Return Value

A handle to the MIXmlNodeList object that contains the children nodes of the given node instance if successful; otherwise NULL.

Description

MIXmlGetChildList( ) is used to get a list of children in the given node. An MIXmlNodeList object is returned even if there are no children of the node. In such a case, the length of the list is set to 0. This value depends on the value of the node type. For more information, refer to the Microsoft MSDN library.

The caller has to dispose of the handle by calling the **MIXmlNodeListDestroy( ) procedure** when the handle is no longer in use.

See Also:

MIXmlNodeListDestroy( ) procedure, **MIXmlSelectNodes( ) function**

MIXmlGetNextAttribute( ) function

Purpose

Returns the next node in the collection.

Syntax

```
MIXmlGetNextAttribute( ByVal hXMLAttributeList As MIXmlNamedNodeMap )
   As MIXmlNode
```

hXMLAttributeList is the MIXmlNamedNodeMap object handle.

Return Value

A handle to the MIXmlNode object which refers to the next node in the collection if successful; returns NULL if there is no next node.

Description

The iterator initially points before the first node in the list so that the first call to the **MIXmlGetNextAttribute**( ) function returns the first node in the list. This functions returns NULL when the current node is the last node or there are no items in the list.

The caller has to dispose of the returned handle by calling **MIXmlNodeDestroy( ) procedure** when the handle is no longer in use.

See Also:

MIXmlGetAttributeList( ) function, **MIXmlNodeDestroy( ) procedure**

MIXmlGetNextNode( ) function

Purpose

Returns the next node in the collection.

Syntax

MIXmlGetNextNode(ByVal *hXMLNodeList* As MIXmlNodeList) As MIXmlNode

hXMLNodeList is the MIXmlNodeList object handle.

Return Value

A handle to the MIXmlNode object which refers to the next node in the collection represented by, *hXMLNodeList*, if successful; returns NULL if there is no next node.

Description

The iterator initially points before the first node in the list so that the first call to the **MIXmlGetNextNode( )** function returns the first node in the list. This functions returns NULL when the current node is the last node or there are no items in the list.

The caller has to dispose of the returned handle by calling the **MIXmlNodeDestroy( ) procedure** when the handle is no longer in use.

See Also:

MIXmlNodeDestroy( ) procedure, **MIXmlSelectNodes( ) function**, **MIXmlGetChildList( ) function**

MIXmlNodeDestroy( ) procedure

Purpose

Disposes of the MIXmlNode object and frees its memory.

Syntax

MIXmlNodeDestroy(ByVal *hXMLNode* As MIXmlNode)

hXMLNode is the MIXmlNode object handle to be disposed of.

Description

The caller has to call this function to free a MIXmlNode object handle obtained, such as by calling **MIXmlDocumentGetRootNode( ) function**, when the handle is no longer in use.

See Also:

MIXmlDocumentDestroy( ) procedure

MIXmlNodeGetAttributeValue( ) function

Purpose

Retrieves the text associated with the specified name.

Syntax

```
MIXmlNodeGetAttributeValue( ByVal hXMLNode As MIXmlNode,
   ByVal strAttributeName As String, pValue As String,
   ByVal nLen As Integer ) As SmallInt
```

hXMLNode is the MIXmlNode object handle.

strAttributeName is a string specifying the name of the attribute.

pValue is a reference to a string that receives the node value of the specified attribute.

nLen is the size of the buffer referenced by *pValue*.

Return Value

Nonzero if successful; otherwise 0.

Description

MIXmlNodeGetAttributeValue( ) first finds out if there is a valid MIXmlNamedNodeMap object with the given node, *hXMLNode*. As it is stated in **MIXmlGetAttributeList( ) function**, this object only contains the nodes which can return attributes (Element, Entity, and Notation nodes). When there is a valid MIXmlNamedNodeMap object and the specified name is found in the object, its node value will fill in *pValue*.

See Also:

MIXmlGetAttributeList( ) function, **MIXmlNodeGetValue( ) function**

MIXmlNodeGetFirstChild( ) function

Purpose

Retrieves the first child of the given node instance.

Syntax

```
MIXmlNodeGetFirstChild( ByVal hXMLNode As MIXmlNode ) As MIXmlNode
```

hXMLNode is the MIXmlNode object handle.

Return Value

A handle to the MIXmlNode object which is the first child of the given node instance, *hXMLNode*, if successful; otherwise NULL.

Description

MIXmlNodeGetFirstChild( ) gets a handle to a MIXmlNode object that is the first child of the given node instance. It returns NULL if no child exists.

The caller has to dispose of the handle by calling **MIXmlNodeDestroy( ) procedure** when the handle is no longer in use.

See Also:

MIXmlNodeDestroy( ) procedure, **MIXmlNodeGetParent( ) function**

MIXmlNodeGetName( ) function

Purpose

Gets the node name of the given node instance.

Syntax

```
MIXmlNodeGetName( ByVal hXMLNode As MIXmlNode, pName As String,
   ByVal nLen As Integer ) As SmallInt
```

hXMLNode is the MIXmlNode object handle.

pName is a reference to a string that receives the node name, which varies depending on the node type.

nLen is the size of the buffer referenced by *pName*.

Return Value

Nonzero if successful; otherwise 0.

Description

This function is used to get the node name with a given node. The node name is the qualified name for the element, attribute, or entity reference. The node name value varies, depending on the note type.

See Also:

MIXmlDocumentGetRootNode( ) function, **MIXmlNodeGetText( ) function**, **MIXmlNodeGetValue( ) function**.

MIXmlNodeGetParent( ) function

Purpose

Retrieves the parent of the given node instance.

Syntax

```
MIXmlNodeGetParent( ByVal hXMLNode As MIXmlNode ) As MIXmlNode
```

hXMLNode is the MIXmlNode object handle.

Return Value

A handle to the MIXmlNode object which is the parent of the given node instance, *hXMLNode*, if successful; otherwise NULL.

Description

MIXmlNodegetParent( ) gets a handle to a MIXmlNode object that is the parent of the given node instance. It returns NULL if no parent exists.

The caller has to dispose of the handle by calling the **MIXmlNodeDestroy( ) procedure** when the handle is no longer in use.

See Also:

MIXmlNodeDestroy( ) procedure

MIXmlNodeGetText( ) function

Purpose

Gets the text content of the given node or the concatenated text representing the node and its descendants.

Syntax

```
MIXmlNodeGetText( ByVal hXMLNode As MIXmlNode, pText As String,
   ByVal nLen As Integer ) As SmallInt
```

hXMLNode is the MIXmlNode object handle.

pText is a reference to a string that receives the text content of the given node and its descendants. This value varies depending on the value of the note type.

nLen is the size of the buffer referenced by *pText*.

Return Value

Nonzero if successful; otherwise 0.

Description

MIXmlNodegettext( ) is used to get the node text with a given node instance. Its value varies, depending on the node type. For more details and more precise control over text manipulation in an XML document, refer to the Microsoft MSDN library.

See Also:

MIXmlDocumentGetRootNode( ) function, **MIXmlNodeGetName( ) function**, **MIXmlNodeGetValue( ) function**

MIXmlNodeGetValue( ) function

Purpose

Gets the text associated with the given node instance.

Syntax

```
MIXmlNodeGetValue( ByVal hXMLNode As MIXmlNode, pValue As String,
   ByVal nLen As Integer ) As SmallInt
```

hXMLNode is the MIXmlNode object handle.

pValue is a reference to a string that receives the value, which varies depending on the node type.

nLen is the size of the buffer referenced by *pValue*.

Return Value

Nonzero if successful; otherwise 0.

Description

This function is used to get the node value with a given node instance. The node value varies, depending on the node type.

See Also:

MIXmlDocumentGetRootNode( ) function, **MIXmlNodeGetName( ) function**, **MIXmlNodeGetText( ) function**.

MIXmlNodeListDestroy( ) procedure

Purpose

Disposes of the MIXmlNodeList object and frees its memory.

Syntax

```
MIXmlNodeListDestroy( ByVal hXMLNodeList As MIXmlNodeList )
```

hXMLNodeList is the MIXmlNodeList object handle to be disposed of.

Description

Use **MIXmlNodeListDestroy( )** to free the MIXmlNodeList handle obtained with functions such as the **MIXmlSelectNodes( ) function**, and the **MIXmlGetChildList( ) function**, when the MIXmlNodeList handle is no longer in use.

See Also:

MIXmlDocumentDestroy( ) procedure, **MIXmlGetChildList( ) function**, **MIXmlSelectNodes( ) function**

MIXmlSCDestroy( ) procedure

Purpose

Disposes of the MIXMLSchemaCollection object and frees its memory.

Syntax

```
MIXmlSCDestroy( ByVal hXMLSchemaCollection As MIXMLSchemaCollection )
```

hXMLSchemaCollection is the MIXMLSchemaCollection object handle to be disposed of.

Description

Use **MIXmlSCDestroy( )** to free the MIXMLSchemaCollection handle obtained with a function such as the **MIXmlDocumentGetNamespaces( ) function**, when the handle is no longer in use.

See Also:

MIXmlDocumentGetNamespaces( ) function

MIXmlSCGetLength( ) function

Purpose

Gets the number of namespaces currently in the collection.

Syntax

```
MIXmlSCGetLength( ByVal hXMLSchemaCollection As MIXMLSchemaCollection )
   As Integer
```

hXMLSchemaCollection is the MIXMLSchemaCollection object handle.

Return Value

The number of namespaces currently in the collection.

Description

MIXmlSCGetLength( ) allows you to retrieve the number of namespaces currently in the collection.

See Also:

MIXmlDocumentGetNamespaces( ) function, **MIXmlSCGetNamespace( ) function**

MIXmlSCGetNamespace( ) function

Purpose

Gets the namespace at the specified index.

Syntax

```
MIXmlSCGetNamespace( ByVal hXMLSchemaCollection As MIXMLSchemaCollection,
   ByVal index As Integer, pNamespace As String, ByVal nLen As Integer )
   As SmallInt
```

hXMLSchemaCollection is the MIXMLSchemaCollection object handle.

index is an integer that indicates the index between 0 and count -1.

pNamespace is a reference to a string that receives the name of the namespace.

nLen is the size of the buffer referenced by *pNamespace*.

Return Value

Nonzero if successful; otherwise 0. To determine the cause of the failure, call the **MIGetErrorMessage( ) function**.

Description

MIXmlSCGetNamespace( ) allows you to iterate through the collection to discover its contents.

See Also:

MIXmlDocumentGetNamespaces( ) function, **MIXmlSCGetLength( ) function**

MIXmlSelectNodes( ) function

Purpose

Applies the specified pattern-matching operation to this node's context and returns the list of matching nodes as an MIXmlNodeList object.

Syntax

```
MIXmlSelectNodes( ByVal hXMLNode As MIXmlNode, ByVal strPattern As String
   ) As MIXmlNodeList
```

hXMLNode is the MIXmlNode object handle.

strPattern is a string specifying an XPath expression.

Return Value

A handle to an MIXmlNodeList object. It is the collection of nodes selected by applying the given pattern-matching operation. If no nodes are selected, returns an empty collection. NULL is returned if it fails.

Description

MIXmlSelectNodes( ) is used to get a collection of matching nodes as an MIXmlNodeList object with the specified pattern-matching operation. The **MIXmlSelectNodes( )** is similar to **MIXmlSelectSingleNode( ) function**, but returns a list of all of the matching nodes rather than the first matching node.

The caller has to dispose of the handle by calling **MIXmlNodeListDestroy( ) procedure** when the handle is no longer in use.

See Also:

MIXmlNodeListDestroy( ) procedure, **MIXmlSelectSingleNode( ) function**, **MIXmlGetChildList( ) function**

MIXmlSelectSingleNode( ) function

Purpose

Applies the specified pattern-matching operation to this node's context and returns the first matching node as an MIXmlNode object.

Syntax

```
MIXmlSelectSingleNode( ByVal hXMLNode As MIXmlNode,
   ByVal strPattern As String ) As MIXmlNode
```

hXMLNode is the MIXmlNode object handle.

strPattern is a string specifying an XPath expression.

Return Value

A handle to the MIXmlNode object. Returns the first node that matches the given pattern-matching operation. If no nodes match the expression, returns a NULL value.

Description

MIXmlSelectSingleNode( ) gets a handle to an MIXmlNode object that is the first matching node with the given pattern-matching operation. It returns NULL if no child exists.
MIXmlSelectSingleNode( ) is similar to the **MIXmlSelectNodes( ) function**, but returns the first matching node rather than a list of all the matching nodes.

The caller has to dispose of the handle by calling the **MIXmlNodeDestroy( ) procedure** when the handle is no longer in use.

MIXmlNodeDestroy( ) procedure, **MIXmlSelectNodes( ) function**

C

Character Code Table

The following table summarizes the displayable portion of the Windows Latin 1 character set. The range of characters from 32 (space) to 126 (tilde) are identical in most other character sets as well. Special characters of interest: 9 is a tab, 10 is a line feed, 12 is a form feed and 13 is a carriage return.

32		64	@	96	`	128	■	160		192	À	224	à
33	!	65	A	97	a	129	■	161	¡	193	Á	225	á
34	"	66	B	98	b	130	■	162	¢	194	Â	226	â
35	#	67	C	99	c	131	■	163	£	195	Ã	227	ã
36	$	68	D	100	d	132	■	164	¤	196	Ä	228	ä
37	%	69	E	101	e	133	■	165	¥	197	Å	229	å
38	&	70	F	102	f	134	■	166	¦	198	Æ	230	æ
39	'	71	G	103	g	135	■	167	§	199	Ç	231	ç
40	(	72	H	104	h	136	■	168	¨	200	È	232	è
41	)	73	I	105	i	137	■	169	©	201	É	233	é
42	*	74	J	106	j	138	■	170	ª	202	Ê	234	ê
43	+	75	K	107	k	139	■	171	«	203	Ë	235	ë
44	,	76	L	108	l	140	■	172	¬	204	Ì	236	ì
45	-	77	M	109	m	141	■	173	-	205	Í	237	í
46	.	78	N	110	n	142	■	174	®	206	Î	238	î
47	/	79	O	111	o	143	■	175	¯	207	Ï	239	ï
48	0	80	P	112	p	144	■	176	°	208	Ð	240	ð
49	1	81	Q	113	q	145	‘	177	±	209	Ñ	241	ñ
50	2	82	R	114	r	146	’	178	²	210	Ò	242	ò
51	3	83	S	115	s	147	■	179	³	211	Ó	243	ó
52	4	84	T	116	t	148	■	180	´	212	Ô	244	ô
53	5	85	U	117	u	149	■	181	µ	213	Õ	245	õ
54	6	86	V	118	v	150	■	182	¶	214	Ö	246	ö
55	7	87	W	119	w	151	■	183	·	215	×	247	÷
56	8	88	X	120	x	152	■	184	¸	216	Ø	248	ø
57	9	89	Y	121	y	153	■	185	¹	217	Ù	249	ù
58	:	90	Z	122	z	154	■	186	º	218	Ú	250	ú
59	;	91	[	123	{	155	■	187	»	219	Û	251	û
60	<	92	\	124	|	156	■	188	¼	220	Ü	252	ü
61	=	93	]	125	}	157	■	189	½	221	Ý	253	ý
62	>	94	^	126	~	158	■	190	¾	222	Þ	254	þ
63	?	95	_	127		159	■	191	¿	223	ß	255	ÿ

D

Summary of Operators

Operators act on one or more values to produce a result. Operators can be classified by the data types they use and the type result they produce.

Topics in this Section:

- Numeric Operators 845
- Comparison Operators 845
- Logical Operators 846
- Geographical Operators 846
- Automatic Type Conversions 847
- Wildcards 848

Numeric Operators

The following numeric operators act on two numeric values, producing a numeric result.

Operator	Performs	Example
+	addition	`a + b`
-	subtraction	`a - b`
*	multiplication	`a * b`
/	division	`a / b`
\	integer divide (drop remainder)	`a \ b`
Mod	remainder from integer division	`a Mod b`
^	exponentiation	`a ^ b`

Two of these operators are also used in other contexts. The plus sign acting on a pair of strings concatenates them into a new string value. The minus sign acting on a single number is a negation operator, producing a numeric result. The ampersand also performs string concatenation.

Operator	Performs	Example
-	numeric negation	`- a`
+	string concatenation	`a + b`
&	string concatenation	`a & b`

Comparison Operators

The comparison operators compare two items of the same general type to produce a logical value of TRUE or FALSE. Although you cannot directly compare numeric data with non-numeric data (e.g., string expressions), a comparison expression can compare integer, SmallInt, and float data types. Comparison operators are often used in conditional expressions, such as If…Then.

Operator	Returns TRUE if:	Example
=	a is equal to b	`a = b`
<>	a is not equal to b	`a <> b`
<	a is less than b	`a < b`

Operator	Returns TRUE if:	Example
>	a is greater than b	`a > b`
<=	a is less than or equal to b	`a <= b`
>=	a is greater than or equal to b	`a >= b`

Logical Operators

The logical operators operate on logical values to produce a logical result of TRUE or FALSE:

Operator	Returns TRUE if:	Example
And	both operands are TRUE	`a And b`
Or	either operand is TRUE	`a Or b`
Not	the operand is FALSE	`Not a`

Geographical Operators

The geographic operators act on objects to produce a logical result of TRUE or FALSE:

Operator	Returns TRUE if:	Example
Contains	first object contains the centroid of the second object	`objectA Contains objectB`
Contains Part	first object contains part of the second object	`objectA Contains Part objectB`
Contains Entire	first object contains all of the second object	`objectA Contains Entire objectB`
Within	first object's centroid is within the second object	`objectA Within objectB`
Partly Within	part of the first object is within the second object	`objectA Partly Within objectB`
Entirely Within	the first object is entirely inside the second object	`objectA Entirely Within objectB`
Intersects	the two objects intersect at some point	`objectA Intersects objectB`

Precedence

A special type of operators are parentheses, which enclose expressions within expressions. Proper use of parentheses can alter the order of processing in an expression, altering the default precedence. The table below identifies the precedence of MapBasic operators. Operators which appear on a single row have equal precedence. Operators of higher priority are processed first. Operators of the same precedence are evaluated left to right in the expression (with the exception of exponentiation, which is evaluated right to left).

Precedence of MapBasic operators	Operators
(Highest Priority)	parenthesis
	exponentiation
	negation
	multiplication, division, Mod, integer division
	addition, subtraction
	geographic operators
	comparison operators, Like operator
	Not
	And
(Lowest Priority)	Or

For example, the expression `3 + 4 * 2` produces a result of 11 (multiplication is performed before addition). The altered expression `(3 + 4) * 2` produces 14 (parentheses cause the addition to be performed first). When in doubt, use parentheses.

Automatic Type Conversions

When you create an expression involving data of different types, MapInfo performs automatic type conversion in order to produce meaningful results. For example, if your program subtracts a Date value from another Date value, MapBasic will calculate the result as an integer value (representing the number of days between the two dates).

The table below summarizes the rules that dictate MapBasic's automatic type conversions. Within this chart, the keyword *Integer* represents an integer value, which can be an integer variable, a SmallInt variable, or an integer constant. The keyword *Number* represents a numeric expression which is not necessarily an integer.

Operator	Combination of Operands	Result
+	Date + Number	Date
	Number + Date	Date
	Integer + Integer	Integer
	Number + Number	Float
	Other + Other	String
-	Date - Number	Date
	Date - Date	Integer
	Integer - Integer	Integer
	Number - Number	Float
*	Integer * Integer	Integer
	Number * Number	Float
/	Number / Number	Float
\	Number \ Number	Integer
MOD	Number MOD Number	Integer
^	Number ^ Number	Float

Wildcards

The LIKE operator uses the percent (%) and underscore (_) characters as wildcards.

Wildcard	Performs	Example
_	matches one character	`select * from table where Name Like "New%" into Selection`
%	matches zero or more characters	`select * from table where Name Like "New Yor_" into Selection`

E

MapBasic Definitions File

The following MAPBASIC.DEF file lists definitions and defaults useful when programming in MapBasic. This file is installed in the MapBasic directory.

MapBasic.DEF File

```
'============================================================================
' MapInfo version 11.0 - System defines
'----------------------------------------------------------------------------
' This file contains defines useful when programming in the MapBasic
' language.  There are three versions of this file:
'     MAPBASIC.DEF  - MapBasic syntax
'     MAPBASIC.BAS  - Visual Basic syntax
'     MAPBASIC.H    - C/C++ syntax
'----------------------------------------------------------------------------
' The defines in this file are organized into the following sections:
'     General Purpose defines:
'       macros, logical constants, angle conversion, colors, string length
'     BrowserInfo() defines
'     ButtonPadInfo() defines
'     ColumnInfo() and column type defines
'     CommandInfo() and task switch defines
'     DateWindow() defines
'     FileAttr() and file access mode defines
'     GetFolderPath$() defines
'     GetPreferencePath$() defines
'     IntersectNodes() parameters
'     LabelInfo() defines
'     GroupLayerInfo() defines
'     LayerListInfo() defines
'     LayerInfo(), display mode, label property, layer type, hotlink defines
'     LegendInfo() and legend orientation defines
'     LegendFrameInfo() and frame type defines
'     LegendStyleInfo() defines
'     LibraryServiceInfo() defines
'     LocateFile$() defines
'     Map3DInfo() defines
'     MapperInfo(), display mode, calculation type, and clip type defines
'     MenuItemInfoByID() and MenuItemInfoByHandler() defines
'     ObjectGeography() defines
```

```
'     ObjectInfo() and object type defines
'     PrismMapInfo() defines
'     SearchInfo() defines
'     SelectionInfo() defines
'     Server statement and function defines
'     SessionInfo() defines
'     Set Next Document Style defines
'     StringCompare() return values
'     StyleAttr() defines
'     SystemInfo(), platform, and version defines
'     TableInfo() and table type defines
'     WindowInfo(), window type and state, and print orientation defines
'     Abbreviated list of error codes
'     Backward Compatibility defines
'=============================================================================
' MAPBASIC.DEF is converted into MAPBASIC.H by doing the following:
'   - concatenate MAPBASIC.DEF and MENU.DEF into MAPBASIC.H
'   - search & replace "'" at begining of a line with "//"
'   - search & replace "Define" at begining of a line with "#define"
'   - delete the following sections:
'         * General Purpose defines:
'              Macros, Logical Constants, Angle Conversions
'         * Abbreviated list of error codes
'         * Backward Compatibility defines
'         * Menu constants whose names have changed
'         * Obsolete menu items
'=============================================================================
' MAPBASIC.DEF is converted into MAPBASIC.BAS by doing the following:
'   - concatenate MAPBASIC.DEF and MENU.DEF into MAPBASIC.BAS
'   - search & replace "Define <name>" with "Global Const <name> ="
'     e.g. "<Define {[!-z]+} +{[!-z]}" with "Global Const \0 = \1" with Brief
'   - delete the following sections:
'         * General Purpose defines:
'             Macros, Logical Constants, Angle Conversions
'         * Abbreviated list of error codes
'         * Backward Compatibility defines
```

```
'         * Menu constants whose names have changed
'         * Obsolete menu items
'=============================================================================

'=============================================================================
' General Purpose defines
'=============================================================================
'-----------------------------------------------------------------------------
' Macros
'-----------------------------------------------------------------------------
Define CLS                                  Print Chr$(12)

'-----------------------------------------------------------------------------
' Logical constants
'-----------------------------------------------------------------------------
Define TRUE                                 1
Define FALSE                                0

'-----------------------------------------------------------------------------
' Angle conversion
'-----------------------------------------------------------------------------
Define DEG_2_RAD                            0.01745329252
Define RAD_2_DEG                            57.29577951

'-----------------------------------------------------------------------------
' Time conversion
'-----------------------------------------------------------------------------
Define SECONDS_PER_DAY                      86400

'-----------------------------------------------------------------------------
' Colors
'-----------------------------------------------------------------------------
Define BLACK                                0
Define WHITE                                16777215
Define RED                                  16711680
Define GREEN                                65280
```

```
Define BLUE                                     255
Define CYAN                                     65535
Define MAGENTA                                  16711935
Define YELLOW                                   16776960

'------------------------------------------------------------------------
'Maximum length for character string
'------------------------------------------------------------------------
Define MAX_STRING_LENGTH                        32767

'========================================================================
' BrowserInfo() defines
'========================================================================
Define BROWSER_INFO_NROWS                       1
Define BROWSER_INFO_NCOLS                       2
Define BROWSER_INFO_CURRENT_ROW                 3
Define BROWSER_INFO_CURRENT_COLUMN              4
Define BROWSER_INFO_CURRENT_CELL_VALUE          5

'========================================================================
' ButtonPadInfo() defines
'========================================================================
Define BTNPAD_INFO_FLOATING                     1
Define BTNPAD_INFO_WIDTH                        2
Define BTNPAD_INFO_NBTNS                        3
Define BTNPAD_INFO_X                            4
Define BTNPAD_INFO_Y                            5
Define BTNPAD_INFO_WINID                        6
Define BTNPAD_INFO_DOCK_POSITION                7

'========================================================================
' New as per MI Pro 10.5.
' Codes returned from ButtonPadInfo() when 'BTNPAD_INFO_DOCK_POSITION' code
' is used to inquiry about the tool bar position
'========================================================================
Define BBTNPAD_INFO_DOCK_NONE                   0
```

```
Define BTNPAD_INFO_DOCK_LEFT                    1
Define BTNPAD_INFO_DOCK_TOP                     2
Define BTNPAD_INFO_DOCK_RIGHT                   3
Define BTNPAD_INFO_DOCK_BOTTOM                  4

'=============================================================================
' ColumnInfo() defines
'=============================================================================
Define COL_INFO_NAME                            1
Define COL_INFO_NUM                             2
Define COL_INFO_TYPE                            3
Define COL_INFO_WIDTH                           4
Define COL_INFO_DECPLACES                       5
Define COL_INFO_INDEXED                         6
Define COL_INFO_EDITABLE                        7

'-----------------------------------------------------------------------------
' Column type defines, returned by ColumnInfo() for COL_INFO_TYPE
'-----------------------------------------------------------------------------
Define COL_TYPE_CHAR                            1
Define COL_TYPE_DECIMAL                         2
Define COL_TYPE_INTEGER                         3
Define COL_TYPE_SMALLINT                        4
Define COL_TYPE_DATE                            5
Define COL_TYPE_LOGICAL                         6
Define COL_TYPE_GRAPHIC                         7
Define COL_TYPE_FLOAT                           8
Define COL_TYPE_TIME                            37
Define COL_TYPE_DATETIME                        38

'=============================================================================
' CommandInfo() defines
'=============================================================================
Define CMD_INFO_X                               1
Define CMD_INFO_Y                               2
Define CMD_INFO_SHIFT                           3
```

```
Define CMD_INFO_CTRL                        4
Define CMD_INFO_X2                          5
Define CMD_INFO_Y2                          6
Define CMD_INFO_TOOLBTN                     7
Define CMD_INFO_MENUITEM                    8
Define CMD_INFO_WIN                         1
Define CMD_INFO_SELTYPE                     1
Define CMD_INFO_ROWID                       2
Define CMD_INFO_INTERRUPT                   3
Define CMD_INFO_STATUS                      1
Define CMD_INFO_MSG                         1000
Define CMD_INFO_DLG_OK                      1
Define CMD_INFO_DLG_DBL                     1
Define CMD_INFO_FIND_RC                     3
Define CMD_INFO_FIND_ROWID                  4
Define CMD_INFO_XCMD                        1
Define CMD_INFO_CUSTOM_OBJ                  1
Define CMD_INFO_TASK_SWITCH                 1
Define CMD_INFO_EDIT_TABLE                  1
Define CMD_INFO_EDIT_STATUS                 2
Define CMD_INFO_EDIT_ASK                    1
Define CMD_INFO_EDIT_SAVE                   2
Define CMD_INFO_EDIT_DISCARD                3
Define CMD_INFO_HL_WINDOW_ID                17
Define CMD_INFO_HL_TABLE_NAME               18
Define CMD_INFO_HL_ROWID                    19
Define CMD_INFO_HL_LAYER_ID                 20
Define CMD_INFO_HL_FILE_NAME                21

'-------------------------------------------------------------------------
' Task Switches, returned by CommandInfo() for CMD_INFO_TASK_SWITCH
'-------------------------------------------------------------------------
Define SWITCHING_OUT_OF_MAPINFO             0
Define SWITCHING_INTO_MAPINFO               1

'=========================================================================
```

```
' DateWindow() defines
'=============================================================================
Define DATE_WIN_SESSION                         1
Define DATE_WIN_CURPROG                         2

'=============================================================================
' FileAttr() defines
'=============================================================================
Define FILE_ATTR_MODE                           1
Define FILE_ATTR_FILESIZE                       2

'-----------------------------------------------------------------------------
' File Access Modes, returned by FileAttr() for FILE_ATTR_MODE
'-----------------------------------------------------------------------------
Define MODE_INPUT                               0
Define MODE_OUTPUT                              1
Define MODE_APPEND                              2
Define MODE_RANDOM                              3
Define MODE_BINARY                              4

'=============================================================================
' GetFolderPath$() defines
'=============================================================================

Define FOLDER_MI_APPDATA                        -1
Define FOLDER_MI_LOCAL_APPDATA                  -2
Define FOLDER_MI_PREFERENCE                     -3
Define FOLDER_MI_COMMON_APPDATA                 -4
Define FOLDER_APPDATA                           26
Define FOLDER_LOCAL_APPDATA                     28
Define FOLDER_COMMON_APPDATA                    35
Define FOLDER_COMMON_DOCS                       46
Define FOLDER_MYDOCS                            5
Define FOLDER_MYPICS                            39

'=============================================================================
```

```
' GetPreferencePath$(), GetCurrentPath$(), and Set Path defines
'============================================================================

Define PREFERENCE_PATH_TABLE          0
Define PREFERENCE_PATH_WORKSPACE      1
Define PREFERENCE_PATH_MBX            2
Define PREFERENCE_PATH_IMPORT         3
Define PREFERENCE_PATH_SQLQUERY       4
Define PREFERENCE_PATH_THEMETHEMPLATE 5
Define PREFERENCE_PATH_MIQUERY        6
Define PREFERENCE_PATH_NEWGRID        7
Define PREFERENCE_PATH_CRYSTAL        8
Define PREFERENCE_PATH_GRAPHSUPPORT   9
Define PREFERENCE_PATH_REMOTETABLE    10
Define PREFERENCE_PATH_SHAPEFILE      11
Define PREFERENCE_PATH_WFSTABLE       12
Define PREFERENCE_PATH_WMSTABLE       13

'============================================================================
' IntersectNodes() defines
'============================================================================
Define INCL_CROSSINGS                          1
Define INCL_COMMON                             6
Define INCL_ALL                                7

'============================================================================
' LabelInfo() defines
'============================================================================
Define LABEL_INFO_OBJECT                       1
Define LABEL_INFO_POSITION                     2
Define LABEL_INFO_ANCHORX                      3
Define LABEL_INFO_ANCHORY                      4
Define LABEL_INFO_OFFSET                       5
Define LABEL_INFO_ROWID                        6
Define LABEL_INFO_TABLE                        7
Define LABEL_INFO_EDIT                         8
```

```
Define LABEL_INFO_EDIT_VISIBILITY               9
Define LABEL_INFO_EDIT_ANCHOR                   10
Define LABEL_INFO_EDIT_OFFSET                   11
Define LABEL_INFO_EDIT_FONT                     12
Define LABEL_INFO_EDIT_PEN                      13
Define LABEL_INFO_EDIT_TEXT                     14
Define LABEL_INFO_EDIT_TEXTARROW                15
Define LABEL_INFO_EDIT_ANGLE                    16
Define LABEL_INFO_EDIT_POSITION                 17
Define LABEL_INFO_EDIT_TEXTLINE                 18
Define LABEL_INFO_SELECT                        19
Define LABEL_INFO_DRAWN                         20
Define LABEL_INFO_ORIENTATION                   21

'============================================================================
' Codes passed to the GroupLayerInfo function to get info about a group layer.
'============================================================================
Define GROUPLAYER_INFO_NAME                     1
Define GROUPLAYER_INFO_LAYERLIST_ID             2
Define GROUPLAYER_INFO_DISPLAY                  3
Define GROUPLAYER_INFO_LAYERS                   4
Define GROUPLAYER_INFO_ALL_LAYERS               5
Define GROUPLAYER_INFO_TOPLEVEL_LAYERS          6
Define GROUPLAYER_INFO_PARENT_GROUP_ID          7

'============================================================================
' Values returned by GroupLayerInfo() for GROUPLAYER_INFO_DISPLAY. These
' defines correspond to the MapBasic defines in MAPBASIC.DEF. If you alter
' these you must alter MAPBASIC.DEF.
'============================================================================
Define GROUPLAYER_INFO_DISPLAY_OFF              0
Define GROUPLAYER_INFO_DISPLAY_ON               1

'****************************************************************************
' Codes passed to the LayerListInfo function to help enumerating all layers in a
Map.
```

```
'************************************************************************
Define LAYERLIST_INFO_TYPE                      1
Define LAYERLIST_INFO_NAME                      2
Define LAYERLIST_INFO_LAYER_ID                  3
Define LAYERLIST_INFO_GROUPLAYER_ID             4

'************************************************************************
' Values returned by LayerListInfo() for LAYERLIST_INFO_TYPE. These
' defines correspond to the MapBasic defines in MAPBASIC.DEF. If you alter
' these you must alter MAPBASIC.DEF.
'************************************************************************

Define LAYERLIST_INFO_TYPE_LAYER                0
Define LAYERLIST_INFO_TYPE_GROUP                1

'========================================================================
' LayerInfo() defines
'========================================================================
Define LAYER_INFO_NAME                          1
Define LAYER_INFO_EDITABLE                      2
Define LAYER_INFO_SELECTABLE                    3
Define LAYER_INFO_ZOOM_LAYERED                  4
Define LAYER_INFO_ZOOM_MIN                      5
Define LAYER_INFO_ZOOM_MAX                      6
Define LAYER_INFO_COSMETIC                      7
Define LAYER_INFO_PATH                          8
Define LAYER_INFO_DISPLAY                       9
Define LAYER_INFO_OVR_LINE                      10
Define LAYER_INFO_OVR_PEN                       11
Define LAYER_INFO_OVR_BRUSH                     12
Define LAYER_INFO_OVR_SYMBOL                    13
Define LAYER_INFO_OVR_FONT                      14
Define LAYER_INFO_LBL_EXPR                      15
Define LAYER_INFO_LBL_LT                        16
Define LAYER_INFO_LBL_CURFONT                   17
Define LAYER_INFO_LBL_FONT                      18
```

```
Define LAYER_INFO_LBL_PARALLEL                  19
Define LAYER_INFO_LBL_POS                       20
Define LAYER_INFO_ARROWS                        21
Define LAYER_INFO_NODES                         22
Define LAYER_INFO_CENTROIDS                     23
Define LAYER_INFO_TYPE                          24
Define LAYER_INFO_LBL_VISIBILITY                25
Define LAYER_INFO_LBL_ZOOM_MIN                  26
Define LAYER_INFO_LBL_ZOOM_MAX                  27
Define LAYER_INFO_LBL_AUTODISPLAY               28
Define LAYER_INFO_LBL_OVERLAP                   29
Define LAYER_INFO_LBL_DUPLICATES                30
Define LAYER_INFO_LBL_OFFSET                    31
Define LAYER_INFO_LBL_MAX                       32
Define LAYER_INFO_LBL_PARTIALSEGS               33
Define LAYER_INFO_HOTLINK_EXPR                  34
Define LAYER_INFO_HOTLINK_MODE                  35
Define LAYER_INFO_HOTLINK_RELATIVE              36
Define LAYER_INFO_HOTLINK_COUNT                 37
Define LAYER_INFO_LBL_ORIENTATION               38
Define LAYER_INFO_LAYER_ALPHA                   39
Define LAYER_INFO_LAYER_TRANSLUCENCY            40
Define LAYER_INFO_LABEL_ALPHA                   41
Define LAYER_INFO_LAYERLIST_ID                  42
Define LAYER_INFO_PARENT_GROUP_ID               43
'Code 44 - 52 are for override styel & label
Define LAYER_INFO_OVR_STYLE_COUNT               44
Define LAYER_INFO_OVR_LBL_COUNT                 45
Define LAYER_INFO_OVR_STYLE_CURRENT             46
Define LAYER_INFO_OVR_LBL_CURRENT               47
Define LAYER_INFO_OVR_LINE_COUNT                48
Define LAYER_INFO_OVR_PEN_COUNT                 49
Define LAYER_INFO_OVR_BRUSH_COUNT               50
Define LAYER_INFO_OVR_SYMBOL_COUNT              51
Define LAYER_INFO_OVR_FONT_COUNT                52
```

```
'------------------------------------------------------------------------
' Values returned by LayerInfo() for LAYER_INFO_LABEL_ORIENTATION and
' LABEL_INFO_ORIENTATION.
'------------------------------------------------------------------------
Define LAYER_INFO_LABEL_ORIENT_HORIZONTAL       0
Define LAYER_INFO_LABEL_ORIENT_PARALLEL         1
Define LAYER_INFO_LABEL_ORIENT_CURVED           2

'------------------------------------------------------------------------
' Display Modes, returned by LayerInfo() for LAYER_INFO_DISPLAY
'------------------------------------------------------------------------
Define LAYER_INFO_DISPLAY_OFF                   0
Define LAYER_INFO_DISPLAY_GRAPHIC               1
Define LAYER_INFO_DISPLAY_GLOBAL                2
Define LAYER_INFO_DISPLAY_VALUE                 3

'------------------------------------------------------------------------
' Label Linetypes, returned by LayerInfo() for LAYER_INFO_LBL_LT
'------------------------------------------------------------------------
Define LAYER_INFO_LBL_LT_NONE                   0
Define LAYER_INFO_LBL_LT_SIMPLE                 1
Define LAYER_INFO_LBL_LT_ARROW                  2

'------------------------------------------------------------------------
' Label Positions, returned by LayerInfo() for LAYER_INFO_LBL_POS
'------------------------------------------------------------------------
Define LAYER_INFO_LBL_POS_CC                    0
Define LAYER_INFO_LBL_POS_TL                    1
Define LAYER_INFO_LBL_POS_TC                    2
Define LAYER_INFO_LBL_POS_TR                    3
Define LAYER_INFO_LBL_POS_CL                    4
Define LAYER_INFO_LBL_POS_CR                    5
Define LAYER_INFO_LBL_POS_BL                    6
Define LAYER_INFO_LBL_POS_BC                    7
Define LAYER_INFO_LBL_POS_BR                    8
```

```
'------------------------------------------------------------------------
' Layer Types, returned by LayerInfo() for LAYER_INFO_TYPE
'------------------------------------------------------------------------
Define LAYER_INFO_TYPE_NORMAL                   0
Define LAYER_INFO_TYPE_COSMETIC                 1
Define LAYER_INFO_TYPE_IMAGE                    2
Define LAYER_INFO_TYPE_THEMATIC                 3
Define LAYER_INFO_TYPE_GRID                     4
Define LAYER_INFO_TYPE_WMS                      5
Define LAYER_INFO_TYPE_TILESERVER               6

'------------------------------------------------------------------------
' Label visibility modes, from LayerInfo() for LAYER_INFO_LBL_VISIBILITY
'------------------------------------------------------------------------
Define LAYER_INFO_LBL_VIS_OFF                   1
Define LAYER_INFO_LBL_VIS_ZOOM                  2
Define LAYER_INFO_LBL_VIS_ON                    3

'------------------------------------------------------------------------
' Code passed to StyleOverrideInfo function to get override style information
'------------------------------------------------------------------------
Define STYLE_OVR_INFO_NAME                      1
Define STYLE_OVR_INFO_VISIBILITY                2
Define STYLE_OVR_INFO_ZOOM_MIN                  3
Define STYLE_OVR_INFO_ZOOM_MAX                  4
Define STYLE_OVR_INFO_ARROWS                    5
Define STYLE_OVR_INFO_NODES                     6
Define STYLE_OVR_INFO_CENTROIDS                 7
Define STYLE_OVR_INFO_ALPHA                     8
Define STYLE_OVR_INFO_TRANSLUCENCY              9
Define STYLE_OVR_INFO_LINE                      10
Define STYLE_OVR_INFO_PEN                       11
Define STYLE_OVR_INFO_BRUSH                     12
Define STYLE_OVR_INFO_SYMBOL                    13
Define STYLE_OVR_INFO_FONT                      14
Define STYLE_OVR_INFO_SYMBOL_COUNT              15
```

```
Define STYLE_OVR_INFO_LINE_COUNT                16
Define STYLE_OVR_INFO_PEN_COUNT                 17
Define STYLE_OVR_INFO_BRUSH_COUNT               18
Define STYLE_OVR_INFO_FONT_COUNT                19

'-------------------------------------------------------------------------
' Possible return value of StyleOverrideInfo for code STYLE_OVR_INFO_VISIBILITY
'-------------------------------------------------------------------------
Define STYLE_OVR_INFO_VIS_OFF                   0
Define STYLE_OVR_INFO_VIS_ON                    1
Define STYLE_OVR_INFO_VIS_OFF_ZOOM              2

'-------------------------------------------------------------------------
' Code passed to LabelOverrideInfo function to get override label information
'-------------------------------------------------------------------------
Define LBL_OVR_INFO_NAME                1
Define LBL_OVR_INFO_VISIBILITY          2
Define LBL_OVR_INFO_ZOOM_MIN            3
Define LBL_OVR_INFO_ZOOM_MAX            4
Define LBL_OVR_INFO_EXPR                5
Define LBL_OVR_INFO_LT                  6
Define LBL_OVR_INFO_FONT                7
Define LBL_OVR_INFO_PARALLEL            8
Define LBL_OVR_INFO_POS                 9
Define LBL_OVR_INFO_OVERLAP             10
Define LBL_OVR_INFO_DUPLICATES          11
Define LBL_OVR_INFO_OFFSET              12
Define LBL_OVR_INFO_MAX                 13
Define LBL_OVR_INFO_PARTIALSEGS         14
Define LBL_OVR_INFO_ORIENTATION         15
Define LBL_OVR_INFO_ALPHA               16
Define LBL_OVR_INFO_AUTODISPLAY         17
Define LBL_OVR_INFO_POS_RETRY           18
Define LBL_OVR_INFO_LINE_PEN            19
Define LBL_OVR_INFO_PERCENT_OVER        20
```

```
'------------------------------------------------------------------------
' Possible return value of LabelOverrideInfo for code LBL_OVR_INFO_VISIBILITY
'------------------------------------------------------------------------
Define LBL_OVR_INFO_VIS_OFF            0
Define LBL_OVR_INFO_VIS_ON             1
Define LBL_OVR_INFO_VIS_OFF_ZOOM       2

'------------------------------------------------------------------------
' LayerControlInfo() defines
'------------------------------------------------------------------------
Define LC_INFO_SEL_COUNT                        1

'------------------------------------------------------------------------
' LayerControlSelectionInfo() defines
'------------------------------------------------------------------------
Define LC_SEL_INFO_NAME                         1
Define LC_SEL_INFO_TYPE                         2
Define LC_SEL_INFO_MAPWIN_ID                    3
Define LC_SEL_INFO_LAYER_ID                     4
Define LC_SEL_INFO_OVR_ID                       5

'------------------------------------------------------------------------
' Values returned by LayerControlSelectionInfo() for LC_SEL_INFO_TYPE.
'------------------------------------------------------------------------
Define LC_SEL_INFO_TYPE_MAP                     0
Define LC_SEL_INFO_TYPE_LAYER                   1
Define LC_SEL_INFO_TYPE_GROUPLAYER              2
Define LC_SEL_INFO_TYPE_STYLE_OVR               3
Define LC_SEL_INFO_TYPE_LABEL_OVR               4

'------------------------------------------------------------------------
' HotlinkInfo() defines'-------------------------------------------------
--------------------
Define HOTLINK_INFO_EXPR                        1
Define HOTLINK_INFO_MODE                        2
Define HOTLINK_INFO_RELATIVE                    3
```

```
Define HOTLINK_INFO_ENABLED                     4
Define HOTLINK_INFO_ALIAS                       5

'-------------------------------------------------------------------------
' Hotlink activation modes, from LayerInfo() for LAYER_INFO_HOTLINK_MODE
'-------------------------------------------------------------------------
Define HOTLINK_MODE_LABEL                       0
Define HOTLINK_MODE_OBJ                         1
Define HOTLINK_MODE_BOTH                        2

'=========================================================================
' LegendInfo() defines
'=========================================================================
Define LEGEND_INFO_MAP_ID                       1
Define LEGEND_INFO_ORIENTATION                  2
Define LEGEND_INFO_NUM_FRAMES                   3
Define LEGEND_INFO_STYLE_SAMPLE_SIZE            4

'=========================================================================
' Orientation codes, returned by LegendInfo() for LEGEND_INFO_ORIENTATION
'=========================================================================
Define ORIENTATION_PORTRAIT                     1
Define ORIENTATION_LANDSCAPE                    2
Define ORIENTATION_CUSTOM                       3

'-------------------------------------------------------------------------
' Style sample codes, from LegendInfo() for LEGEND_INFO_STYLE_SAMPLE_SIZE
'-------------------------------------------------------------------------
Define STYLE_SAMPLE_SIZE_SMALL                  0
Define STYLE_SAMPLE_SIZE_LARGE                  1

'=========================================================================
' LegendFrameInfo() defines
'=========================================================================
Define FRAME_INFO_TYPE                          1
Define FRAME_INFO_MAP_LAYER_ID                  2
```

```
Define FRAME_INFO_REFRESHABLE                   3
Define FRAME_INFO_POS_X                         4
Define FRAME_INFO_POS_Y                         5
Define FRAME_INFO_WIDTH                         6
Define FRAME_INFO_HEIGHT                        7
Define FRAME_INFO_TITLE                         8
Define FRAME_INFO_TITLE_FONT                    9
Define FRAME_INFO_SUBTITLE                      10
Define FRAME_INFO_SUBTITLE_FONT                 11
Define FRAME_INFO_BORDER_PEN                    12
Define FRAME_INFO_NUM_STYLES                    13
Define FRAME_INFO_VISIBLE                       14
Define FRAME_INFO_COLUMN                        15
Define FRAME_INFO_LABEL                         16

'=============================================================================
' Frame Types, returned by LegendFrameInfo() for FRAME_INFO_TYPE
'=============================================================================
Define FRAME_TYPE_STYLE                         1
Define FRAME_TYPE_THEME                         2

'=============================================================================
' Geocode Attributes, returned by GeocodeInfo()
'=============================================================================
Define GEOCODE_STREET_NAME                      1
Define GEOCODE_STREET_NUMBER                    2
Define GEOCODE_MUNICIPALITY                     3
Define GEOCODE_MUNICIPALITY2                    4
Define GEOCODE_COUNTRY_SUBDIVISION              5
Define GEOCODE_COUNTRY_SUBDIVISION2             6
Define GEOCODE_POSTAL_CODE                      7

Define GEOCODE_DICTIONARY                       9
Define GEOCODE_BATCH_SIZE                       10
Define GEOCODE_FALLBACK_GEOGRAPHIC              11
Define GEOCODE_FALLBACK_POSTAL                  12
```

```
Define GEOCODE_OFFSET_CENTER                    13
Define GEOCODE_OFFSET_CENTER_UNITS              14
Define GEOCODE_OFFSET_END                       15
Define GEOCODE_OFFSET_END_UNITS                 16
Define GEOCODE_MIXED_CASE                       17
Define GEOCODE_RESULT_MARK_MULTIPLE             18
Define GEOCODE_COUNT_GEOCODED                   19
Define GEOCODE_COUNT_NOTGEOCODED                20
Define GEOCODE_UNABLE_TO_CONVERT_DATA           21
Define GEOCODE_MAX_BATCH_SIZE                   22
Define GEOCODE_PASSTHROUGH                      100

Define DICTIONARY_ALL                           1
Define DICTIONARY_ADDRESS_ONLY                  2
Define DICTIONARY_USER_ONLY                     3
Define DICTIONARY_PREFER_ADDRESS                4
Define DICTIONARY_PREFER_USER                   5

'================================================================================
' ISOGRAM Attributes, returned by IsogramInfo()
'================================================================================
Define ISOGRAM_BANDING                          1
Define ISOGRAM_MAJOR_ROADS_ONLY                 2
Define ISOGRAM_RETURN_HOLES                     3
Define ISOGRAM_MAJOR_POLYGON_ONLY               4
Define ISOGRAM_MAX_OFFROAD_DIST                 5
Define ISOGRAM_MAX_OFFROAD_DIST_UNITS           6
Define ISOGRAM_SIMPLIFICATION_FACTOR            7
Define ISOGRAM_DEFAULT_AMBIENT_SPEED            8
Define ISOGRAM_AMBIENT_SPEED_DIST_UNIT          9
Define ISOGRAM_AMBIENT_SPEED_TIME_UNIT          10
Define ISOGRAM_PROPAGATION_FACTOR               11
Define ISOGRAM_BATCH_SIZE                       12
Define ISOGRAM_POINTS_ONLY                      13
Define ISOGRAM_RECORDS_INSERTED                 14
Define ISOGRAM_RECORDS_NOTINSERTED              15
```

```
Define ISOGRAM_MAX_BATCH_SIZE                   16
Define ISOGRAM_MAX_BANDS                        17
Define ISOGRAM_MAX_DISTANCE                     18
Define ISOGRAM_MAX_DISTANCE_UNITS               19
Define ISOGRAM_MAX_TIME                         20
Define ISOGRAM_MAX_TIME_UNITS                   21

'============================================================================
' LegendStyleInfo() defines
'============================================================================
Define LEGEND_STYLE_INFO_TEXT                   1
Define LEGEND_STYLE_INFO_FONT                   2
Define LEGEND_STYLE_INFO_OBJ                    3

'============================================================================
' LocateFile$() defines
'============================================================================

Define LOCATE_PREF_FILE                         0
Define LOCATE_DEF_WOR                           1
Define LOCATE_CLR_FILE                          2
Define LOCATE_PEN_FILE                          3
Define LOCATE_FNT_FILE                          4
Define LOCATE_ABB_FILE                          5
Define LOCATE_PRJ_FILE                          6
Define LOCATE_MNU_FILE                          7
Define LOCATE_CUSTSYMB_DIR                      8
Define LOCATE_THMTMPLT_DIR                      9
Define LOCATE_GRAPH_DIR                         10
Define LOCATE_WMS_SERVERLIST                    11
Define LOCATE_WFS_SERVERLIST                    12
Define LOCATE_GEOCODE_SERVERLIST                13
Define LOCATE_ROUTING_SERVERLIST                14
Define LOCATE_LAYOUT_TEMPLATE_DIR               15

'============================================================================
```

```
' Map3DInfo() defines
'============================================================================
Define MAP3D_INFO_SCALE                     1
Define MAP3D_INFO_RESOLUTION_X              2
Define MAP3D_INFO_RESOLUTION_Y              3
Define MAP3D_INFO_BACKGROUND                4
Define MAP3D_INFO_UNITS                     5
Define MAP3D_INFO_LIGHT_X                   6
Define MAP3D_INFO_LIGHT_Y                   7
Define MAP3D_INFO_LIGHT_Z                   8
Define MAP3D_INFO_LIGHT_COLOR               9
Define MAP3D_INFO_CAMERA_X                  10
Define MAP3D_INFO_CAMERA_Y                  11
Define MAP3D_INFO_CAMERA_Z                  12
Define MAP3D_INFO_CAMERA_FOCAL_X            13
Define MAP3D_INFO_CAMERA_FOCAL_Y            14
Define MAP3D_INFO_CAMERA_FOCAL_Z            15
Define MAP3D_INFO_CAMERA_VU_1               16
Define MAP3D_INFO_CAMERA_VU_2               17
Define MAP3D_INFO_CAMERA_VU_3               18
Define MAP3D_INFO_CAMERA_VPN_1              19
Define MAP3D_INFO_CAMERA_VPN_2              20
Define MAP3D_INFO_CAMERA_VPN_3              21
Define MAP3D_INFO_CAMERA_CLIP_NEAR          22
Define MAP3D_INFO_CAMERA_CLIP_FAR           23

'============================================================================
' MapperInfo() defines
'============================================================================
Define MAPPER_INFO_ZOOM                     1
Define MAPPER_INFO_SCALE                    2
Define MAPPER_INFO_CENTERX                  3
Define MAPPER_INFO_CENTERY                  4
Define MAPPER_INFO_MINX                     5
Define MAPPER_INFO_MINY                     6
Define MAPPER_INFO_MAXX                     7
```

```
Define MAPPER_INFO_MAXY                          8
Define MAPPER_INFO_LAYERS                        9
Define MAPPER_INFO_EDIT_LAYER                    10
Define MAPPER_INFO_XYUNITS                       11
Define MAPPER_INFO_DISTUNITS                     12
Define MAPPER_INFO_AREAUNITS                     13
Define MAPPER_INFO_SCROLLBARS                    14
Define MAPPER_INFO_DISPLAY                       15
Define MAPPER_INFO_NUM_THEMATIC                  16
Define MAPPER_INFO_COORDSYS_CLAUSE               17
Define MAPPER_INFO_COORDSYS_NAME                 18
Define MAPPER_INFO_MOVE_DUPLICATE_NODES          19
Define MAPPER_INFO_DIST_CALC_TYPE                20
Define MAPPER_INFO_DISPLAY_DMS                   21
Define MAPPER_INFO_COORDSYS_CLAUSE_WITH_BOUNDS   22
Define MAPPER_INFO_CLIP_TYPE                     23
Define MAPPER_INFO_CLIP_REGION                   24
Define MAPPER_INFO_REPROJECTION                  25
Define MAPPER_INFO_RESAMPLING                    26
Define MAPPER_INFO_MERGE_MAP                     27
Define MAPPER_INFO_ALL_LAYERS                    28
Define MAPPER_INFO_GROUPLAYERS                   29
Define MAPPER_INFO_NUM_ADORNMENTS                200
Define MAPPER_INFO_ADORNMENT                     200

'-------------------------------------------------------------------------
' Display Modes, returned by MapperInfo() for MAPPER_INFO_DISPLAY_DMS
'-------------------------------------------------------------------------
Define MAPPER_INFO_DISPLAY_DECIMAL               0
Define MAPPER_INFO_DISPLAY_DEGMINSEC             1
Define MAPPER_INFO_DISPLAY_MGRS                  2
Define MAPPER_INFO_DISPLAY_USNG_WGS84            3
Define MAPPER_INFO_DISPLAY_USNG_NAD27            4

'-------------------------------------------------------------------------
' Display Modes, returned by MapperInfo() for MAPPER_INFO_DISPLAY
```

```
'------------------------------------------------------------------------
Define MAPPER_INFO_DISPLAY_SCALE                0
Define MAPPER_INFO_DISPLAY_ZOOM                 1
Define MAPPER_INFO_DISPLAY_POSITION             2

'------------------------------------------------------------------------
' Distance Calculation Types from MapperInfo() for MAPPER_INFO_DIST_CALC_TYPE
'------------------------------------------------------------------------
Define MAPPER_INFO_DIST_SPHERICAL               0
Define MAPPER_INFO_DIST_CARTESIAN               1

'------------------------------------------------------------------------
' Clip Types, returned by MapperInfo() for MAPPER_INFO_CLIP_TYPE
'------------------------------------------------------------------------
Define MAPPER_INFO_CLIP_DISPLAY_ALL             0
Define MAPPER_INFO_CLIP_DISPLAY_POLYOBJ         1
Define MAPPER_INFO_CLIP_OVERLAY                 2

'========================================================================
' MenuItemInfoByID() and MenuItemInfoByHandler() defines
'========================================================================
Define MENUITEM_INFO_ENABLED                    1
Define MENUITEM_INFO_CHECKED                    2
Define MENUITEM_INFO_CHECKABLE                  3
Define MENUITEM_INFO_SHOWHIDEABLE               4
Define MENUITEM_INFO_ACCELERATOR                5
Define MENUITEM_INFO_TEXT                       6
Define MENUITEM_INFO_HELPMSG                    7
Define MENUITEM_INFO_HANDLER                    8
Define MENUITEM_INFO_ID                         9

'========================================================================
' ObjectGeography() defines
'========================================================================
Define OBJ_GEO_MINX                             1
Define OBJ_GEO_LINEBEGX                         1
```

```
Define OBJ_GEO_POINTX                   1
Define OBJ_GEO_MINY                     2
Define OBJ_GEO_LINEBEGY                 2
Define OBJ_GEO_POINTY                   2
Define OBJ_GEO_MAXX                     3
Define OBJ_GEO_LINEENDX                 3
Define OBJ_GEO_MAXY                     4
Define OBJ_GEO_LINEENDY                 4
Define OBJ_GEO_ARCBEGANGLE              5
Define OBJ_GEO_TEXTLINEX                5
Define OBJ_GEO_ROUNDRADIUS              5
Define OBJ_GEO_CENTROID                 5
Define OBJ_GEO_ARCENDANGLE              6
Define OBJ_GEO_TEXTLINEY                6
Define OBJ_GEO_TEXTANGLE                7
Define OBJ_GEO_POINTZ                   8
Define OBJ_GEO_POINTM                   9

'=============================================================================
' ObjectInfo() defines
'=============================================================================
Define OBJ_INFO_TYPE                    1
Define OBJ_INFO_PEN                     2
Define OBJ_INFO_SYMBOL                  2
Define OBJ_INFO_TEXTFONT                2
Define OBJ_INFO_BRUSH                   3
Define OBJ_INFO_NPNTS                   20
Define OBJ_INFO_TEXTSTRING              3
Define OBJ_INFO_SMOOTH                  4
Define OBJ_INFO_FRAMEWIN                4
Define OBJ_INFO_NPOLYGONS               21
Define OBJ_INFO_TEXTSPACING             4
Define OBJ_INFO_TEXTJUSTIFY             5
Define OBJ_INFO_FRAMETITLE              6
Define OBJ_INFO_TEXTARROW               6
Define OBJ_INFO_FILLFRAME               7
```

```
Define OBJ_INFO_REGION                          8
Define OBJ_INFO_PLINE                           9
Define OBJ_INFO_MPOINT                          10
Define OBJ_INFO_NONEMPTY                        11
Define OBJ_INFO_Z_UNIT_SET                      12
Define OBJ_INFO_Z_UNIT                          13
Define OBJ_INFO_HAS_Z                           14
Define OBJ_INFO_HAS_M                           15

'-------------------------------------------------------------------------
' Object types, returned by ObjectInfo() for OBJ_INFO_TYPE
'-------------------------------------------------------------------------
Define OBJ_TYPE_ARC                             1
Define OBJ_TYPE_ELLIPSE                         2
Define OBJ_TYPE_LINE                            3
Define OBJ_TYPE_PLINE                           4
Define OBJ_TYPE_POINT                           5
Define OBJ_TYPE_FRAME                           6
Define OBJ_TYPE_REGION                          7
Define OBJ_TYPE_RECT                            8
Define OBJ_TYPE_ROUNDRECT                       9
Define OBJ_TYPE_TEXT                            10
Define OBJ_TYPE_MPOINT                          11
Define OBJ_TYPE_COLLECTION                      12

'----------------------*
' RegionInfo() Defines
'----------------------*
Define REGION_INFO_IS_CLOCKWISE                 1

'=========================================================================
' PrismMapInfo() defines
'=========================================================================

Define PRISMMAP_INFO_SCALE                      1
Define PRISMMAP_INFO_BACKGROUND                 4
```

```
Define PRISMMAP_INFO_LIGHT_X                    6
Define PRISMMAP_INFO_LIGHT_Y                    7
Define PRISMMAP_INFO_LIGHT_Z                    8
Define PRISMMAP_INFO_LIGHT_COLOR                9
Define PRISMMAP_INFO_CAMERA_X                   10
Define PRISMMAP_INFO_CAMERA_Y                   11
Define PRISMMAP_INFO_CAMERA_Z                   12
Define PRISMMAP_INFO_CAMERA_FOCAL_X             13
Define PRISMMAP_INFO_CAMERA_FOCAL_Y             14
Define PRISMMAP_INFO_CAMERA_FOCAL_Z             15
Define PRISMMAP_INFO_CAMERA_VU_1                16
Define PRISMMAP_INFO_CAMERA_VU_2                17
Define PRISMMAP_INFO_CAMERA_VU_3                18
Define PRISMMAP_INFO_CAMERA_VPN_1               19
Define PRISMMAP_INFO_CAMERA_VPN_2               20
Define PRISMMAP_INFO_CAMERA_VPN_3               21
Define PRISMMAP_INFO_CAMERA_CLIP_NEAR           22
Define PRISMMAP_INFO_CAMERA_CLIP_FAR            23
Define PRISMMAP_INFO_INFOTIP_EXPR               24

'=============================================================================
' SearchInfo() defines
'=============================================================================
Define SEARCH_INFO_TABLE                        1
Define SEARCH_INFO_ROW                          2

'=============================================================================
' SelectionInfo() defines
'=============================================================================
Define SEL_INFO_TABLENAME                       1
Define SEL_INFO_SELNAME                         2
Define SEL_INFO_NROWS                           3

'=============================================================================
' Server statement and function defines
'=============================================================================
```

```
'------------------------------------------------------------------------
' Return Codes
'------------------------------------------------------------------------
Define SRV_SUCCESS                          0
Define SRV_SUCCESS_WITH_INFO                1
Define SRV_ERROR                            -1
Define SRV_INVALID_HANDLE                   -2
Define SRV_NEED_DATA                        99
Define SRV_NO_MORE_DATA                     100

'------------------------------------------------------------------------
' Special values for the status associated with a fetched value
'------------------------------------------------------------------------
Define SRV_NULL_DATA                        -1
Define SRV_TRUNCATED_DATA                   -2

'------------------------------------------------------------------------
' Server_ColumnInfo() defines
'------------------------------------------------------------------------
Define SRV_COL_INFO_NAME                    1
Define SRV_COL_INFO_TYPE                    2
Define SRV_COL_INFO_WIDTH                   3
Define SRV_COL_INFO_PRECISION               4
Define SRV_COL_INFO_SCALE                   5
Define SRV_COL_INFO_VALUE                   6
Define SRV_COL_INFO_STATUS                  7
Define SRV_COL_INFO_ALIAS                   8

'------------------------------------------------------------------------
' Column types, returned by Server_ColumnInfo() for SRV_COL_INFO_TYPE
'------------------------------------------------------------------------
Define SRV_COL_TYPE_NONE                    0
Define SRV_COL_TYPE_CHAR                    1
Define SRV_COL_TYPE_DECIMAL                 2
Define SRV_COL_TYPE_INTEGER                 3
Define SRV_COL_TYPE_SMALLINT                4
```

```
Define SRV_COL_TYPE_DATE                        5
Define SRV_COL_TYPE_LOGICAL                     6
Define SRV_COL_TYPE_FLOAT                       8
Define SRV_COL_TYPE_FIXED_LEN_STRING            16
Define SRV_COL_TYPE_BIN_STRING                  17

'-------------------------------------------------------------------------
' Server_DriverInfo() Attr defines
'-------------------------------------------------------------------------
Define SRV_DRV_INFO_NAME                        1
Define SRV_DRV_INFO_NAME_LIST                   2
Define SRV_DRV_DATA_SOURCE                      3

'-------------------------------------------------------------------------
' Server_ConnectInfo() Attr defines
'-------------------------------------------------------------------------
Define SRV_CONNECT_INFO_DRIVER_NAME             1
Define SRV_CONNECT_INFO_DB_NAME                 2
Define SRV_CONNECT_INFO_SQL_USER_ID             3
Define SRV_CONNECT_INFO_DS_NAME                 4
Define SRV_CONNECT_INFO_QUOTE_CHAR              5

'-------------------------------------------------------------------------
' Fetch Directions (used by ServerFetch function in some code libraries)
'-------------------------------------------------------------------------
Define SRV_FETCH_NEXT                          -1
Define SRV_FETCH_PREV                          -2
Define SRV_FETCH_FIRST                         -3
Define SRV_FETCH_LAST                          -4

'-------------------------------------------------------------------------
'Oracle workspace manager
'-------------------------------------------------------------------------
Define SRV_WM_HIST_NONE                         0
Define SRV_WM_HIST_OVERWRITE                    1
Define SRV_WM_HIST_NO_OVERWRITE                 2
```

```
'============================================================================
' SessionInfo() defines
'============================================================================

Define SESSION_INFO_COORDSYS_CLAUSE             1
Define SESSION_INFO_DISTANCE_UNITS              2
Define SESSION_INFO_AREA_UNITS                  3
Define SESSION_INFO_PAPER_UNITS                 4

'============================================================================
' Set Next Document Style defines
'============================================================================
Define WIN_STYLE_STANDARD                       0
Define WIN_STYLE_CHILD                          1
Define WIN_STYLE_POPUP_FULLCAPTION              2
Define WIN_STYLE_POPUP                          3

'============================================================================
' StringCompare() defines
'============================================================================
Define STR_LT                                   -1
Define STR_GT                                   1
Define STR_EQ                                   0

'============================================================================
' StyleAttr() defines
'============================================================================
Define PEN_WIDTH                                1
Define PEN_PATTERN                              2
Define PEN_COLOR                                4
Define PEN_INDEX                                5
Define PEN_INTERLEAVED                          6
Define BRUSH_PATTERN                            1
Define BRUSH_FORECOLOR                          2
Define BRUSH_BACKCOLOR                          3
```

```
Define FONT_NAME                        1
Define FONT_STYLE                       2
Define FONT_POINTSIZE                   3
Define FONT_FORECOLOR                   4
Define FONT_BACKCOLOR                   5
Define SYMBOL_CODE                      1
Define SYMBOL_COLOR                     2
Define SYMBOL_POINTSIZE                 3
Define SYMBOL_ANGLE                     4
Define SYMBOL_FONT_NAME                 5
Define SYMBOL_FONT_STYLE                6
Define SYMBOL_KIND                      7
Define SYMBOL_CUSTOM_NAME               8
Define SYMBOL_CUSTOM_STYLE              9

'-------------------------------------------------------------------------
' Symbol kinds returned by StyleAttr() for SYMBOL_KIND
'-------------------------------------------------------------------------
Define SYMBOL_KIND_VECTOR               1
Define SYMBOL_KIND_FONT                 2
Define SYMBOL_KIND_CUSTOM               3

'=========================================================================
' SystemInfo() defines
'=========================================================================
Define SYS_INFO_PLATFORM                1
Define SYS_INFO_APPVERSION              2
Define SYS_INFO_MIVERSION               3
Define SYS_INFO_RUNTIME                 4
Define SYS_INFO_CHARSET                 5
Define SYS_INFO_COPYPROTECTED           6
Define SYS_INFO_APPLICATIONWND          7
Define SYS_INFO_DDESTATUS               8
Define SYS_INFO_MAPINFOWND              9
Define SYS_INFO_NUMBER_FORMAT           10
Define SYS_INFO_DATE_FORMAT             11
```

```
Define SYS_INFO_DIG_INSTALLED                   12
Define SYS_INFO_DIG_MODE                        13
Define SYS_INFO_MIPLATFORM                      14
Define SYS_INFO_MDICLIENTWND                    15
Define SYS_INFO_PRODUCTLEVEL                    16
Define SYS_INFO_APPIDISPATCH                    17
Define SYS_INFO_MIBUILD_NUMBER                  18

'------------------------------------------------------------------------
' Platform, returned by SystemInfo() for SYS_INFO_PLATFORM
'------------------------------------------------------------------------
Define PLATFORM_SPECIAL                         0
Define PLATFORM_WIN                             1
Define PLATFORM_MAC                             2
Define PLATFORM_MOTIF                           3
Define PLATFORM_X11                             4
Define PLATFORM_XOL                             5

'------------------------------------------------------------------------
' Version, returned by SystemInfo() for SYS_INFO_MIPLATFORM
'------------------------------------------------------------------------
Define MIPLATFORM_SPECIAL                       0
Define MIPLATFORM_WIN16                         1
Define MIPLATFORM_WIN32                         2
Define MIPLATFORM_POWERMAC                      3
Define MIPLATFORM_MAC68K                        4
Define MIPLATFORM_HP                            5
Define MIPLATFORM_SUN                           6

'========================================================================
' TableInfo() defines
'========================================================================
Define TAB_INFO_NAME                            1
Define TAB_INFO_NUM                             2
Define TAB_INFO_TYPE                            3
Define TAB_INFO_NCOLS                           4
```

```
Define TAB_INFO_MAPPABLE                        5
Define TAB_INFO_READONLY                        6
Define TAB_INFO_TEMP                            7
Define TAB_INFO_NROWS                           8
Define TAB_INFO_EDITED                          9
Define TAB_INFO_FASTEDIT                        10
Define TAB_INFO_UNDO                            11
Define TAB_INFO_MAPPABLE_TABLE                  12
Define TAB_INFO_USERMAP                         13
Define TAB_INFO_USERBROWSE                      14
Define TAB_INFO_USERCLOSE                       15
Define TAB_INFO_USEREDITABLE                    16
Define TAB_INFO_USERREMOVEMAP                   17
Define TAB_INFO_USERDISPLAYMAP                  18
Define TAB_INFO_TABFILE                         19
Define TAB_INFO_MINX                            20
Define TAB_INFO_MINY                            21
Define TAB_INFO_MAXX                            22
Define TAB_INFO_MAXY                            23
Define TAB_INFO_SEAMLESS                        24
Define TAB_INFO_COORDSYS_MINX                   25
Define TAB_INFO_COORDSYS_MINY                   26
Define TAB_INFO_COORDSYS_MAXX                   27
Define TAB_INFO_COORDSYS_MAXY                   28
Define TAB_INFO_COORDSYS_CLAUSE                 29
Define TAB_INFO_COORDSYS_NAME                   30
Define TAB_INFO_NREFS                           31
Define TAB_INFO_SUPPORT_MZ                      32
Define TAB_INFO_Z_UNIT_SET                      33
Define TAB_INFO_Z_UNIT                          34
Define TAB_INFO_BROWSER_LIST                    35
Define TAB_INFO_THEME_METADATA                  36
Define TAB_INFO_COORDSYS_CLAUSE_WITHOUT_BOUNDS  37
Define TAB_INFO_DESCRIPTION                     38
Define TAB_INFO_ID                              39
Define TAB_INFO_PARENTID                        40
```

```
Define TAB_INFO_ISMANAGED                       41
'------------------------------------------------------------------------------
' Table type defines, returned by TableInfo() for TAB_INFO_TYPE
'------------------------------------------------------------------------------
Define TAB_TYPE_BASE                            1
Define TAB_TYPE_RESULT                          2
Define TAB_TYPE_VIEW                            3
Define TAB_TYPE_IMAGE                           4
Define TAB_TYPE_LINKED                          5
Define TAB_TYPE_WMS                             6
Define TAB_TYPE_WFS                             7
Define TAB_TYPE_FME                             8
Define TAB_TYPE_TILESERVER                      9

'------------------------------------------------------------------------------
' TableListInfo() defines
'------------------------------------------------------------------------------
Define TL_INFO_SEL_COUNT                        1

'------------------------------------------------------------------------------
' Defines used in LibraryServiceInfo function for what information to return.
'------------------------------------------------------------------------------
Define LIBSRVC_INFO_LIBSRVCMODE                 1
Define LIBSRVC_INFO_LIBVERSION                  2
Define LIBSRVC_INFO_DEFURLPATH                  3
Define LIBSRVC_INFO_LISTCSWURL                  4

'------------------------------------------------------------------------------
' TableListSelectionInfo() defines
'------------------------------------------------------------------------------
Define TL_SEL_INFO_NAME                         1
Define TL_SEL_INFO_ID                           2

'------------------------------------------------------------------------------
' RasterTableInfo() defines
'------------------------------------------------------------------------------
```

```
Define RASTER_TAB_INFO_IMAGE_NAME               1
Define RASTER_TAB_INFO_WIDTH                    2
Define RASTER_TAB_INFO_HEIGHT                   3
Define RASTER_TAB_INFO_IMAGE_TYPE               4
Define RASTER_TAB_INFO_BITS_PER_PIXEL           5
Define RASTER_TAB_INFO_IMAGE_CLASS              6
Define RASTER_TAB_INFO_NUM_CONTROL_POINTS       7
Define RASTER_TAB_INFO_BRIGHTNESS               8
Define RASTER_TAB_INFO_CONTRAST                 9
Define RASTER_TAB_INFO_GREYSCALE                10
Define RASTER_TAB_INFO_DISPLAY_TRANSPARENT      11
Define RASTER_TAB_INFO_TRANSPARENT_COLOR        12
Define RASTER_TAB_INFO_ALPHA                    13

'------------------------------------------------------------------------
' Image type defines returned by RasterTableInfo() for
RASTER_TAB_INFO_IMAGE_TYPE
'------------------------------------------------------------------------
Define IMAGE_TYPE_RASTER                        0
Define IMAGE_TYPE_GRID                          1

'------------------------------------------------------------------------
' Image class defines returned by RasterTableInfo() for
RASTER_TAB_INFO_IMAGE_CLASS
'------------------------------------------------------------------------
Define IMAGE_CLASS_BILEVEL                      0
Define IMAGE_CLASS_GREYSCALE                    1
Define IMAGE_CLASS_PALETTE                      2
Define IMAGE_CLASS_RGB                          3

'------------------------------------------------------------------------
'  GridTableInfo() defines
'------------------------------------------------------------------------
Define GRID_TAB_INFO_MIN_VALUE                  1
Define GRID_TAB_INFO_MAX_VALUE                  2
Define GRID_TAB_INFO_HAS_HILLSHADE              3
```

```
'------------------------------------------------------------------------
'  ControlPointInfo() defines
'------------------------------------------------------------------------
Define RASTER_CONTROL_POINT_X                   1
Define RASTER_CONTROL_POINT_Y                   2
Define GEO_CONTROL_POINT_X                      3
Define GEO_CONTROL_POINT_Y                      4
Define TAB_GEO_CONTROL_POINT_X                  5
Define TAB_GEO_CONTROL_POINT_Y                  6

'========================================================================
' WindowInfo() defines
'========================================================================
Define WIN_INFO_NAME                            1
Define WIN_INFO_TYPE                            3
Define WIN_INFO_WIDTH                           4
Define WIN_INFO_HEIGHT                          5
Define WIN_INFO_X                               6
Define WIN_INFO_Y                               7
Define WIN_INFO_TOPMOST                         8
Define WIN_INFO_STATE                           9
Define WIN_INFO_TABLE                           10
Define WIN_INFO_LEGENDS_MAP                     10
Define WIN_INFO_ADORNEMNTS_MAP                  10
Define WIN_INFO_OPEN                            11
Define WIN_INFO_WND                             12
Define WIN_INFO_WINDOWID                        13
Define WIN_INFO_WORKSPACE                       14
Define WIN_INFO_CLONEWINDOW                     15
Define WIN_INFO_SYSMENUCLOSE                    16
Define WIN_INFO_AUTOSCROLL                      17
Define WIN_INFO_SMARTPAN                        18
Define WIN_INFO_SNAPMODE                        19
Define WIN_INFO_SNAPTHRESHOLD                   20
Define WIN_INFO_PRINTER_NAME                    21
```

```
Define WIN_INFO_PRINTER_ORIENT                  22
Define WIN_INFO_PRINTER_COPIES                  23
Define WIN_INFO_PRINTER_PAPERSIZE               24
Define WIN_INFO_PRINTER_LEFTMARGIN              25
Define WIN_INFO_PRINTER_RIGHTMARGIN             26
Define WIN_INFO_PRINTER_TOPMARGIN               27
Define WIN_INFO_PRINTER_BOTTOMMARGIN            28
Define WIN_INFO_PRINTER_BORDER                  29
Define WIN_INFO_PRINTER_TRUECOLOR               30
Define WIN_INFO_PRINTER_DITHER                  31
Define WIN_INFO_PRINTER_METHOD                  32
Define WIN_INFO_PRINTER_TRANSPRASTER            33
Define WIN_INFO_PRINTER_TRANSPVECTOR            34
Define WIN_INFO_EXPORT_BORDER                   35
Define WIN_INFO_EXPORT_TRUECOLOR                36
Define WIN_INFO_EXPORT_DITHER                   37
Define WIN_INFO_EXPORT_TRANSPRASTER             38
Define WIN_INFO_EXPORT_TRANSPVECTOR             39
Define WIN_INFO_PRINTER_SCALE_PATTERNS          40
Define WIN_INFO_EXPORT_ANTIALIASING             41
Define WIN_INFO_EXPORT_THRESHOLD                42
Define WIN_INFO_EXPORT_MASKSIZE                 43
Define WIN_INFO_EXPORT_FILTER                   44
Define WIN_INFO_ENHANCED_RENDERING              45
Define WIN_INFO_SMOOTH_TEXT                     46
Define WIN_INFO_SMOOTH_IMAGE                    47
Define WIN_INFO_SMOOTH_VECTOR                   48

'-------------------------------------------------------------------------
' Window types, returned by WindowInfo() for WIN_INFO_TYPE
'-------------------------------------------------------------------------
Define WIN_MAPPER                               1
Define WIN_BROWSER                              2
Define WIN_LAYOUT                               3
Define WIN_GRAPH                                4
Define WIN_BUTTONPAD                            19
```

```
Define WIN_TOOLBAR                          25
Define WIN_CART_LEGEND                      27
Define WIN_3DMAP                            28
Define WIN_ADORNMENT                        32
Define WIN_HELP                             1001
Define WIN_MAPBASIC                         1002
Define WIN_MESSAGE                          1003
Define WIN_RULER                            1007
Define WIN_INFO                             1008
Define WIN_LEGEND                           1009
Define WIN_STATISTICS                       1010
Define WIN_MAPINFO                          1011
'-------------------------------------------------------------------------
' Version 2 window types no longer used in version 3 or later versions
'-------------------------------------------------------------------------
Define WIN_TOOLPICKER                       1004
Define WIN_PENPICKER                        1005
Define WIN_SYMBOLPICKER                     1006

'-------------------------------------------------------------------------
' Window states, returned by WindowInfo() for WIN_INFO_STATE
'-------------------------------------------------------------------------
Define WIN_STATE_NORMAL                     0
Define WIN_STATE_MINIMIZED                  1
Define WIN_STATE_MAXIMIZED                  2

'-------------------------------------------------------------------------
' Print orientation, returned by WindowInfo() for WIN_INFO_PRINTER_ORIENT
'-------------------------------------------------------------------------
Define WIN_PRINTER_PORTRAIT                 1
Define WIN_PRINTER_LANDSCAPE                2

'-------------------------------------------------------------------------
' Antialiasing filters, returned by WindowInfo() for WIN_INFO_EXPORT_FILTER
'-------------------------------------------------------------------------
Define FILTER_VERTICALLY_AND_HORIZONTALLY   0
```

```
Define FILTER_ALL_DIRECTIONS_1                  1
Define FILTER_ALL_DIRECTIONS_2                  2
Define FILTER_DIAGONALLY                        3
Define FILTER_HORIZONTALLY                      4
Define FILTER_VERTICALLY                        5

'============================================================================
' Abbreviated list of error codes
'
' The following are error codes described in the Reference manual.  All
' other errors are listed in ERRORS.DOC.
'============================================================================
Define ERR_BAD_WINDOW                           590
Define ERR_BAD_WINDOW_NUM                       648
Define ERR_CANT_INITIATE_LINK                   698
Define ERR_CMD_NOT_SUPPORTED                    642
Define ERR_FCN_ARG_RANGE                        644
Define ERR_FCN_INVALID_FMT                      643
Define ERR_FCN_OBJ_FETCH_FAILED                 650
Define ERR_FILEMGR_NOTOPEN                      366
Define ERR_FP_MATH_LIB_DOMAIN                   911
Define ERR_FP_MATH_LIB_RANGE                    912
Define ERR_INVALID_CHANNEL                      696
Define ERR_INVALID_READ_CONTROL                 842
Define ERR_INVALID_TRIG_CONTROL                 843
Define ERR_NO_FIELD                             319
Define ERR_NO_RESPONSE_FROM_APP                 697
Define ERR_PROCESS_FAILED_IN_APP                699
Define ERR_NULL_SELECTION                       589
Define ERR_TABLE_NOT_FOUND                      405
Define ERR_WANT_MAPPER_WIN                      313
Define ERR_CANT_ACCESS_FILE                     825

'============================================================================
' Backward Compatibility defines
'
```

```
' These defines are provided so that existing MapBasic code will continue
' to compile & run correctly.  Please use the new define (on the right)
' when writing new code.
'============================================================================
Define OBJ_ARC                                  OBJ_TYPE_ARC
Define OBJ_ELLIPSE                              OBJ_TYPE_ELLIPSE
Define OBJ_LINE                                 OBJ_TYPE_LINE
Define OBJ_PLINE                                OBJ_TYPE_PLINE
Define OBJ_POINT                                OBJ_TYPE_POINT
Define OBJ_FRAME                                OBJ_TYPE_FRAME
Define OBJ_REGION                               OBJ_TYPE_REGION
Define OBJ_RECT                                 OBJ_TYPE_RECT
Define OBJ_ROUNDRECT                            OBJ_TYPE_ROUNDRECT
Define OBJ_TEXT                                 OBJ_TYPE_TEXT

'============================================================================
' Codes used to position Adornments relative to mapper
'============================================================================

Define ADORNMENT_INFO_MAP_POS_TL     0
Define ADORNMENT_INFO_MAP_POS_TC     1
Define ADORNMENT_INFO_MAP_POS_TR     2
Define ADORNMENT_INFO_MAP_POS_CL     3
Define ADORNMENT_INFO_MAP_POS_CC     4
Define ADORNMENT_INFO_MAP_POS_CR     5
Define ADORNMENT_INFO_MAP_POS_BL     6
Define ADORNMENT_INFO_MAP_POS_BC     7
Define ADORNMENT_INFO_MAP_POS_BR     8
Define SCALEBAR_INFO_BARTYPE_CHECKEDBAR    0
Define SCALEBAR_INFO_BARTYPE_SOLIDBAR      1
Define SCALEBAR_INFO_BARTYPE_LINEBAR       2
Define SCALEBAR_INFO_BARTYPE_TICKBAR       3

'============================================================================
' Coordinate system datum id's.  These match the id's from mapinfow.prj.
'============================================================================
```

```
Define DATUMID_NAD27  62
Define DATUMID_NAD83  74
Define DATUMID_WGS84 104

'=============================================================================
' end of MAPBASIC.DEF
'=============================================================================
```

Index

Symbols

! (exclamation point) in menus 187
& (ampersand)
dialog hotkeys **251**
finding street intersections **292–295**
hexadecimal numbers **770**
menu hotkeys **188**
string concatenation **845**
((open parenthesis) in menus 187
*** (asterisk)**
fixed length strings **257**
multiplication **845**
+ (plus) 845
.Net Interoperability
Declare Method statement **241**
/ (slash)
division **845**
in menus **186**, **188**
< (less than) character
in menus **186**
\ (backslash)
in menus **186**
integer division **845**
^ (caret)
exponentiation **845**
show/hide menu text **188**

Numerics

3D maps
changing window settings **677–678**
creating **183–185**
prism maps **202–204**
reading window settings **399–402**

A

Abs( ) function 39
absolute value, Abs( ) function 39
accelerator keys
in dialog boxes **251**
in menus **188**
Access databases
connection string attributes **568**
Acos( ) function 39
Add Cartographic Frame statement 40
Add Column statement 42
Add Map statement 48
adding
animation layers **50**
buttons **52–57**
columns to a table **42–48**, **76–78**
map layers **48–51**
menu items **62–67**
nodes **74**, **459**, **477**
addresses
finding **292–295**
aggregate functions 555–556
alias variables 258
all-caps text 298–300
Alter Button statement 51
Alter ButtonPad statement 52–57
Alter Cartographic Frame statement 57
Alter Control statement 58
Alter MapInfoDialog statement 60
Alter Menu Bar statement 67
Alter Menu Item statement 68
Alter Menu statement 62
Alter Object statement 70
Alter Table statement 76
animation layers
adding **50**
removing **520**
ApplicationDirectory$( ) function 78
ApplicationName$() function 79
arc objects
creating **159**
determining length of **438**
modifying **70–74**
querying the pen style **433–438**
storing in a new row **344–346**

storing in an existing row 766
area
spherical calculation 718
units of measure 608
Area( ) function 79
AreaOverlap( ) function 80
arithmetic functions. ***See*** **math functions** 38
array variables
declaring 258
determining size of array 762
resizing 507–508
Asc( ) function 81
ASCII files
exporting 279
using as tables 508–515
See also file input/output 38
Asin( ) function 82
Ask( ) function 83
assigning local storage
Server Bind Column statement 562–563
Atn( ) function 84
AutoCAD
importing files 337–342
AutoLabel statement 85
automatic type conversions 847
automation
handling button event 163
handling menu event 189
autoscroll feature
list of affected draw modes 55–56
reading current setting 779
turning on or off 700
WinChangedHandler 774
Avg( ) aggregate function 555–556

B

background colors
Brush clause 90–91
Font clause 298–300
MakeBrush( ) function 389
MakeFont( ) function 391–392
bar charts
in graph windows 330–331
in thematic maps 714–716
beep statement 86
beginning a transaction
Server Begin Transaction 561
binary file i/o
closing files 113
opening files 471
reading data 320–321
writing data 500
bitmap (*.bmp) files
creating 540–542
bold text 298–300
bounding rectangle 408
branching
Do Case...End Case statement 262
If...Then statement 336
breakpoints (debugging) 727
Browse statement 86
Browser windows
closing 115
determining the name of the table 780
modifying 609, 692–702
opening 86–87
restricting which columns appear 523
BrowserInfo function 88
Brush clause 89–91
brush styles
Brush clause defined 89–91
creating 389
modifying an object's style 71–72
querying an object's style 433–438
querying parts of 733–736
reading current style 224
setting current style 688–689
brush variables 258
BrushPicker controls 140
buffer regions
Buffer( ) function 92
CartesianBuffer( ) function 97
Create Object statement 194
Buffer( ) function 92
button controls (in dialog boxes) 132
ButtonPadInfo( ) function 93
ButtonPads
adding/removing a button 52–57, 757–759
creating a new pad 160–164
docked vs. floating 54, 162
drawing modes 55–57
enabling/disabling a button 51
querying current settings 93
resetting to defaults 164–165
responding to user action 122
selecting/deselecting a button 51
setting which button is active 534
showing/hiding a pad 52–57
byte order in file i/o 472

C

Call statement 94
Calling clause 163, 189
callout lines

map labels 652, 661, 664
text objects 221
CancelButton clause 132
capitalization
lower case 375
mixed case 499
upper case 763
CartesianArea( ) function 96
CartesianBuffer( ) function 97
CartesianDistance( ) function 99
CartesianObjectDistance( ) function 100
CartesianObjectLen( ) function 100
CartesianOffset( ) function 101
CartesianOffsetXY( ) function 102
CartesianPerimeter( ) function 103
cartographic legends
adding frames, Add Cartographic Frame statement 40–42
changing a frame 57
controlling settings 611
creating 165–169
removing a frame 519
case, converting
LCase$( ) function 375
Proper$( ) function 499
UCase$( ) function 763
Centroid( ) function 104
centroids
displaying 648, 655, 658
setting a region's 74
CentroidX( ) function 105
CentroidY( ) function 106
character codes
character sets 107–110
converting codes to strings 111
converting strings to codes 81
CharSet clause 107
checkable menu items, creating 187
CheckBox controls 132
checking
dialog box check boxes (custom) 58
dialog box check boxes (standard) 60–62
menu items 68–70
ChooseProjection$( ) function 110
Chr$( ) function 111
circle objects
creating 169–171, 173
determining area of 79
determining perimeter of 485
modifying 70–74
querying the pen or brush style 433–438
storing in a new row 344–346
storing in an existing row 766
cleaning objects 447
clicking and dragging. *See* ButtonPads 38
clipping a map 643
cloning a map 534
Close All statement 112
Close Connection statement 112
Close File statement 113
Close Table statement 114
Close Window statement 115
closing tables
Server Close statement 563
collection objects
combining 118
creating 171
resetting objects within collection 75–76
color
RGB values 524
ColumnInfo( ) function 116
columns in a table
adding 42–48, 76–78
deleting 76–78
determining column information 116–118, 426
dynamic columns 48
indexing 179
Combine( ) function 118
combining objects
Combine( ) function 118
Create Object statement 194
Objects Clean statement 447
Objects Combine statement 449
CommandInfo( ) function 119
Commit Table statement 124
comparing strings 382, 730–731
comparison operators 845–846
compiler directives
Define statement 246
Include statement 343
concatenating strings
& operator 845
+ operator 845
conditional execution
Do Case...End Case statement 262
If...Then statement 335–336
Conflict Resolution dialog box 127
Connect option
DLG=1 568
connect_string, defined 568
connecting to a data source
Server_Connect 567–574
connection number, returning 567–574
ConnectObjects( ) function 129
Continue statement 130
Control BrushPicker clause 140

Control Button clause 132
Control DocumentWindow clause 134
Control EditText clause 135
Control FontPicker clause 140
Control GroupBox clause 136
control key
detecting control-click 122
entering line feeds in EditText boxes 135
selecting multiple list items 137–139
Control ListBox clause 137
Control MultiListBox clause 137
control panels
date formatting 627–628
number formatting 627–628
Control PenPicker clause 140
Control PopupMenu clause 142
Control RadioGroup clause 143
Control StaticText clause 144
Control SymbolPicker clause 140
ControlPointInfo() function 141
controls in dialog boxes
BrushPicker 140
Button 132
CancelButton 132
EditText 135
FontPicker 140
GroupBox 136
ListBox 137–139
MultiListBox 137
OKButton 132
PenPicker 140
RadioGroup 143–144
StaticText 144
SymbolPicker 140
converting
character codes to strings 111
numbers to dates 424–425
numbers to strings 728
objects to polylines 145
objects to regions 146
strings to character codes 81
strings to dates 731–732
strings to numbers 770
text to lower case 375
text to mixed case 499
text to upper case 763
two-digit input into four-digit years 620
ConvertToPline( ) function 145
ConvertToRegion( ) function 146
convex hull
Create Object parameter 195
ConvexHull( ) function 146
coordinate systems 669–670
CoordSys clause
changing a table's CoordSys 124–129
querying a table's coordinate system 747
querying a window's coordinate system 404
setting current MapBasic coordinate system 619
specifying a coordinate system 147
CoordSysName$( ) function 151
CoordSysStringToEPSG() function 152
CoordSysStringToPRJ$() function 153
CoordSysStringToWKT$() function 153
copying
a projection
from a table 147–151
from a window 147–151
an object
offset by distance 722
offset by XY values 723
offset from source 458–459
files 538
object offset 465–466
objects
offset by specified distance 101
offset by XY values 102
tables 124–129
Cos( ) function 151, 154
cosmetic layer
accessing as a table 780
Count( ) aggregate function 555–556
Create Adornment statement 155
Create Arc statement 159
Create ButtonPad statement 160
Create ButtonPads As Default statement 164
Create Cartographic Legend statement 165
Create Collection statement 171
Create Cutter statement 172
Create Frame statement 174
Create Grid statement 176
Create Index statement 179
Create Legend statement 179
Create Line statement 181
Create Map statement 182
Create Map3D statement 183
Create Menu Bar statement 190
Create Menu statement 185
Create Multipoint statement 192
Create Object statement 194
Create Pline statement 199
Create Point statement 201
Create PrismMap statement 202
Create Ranges statement 204
Create Rect statement 207
Create Redistricter statement 208
Create Region statement 209

Create Report From Table statement 211
Create RoundRect statement 212
Create Styles statement 213
Create Table statement 214
Create Text statement 221
CreateCircle( ) function 169
CreateLine( ) function 180
CreatePoint( ) function 200
CreateText( ) function 219
Crystal Reports
creating 211
loading 472
CurDate( ) function 222
CurrDateTime function 223
CurrentBorder Pen( ) function 223
CurrentBrush( ) function 224
CurrentFont( ) function 225
CurrentLinePen( ) function 226
CurrentPen( ) function 226
CurrentSymbol( ) function 227–228
cursor coordinates, displaying 643
cursor shapes 55
cursor, position in table
end-of-table condition 271
positioning the row cursor 286–288
CurTime function 228
custom symbols
Reload Symbols statement 516
syntax 740–744
cutter objects, creating 172

D

data aggregation
combining objects 447, 449–450
filling a column with data from another table 44–48
grouping rows 555–556
data disaggregation
erasing part of an object 453–456
splitting objects 463–465
data structures 761
databases
using as tables 508–515
date
variables 258
date functions
converting numbers to dates 424
converting strings to dates 627–628, 731–732
current date 222
date window setting 229
extracting day-of-month 229
extracting day-of-week 771
extracting the month 419
extracting the year 786
formatting based on locale 627–628
DateTime feature
description 258
DateWindow( ) function 229
Day( ) function 229
DBF files, exporting 279
DDE, acting as client
closing a conversation 238–239
executing a command 230
initiating a conversation 231–234
reading data from the server 236–238
sending data to the server 234
DDE, acting as server
handling execute event 517
handling peek request 518
retrieving execute string 121
DDEExecute statement 230
DDEInitiate function 231
DDEPoke statement 234
DDERequest$( ) function 236
DDETerminate statement 238
DDETerminateAll statement 239
debugging
Continue statement 130
Stop statement 727
decimal separators 247, 306, 627–628
decision-making
Do Case...End Case statement 262
If...Then statement 335–336
Declare Function statement 239
Declare Method statement 241
Declare Sub statement 244
Define statement 246
definitions file 849
DeformatNumber$( ) function 247
delaying when user drags mouse 625
Delete statement 248
deleting
all objects from a table 266
columns from a table 76–78
files 352
nodes from an object 70
rows or objects 248
tables 267
dialog boxes, custom
accelerator keys 251
creating 249–255
determining ID of a control 759
determining if user clicked OK 120
determining if user double-clicked 120
modal vs. modeless 250

modifying 58
preserving after user clicks OK 255
reading user's input 251, 504–507
sizes of dialog boxes and controls 250
tab order 252
terminating 251, 256
dialog boxes, standard
altering MapInfo dialog boxes 60–62
asking OK/Cancel question 83
opening a file 289–291
percent complete 496–498
saving a file 291
simple messages 423
suppressing progress bars 684
Dialog Preserve statement 255
Dialog Remove statement 256
Dialog statement 249
digitizer
setup 621–623
status 745
digitizer setup 622
Dim statement 257
directory names
extracting from a file name 479
user's home directory 334
user's windows directory 334
where application is installed 78
where MapInfo is installed 496
disabling
ButtonPad buttons 51
dialog box controls (custom) 58
dialog box controls (standard) 60–62
handler procedures 633
menu items 68–70
progress bar dialog boxes 684
shortcut menus 190
system menu's Close command 699
discarding changes
to a local table 527
to a remote server 595
distance
spherical calculation 719
units of measure 624
Distance( ) function 261
DLG=1 connect option 568
DLLs
declaring as functions 240–241
declaring as procedures 245
Do Case...End Case statement 262
Do...Loop statement 264
dockable
ButtonPads, querying current status 93
document conventions 20, 38
DOS commands, executing 537
dot density maps
thematic maps 710–711
double byte character sets (DBCS)
extracting part of a DBCS string 417
double-clicking in dialog boxes 120, 138
dragging with the mouse
time threshold 625
turning off autoscroll 700
drawing
modes 55–56
drawing objects on a map
See objects, creating 38
drawing tools, custom 52–57
Drop Index statement 266
Drop Map statement 266
Drop Table statement 267
duplicating a map 534
DXF files
exporting 279
importing 337–342
dynamic columns 48
dynamic link libraries. *See* DLLs

E

editable map layers 403, 648
editing objects
See specific object type
arc, ellipse, frame, line, point, polyline, rectangle, region, rounded rectangle, text 38
edits
determining if there are unsaved edits 746–751
discarding 527
saving 124–129
EditText controls 135
elapsed time 757
ellipse objects
Cartesian area of 96
Cartesian perimeter of 103
creating 169–171, 173
determining area of 79
determining perimeter of 485
modifying 70–74
querying pen or brush style 433–438
storing in a new row 344–346
storing in an existing row 766
enabling
ButtonPad buttons 51
dialog box controls 58
menu items 68–70
End MapInfo statement 268

End Program statement 269
EndHandler procedure 270
enlarging arrays 507–508
EOF( ) function 270
EOT( ) function 271
EPSGToCoordSysString$( ) function 272
Erase( ) function 273
erasing
entire objects 248
files 352
part of an object 273, 453–456
tables 267
Err( ) function 274
error handling
determining error code 274
determining error message 275
enabling an error handler 467–468
generating an error 275
returning from an error handler 523
Error statement 275
Error$( ) function 275
Escape key
cancelling draw operations 54–57
dismissing a dialog box 120
interrupting selection 550
events, handling
application terminated 270
automation method used 516
execute string received 121, 517
map window changed 121, 774
MapInfo got or lost focus 122, 302
peek request received 518
selection changed 121
user clicked with custom tool 122, 757–759
user double-clicked in a dialog box 120
window closed 121, 775
window focus changed 784
See also error handling 38
Excel files
opening 508–515
executing
interpreted strings 532–534
menu commands 534
Run Application statement 531
Run Program statement 536
executing an SQL string
Server_Execute( ) 585
execution speed
animation layers 50
screen updates 625
table editing 689–691
Exit Do statement 276
Exit For statement 277
Exit Function statement 277
Exit Sub statement 278
exiting MapInfo Professional
End MapInfo statement 268
Exp( ) function 279
expanded text 298–300
exponentiation 279
Export statement 279
extents of entire table 748
external functions 240–241
extracting part of a string
Left$( ) function 376
Mid$( ) function 416
MidByte$( ) function 417
Right$( ) function 525
ExtractNodes( ) function 282

F

Farthest statement 283
Fetch statement 286
file input/output
closing a file 113
determining if file exists 289
end-of-file condition 270
file attributes, reading 288
length of file 386
opening a file 470–472
reading current position 548
reading data in binary mode 320–321
reading data in random mode 320–321
reading data in sequential mode 343, 383
setting current position 549
writing data in binary mode 500
writing data in random mode 500
writing data in sequential mode 491, 785
file names
determining full file spec 760
determining temporary name 754
extracting directory from 479
extracting from full file spec 480
file sharing conflicts 626
FileAttr( ) function 288
FileExists( ) function 289
FileOpenDlg( ) function 289
files
copying 538
deleting 352
determining if file exists 289
importing 337–342
length 386
locating 384
renaming 521

FileSaveAsDlg( ) function 291
fill styles. *See* **brush styles**
filtering data 553–555
Find statement 292
Find Using statement 296
finding
a substring within a string 346
an address in a map 292–295
an intersection of two streets 292–295
objects from map coordinates 543–548
Fix( ) function 297
fixed length strings 257
floating point variables 257
flow control
exiting a Do loop 276
exiting a For loop 277
exiting a function 277
exiting a procedure 278
exiting an application 269
exiting MapInfo 268
halting another application 755
unconditional jump 329
focus
active window changes 784
getting or losing 122, 302
within a dialog box 59
folder names. *See* **directory names**
Font clause 298–300
font styles
creating 392
Font clause defined 298–300
modifying an object's style 71–72
querying an object's style 433–438
querying parts of 733–736
reading current style 225
setting current style 688–689
FontPicker controls 140
fonts
variables 258
For...Next statement 300
ForegroundTaskSwitchHandler procedure 302
foreign character sets 107–110
Format$( ) function 303
FormatDate$( ) function 305
FormatNumber$( ) function 306
FormatTime$ function 307
frame objects
creating 174–176
inserting into a layout 344–346
modifying 70–74, 766
querying the pen or brush style 433–438
frames, cartographic legend
adding a frame 40–42
controlling settings 611
creating 165–169
modifying 57
removing 519
FrontWindow( ) function 307, 309
FTP Library 793
Function...End Function statement 309
functions, creating
Declare Function statement 239
Exit Function statement 277
Function....End Function statement 309

G

gaps
checking in regions 446–447
cleaning 447
snapping nodes 461–463
Geocode statement 312
GeocodeInfo() function 317
geographic
operators 558
geographic calculations
area of object 79
area of overlap 80
distance 261
length of object 438
perimeter of object 485
See objects, querying 38
geographic operators 846
Get statement 320
GetCurrentPath() function 322
GetDate function 323
GetFolderPath$() function 323
GetGridCellValue() function 324
GetMetadata$( ) function 325
GetPreferencePath$() function 326
GetSeamlessSheet( ) function 327
GetTime function 328
Global statement 328
GML files, importing 337–342
Goto statement 329
GPS applications 50–51
Graph statement 330
Graph windows
closing 115
determining the name of the table 780
modifying 628–633, 692–702
opening 323, 330–331
great circle distance 261
grid
tables, adding relief shade information 515
grid surface maps

in thematic maps 176–179
modifying 641–671
GridTableInfo( ) function 331
Group By clause 555–557
group layers 666–667
GroupBox controls 136
GroupLayerInfo function 332

H

halo text 298–300
halting another application 755
handlers, assigning to menu items 186
hardware platform, determining 744–746
help messages
button tooltips 54, 161
status bar messages 54, 161
Help window
closing 115
modifying 692–702
opening 475, 692–702
hexadecimal numbers 770
hiding
ButtonPads 52–57
dialog box controls (custom) 58
dialog box controls (standard) 60–62
menu bar 409
progress bar dialog boxes 684
screen activity 625
hierarchical menus
Alter Menu statement 66
Create Menu statement 187
HomeDirectory$( ) function 334
HotLink tool
querying object attributes 123
HotLinkInfo function 334
HotLinks 671–674
adding 674–675
modifying 675–676
querying 364
removing 676
reordering 676
Hour function 335
HTTP and FTP Library
MICloseContent() procedure 794
MICloseFtpConnection() procedure 794
MICloseFtpFileFind() procedure 794
MICloseHttpConnection() procedure 795
MICloseHttpFile() procedure 795
MICloseSession() procedure 796
MICreateSession() function 796
MICreateSessionFull() function 797
MIErrorDlg() function 798
MIFindFtpFile() function 800
MIFindNextFtpFile() function 801
MIGetContent() function 801
MIGetContentBuffer() function 803
MIGetContentLen() function 803
MIGetContentString() function 804
MIGetContentToFile() function 804
MIGetContentType() function 805
MIGetCurrent FtpDirectory() function 806
MIGetErrorCode() function 806
MIGetErrorMessage() function 807
MIGetFileURL() function 807
MIGetFtpConnection() function 808
MIGetFtpFile() function 809
MIGetFtpFileFind() function 810
MIGetFtpFileName() procedure 811
MIGetHttpConnection() function 811
MIIsFtpDirectory() function 812
MIIsFtpDots() function 813
MIOpenRequest() function 813
MIOpenRequestFull() function 814
MIParseURL() function 816
MIPutFtpFile() function 817
MIQueryInfo() function 818
MIQueryInfoStatusCode() function 819
MISaveContent() function 820
MISendRequest() function 820
MISendSimpleRequest() function 821
MISetCurrentFtpDirectory() function 822
MISetSessionTimeout() function 822
HWND values, querying
SystemInfo( ) function 744
WindowInfo( ) function 780

I

iconizing MapInfo
Set Window statement 692
suppressing progress bars 684
icons for ButtonPads 55
identifiers, defining 246
If...Then statement 335–336
Import statement 337
Include statement 343
indexed columns
creating an index 179
deleting an index 266
infinite loops, avoiding 633
Info tool
closing Info window 115
modifying Info window 692–702
opening Info window 475
setting to read-only 701

setting which data displays 701
initializing variables 261
Input # statement 343
input/output. ***See*** **file input/output** 38
Insert statement 344
inserting
columns in a table 42–48, 76–78
nodes in an object 74
rows in a table 344–346
InStr( ) function 346
Int( ) function 347
integer division 845
integer variables 257
integrated mapping
managing legends 179
reparenting dialog boxes 607
reparenting document windows 679–680
international
character sets 107–110
formatting 627–628
interpreting strings as commands 532–534
interrupting the selection 550
intersection of objects
Create Object statement 194
Intersects operator 558
Objects Intersect statement 455
Overlap( ) function 476
intersection of two streets, finding 292–295
IntersectNodes( ) function 348
IsGridCellNull( ) function 348
IsogramInfo( ) function 349
IsPenWidthPixels( ) function 352
italic text 298–300

J

joining tables 553–555
JPEG file interchange format (*.jpg)
creating 540–542

K

keys, metadata 413–414
keywords 259, 555
Kill statement 352

L

LabelFindByID( ) function 353
LabelFindFirst( ) function 354
LabelFindNext( ) function 355
LabelInfo( ) function 356
LabelOverrideInfo( ) funtion 359
labels
in dialog boxes 144
in programs 329
on maps 85, 353–359, 650–653
reading label expressions 366
launching other applications
Run Application statement 531
Run Program statement 536
LayerControlSelectionInfo( ) function 363
LayerInfo( ) function 363–364
LayerListInfo function 372
layers
adding 48–51
cosmetic 780
group 666–667
groups 667–668
modifying settings 641–671
reading settings 364–371
removing 520
thematic maps 687, 704–716
LayerStyleInfo( ) funtion 373
Layout statement 374
Layout windows
accessing as tables 780
closing 115
creating frames 174–176
modifying 634–636, 692–702
opening 374
specifying layout coordinates 147–151, 619
LCase$( ) function 375
Left$( ) function 376
legend frames
querying attributes 377–378
querying styles 379
Legend windows
closing 115
modifying 636–639, 692–702
opening 179, 475
querying 378
legend, cartographic
adding a frame 40–42
controlling settings 611
creating 165–169
modifying a frame 57
removing a frame 519
LegendFrameInfo( ) function 377–378
LegendInfo( ) function 378
LegendStyleInfo( ) function 379
Len( ) function 380
length
of a file 386
of an object 438
spherical calculation of 721
LibraryServiceInfo( ) function 381
Like( ) function 382

line feed character
Chr$(10) function 111
used in EditText controls 135
used in text objects 221
Line Input statement 383
line objects
Cartesian length of 100
creating 180–181
determining length of 438
modifying 70–74
querying the pen style 433–438
storing in a new row 344–346
storing in an existing row 766
line styles
See pen styles
linked tables
creating 590–592
determining if table is linked 750
refreshing 594
saving 127
unlinking 766
ListBox controls 137–139
locale settings 247, 306, 627–628
LocateFile$( ) function 384
LOF( ) function 386
Log( ) function 386
logical
operators 846
variables 258
looping
Do...Loop statement 264
For...Next statement 300
While...Wend statement 772
Lotus 1-2-3 tables
opening files 508–515
lower case, converting to 375
LTrim$( ) function 387

M

Main procedure 388–389
MakeBrush( ) function 389
MakeCustomSymbol( ) function 390
MakeDateTime function 391
MakeFont( ) function 391–392
MakeFontSymbol( ) function 393
MakePen( ) function 394
MakeSymbol( ) function 395
map projections
changing a table's projection 124–129
copying from a table or window 147–151
querying a table's CoordSys 747
querying a window's CoordSys 404
setting 669–670
setting the current MapBasic CoordSys 619
map scale
determining in map windows 403
displaying 645
Map statement 396
Map windows
adding map layers 48–51
clipping 643
closing 115
controlling redrawing 48, 625, 643
creating thematic layers 704–716
duplicating 534
handling window-changed event 121, 774
labeling 650
modifying 641–671, 692–702
modifying thematic layers 687
opening 396
prism 202–204
reading layer settings 364–371
reading window settings 402–407
removing map layers 520
Map3DInfo( ) function 399
MapBasic
definitions file 849
language overview 20–38
MapBasic window
Abs function 39
Acos() function 39
Add Cartographic Frame statement 40
Add Column statement 42
Add Map statement 48
Alter Cartographic Frame statement 57
Alter Object statement 70
Alter Table statement 76
ApplicationDirectory$( ) function 78
ApplicationName$( ) function 79
Area() function 79
AreaOverlap() function 80
Asc() function 81
Asin() function 82
Ask() function 83
Atn() function 84
AutoLabel statement 85
Beep statement 86
Browse statement 86
BrowserInfo function 88
Brush clause 89
Buffer() function 92
ButtonPadInfo() function 93
CartesianArea() function 96
CartesianBuffer() function 97
CartesianConnectObjects() function 98

CartesianDistance() function 99
CartesianObjectDistance() function 100
CartesianObjectLen() function 100
CartesianOffset() function 101
CartesianOffsetXY() function 102
CartesianPerimeter() function 103
Centroid() function 104
CentroidX() function 105
CentroidY() function 106
CharSet clause 107
ChooseProjection$() function 110
Chr$() function 111
Close All statement 112
Close Connection statement 112
Close Table statement 114
Close Window statement 115
ColumnInfo() function 116
Combine() function 118
CommandInfo() function 119
Commit Table statement 124
ConnectObjects() function 129
Continue statement 130
ControlPointInfo() function 141
ConvertToPline() function 145
ConvertToRegion() function 146
ConvexHull() function 146
CoordSys clause 147
CoordSysName$( ) function 151
CoordSysStringToEPSG() function 152
CoordSysStringToPRJ$() function 153
CoordSysStringToWKT() function 153
Cos() function 154
Create Adornment statement 155
Create Arc statement 159
Create ButtonPad statement 160
Create ButtonPads As Default statement 164–165
Create Cartographic Legend statement 165
Create Collection statement 171
Create Cutter statement 172
Create Ellipse statement 173
Create Frame statement 174
Create Grid statement 176
Create Index statement 179
Create Legend statement 179
Create Line statement 181
Create Map statement 182
Create Map3D statement 183
Create Menu Bar statement 190
Create Menu statement 185
Create MultiPoint statement 192
Create Object statement 194
Create Pline statement 199
Create Point statement 201
Create PrismMap statement 202
Create Ranges statement 204
Create Rect statement 207
Create Redistricter statement 208
Create Region statement 209
Create Report From Table statement 211
Create RoundRect statement 212
Create Styles statement 213
Create Table statement 214
Create Text statement 221
CreateCircle() function 169
CreateLine() function 180
CreatePoint() function 200
CreateText() function 219
CurDate() function 222
CurDateTime function 223
CurrentBorderPen() function 223
CurrentBrush() function 224
CurrentFont() function 225
CurrentLinePen() function 226, 229
CurrentPen() function 226
CurrentSymbol() function 227
CurrTime function 228
Day() function 229
DeformatNumber$() function 247
Delete statement 248
Dim statement 257
Distance() function 261
Drop Index statement 266
Drop Map statement 266
Drop Table statement 267
EOT() function 271
EPSGToCoordSysString$() function 272
Erase() function 273
Err() function 274
Error statement 275
Error$() function 275
Exp() function 279
Export statement 279
ExtractNodes() function 282
Farthest statement 283
Fetch statement 286
File Using statement 296
FileAttr() function 288
FileExists() function 289
FileOpenDlg() function 289
FileSaveAsDlg() function 291
Find statement 292
Fix() function 297
Font clause 298
Format$() function 303
FormatDate$() function 305

FormatNumber$() function 306
FormatTime$ function 307
FrontWindow() function 309
Geocode statement 312
GeocodeInfo() function 317
GetCurrentPath$() function 322
GetDate function 323
GetFolderPath$() function 323
GetGridCellValue() function 324
GetMetadata$() function 325
GetPreferencePath$() function 326
GetSeamlessSheet() function 327
GetTime function 328
Graph statement 330
GridTableInfo( ) function 331
GroupLayerInfo function 332
HomeDirectory$() function 334
HotLinkInfo function 334
Hour function 335
Import statement 337
Include statement 343
Insert statement 344
InStr() function 346
Int() function 347
IntersectNodes() function 348
IsGridCellNull( ) function 348
IsogramInfo() function 349
IsPenWidthPixels() function 352
Kill statement 352
LabelFindByID() function 353
LabelFindFirst() function 354
LabelFindNext() function 355
Labelinfo() function 356
LabelOverrideInfo( ) funtion 359
LayerControlInfo() function 363
LayerControlSelectionInfo() function 363
LayerInfo() function 364
LayerListInfo function 372
LayerStyleInfo( ) funtion 373
Layout statement 374
LCase$() function 375
Left$() function 376
LegendFrameInfo() function 377
LegendInfo() function 378
LegendStyleInfo() function 379
Len() function 380
LibraryServiceInfo() function 381
Like() function 382
LocateFile$() function 384
Log() function 386
LTrim$() function 387
MakeBrush() function 389
MakeCustomSymbol() function 390
MakeDateTime function 391
MakeFont() function 392
MakeFontSymbol() function 393
MakePen() function 394
MakeSymbol() function 395
Map statement 396
Map3DInfo() function 399
MapperInfo() function 402
Maximum() function 407
MBR() function 408
Menu Bar statement 409
Metadata statement 412
MGRSToPoint() function 415
Mid$() function 416
MidByte$() function 417
Minimum() function 417
Minute function 418
Month() function 419
Nearest statement 420
Note statement 423
NumAllWindows() function 423
NumberToDate() function 424
NumberToDateTime() function 425
NumCols() function 426
NumTables() function 427
NumWindows() function 427
ObjectDistance() function 431
ObjectGeography() function 431
ObjectInfo() function 433
ObjectLen() function 438
ObjectNodeHasM() function 439
ObjectNodeHasZ() function 440
ObjectNodeM() function 441
ObjectNodeX() function 443
ObjectNodeY() function 444
ObjectNodeZ() function 445
Objects Check statement 446
Objects Clean statement 447
Objects Combine statement 449
Objects Disaggregate statement 450
Objects Enclose statement 452
Objects Erase statement 453
Objects Intersect statement 455
Objects Move statement 457
Objects Offset statement 458
Objects Overlay statement 459
Objects Pline statement 460
Objects Snap statement 461
Objects Split statement 463
Offset() function 465
OffsetXY() function 466
Open Connection statement 469
Open Report statement 472

Open Table statement 473
Open Window statement 475
Overlap() function 476
OverlayNodes() function 477
Pack Table statement 478
PathToDirectory$() function 479
PathToFileName$() function 480
PathToTableName$() function 480
Pen clause 481
PenWidthToPoints() function 484
Perimeter() function 485
PointsToPenWidth() function 486
PointToMGRS$( ) function 487
PointToUSNG$( ) function 488
Print statement 490
PrintWin statement 492
PrismMapInfo() function 493
ProgramDirectory$() function 496
ProgressBar statement 496
Proper$() function 499
ProportionOverlap() function 499
Randomize statement 501
RasterTableInfo() 502
reference description 38
Register Table statement 508
Relief Shade statement 515
Reload Symbols statement 516
RemoteQueryHandler() function 518
Remove Cartographic Frame statement 519
Remove Map statement 520
Rename File statement 521
Rename Table statement 522
RGB() function 524
Right$() function 525
Rnd() function 526
Rollback statement 527
Rotate() function 528
RotateAtPoint() function 529
Round() function 530
RTrim$() function 531
Run Application statement 531
Run Command statement 532
Run Menu Command statement 534
Run Program statement 536
Save File statement 538
Save MWS statement 538
Save Window statement 540
Save Workspace statement 542
SearchInfo() function 543
SearchPoint() function 546
SearchRect() function 547
Second function 548
Select statement 551
SelectionInfo() function 560
SelIver Begin Transaction statement 561
Serve Create Map statement 576
Server Bind Column statement 562
Server Close statement 563
Server Commit statement 566
Server Create Style statement 578
Server Create Table statement 579
Server Create Workspace statement 581
Server Disconnect statement 582
Server Fetch statement 586
Server Link Table statement 590
Server Refresh statement 594
Server Remove Workspace statement 595
Server Rollback statement 595
Server Set Map statement 596
Server Versioning statement 597
Server Workspace Merge statement 599
Server Workspace Refresh statement 601
Server_ColumnInfo() function 564
Server_Connect() function 567
Server_ConnectInfo() function 575
Server_DriverInfo() function 583
Server_EOT() function 584
Server_Execute() function 585
Server_GetODBCHConn() function 588
Server_GetODBCHStmt() function 589
Server_NumCols() function 592
Server_NumDrivers() function 593
SessionInfo() function 603
Set Adornment statement 604
Set Application Window statement 607
Set Area Units statement 608
Set Browse statement 609
Set Buffer Version statement 610
Set Cartographic Legend statement 611
Set Combine Version statement 612
Set Command Info statement 613
Set Connection Geocode statement 614
Set Connection Isogram statement 617
Set CoordSys statement 619
Set Date Window statement 620
Set Datum Version statement 621
Set Digitizer statement 622
Set Distance Units statement 624
Set Drag Threshold statement 625
Set Event Processing statement 625
Set File Timeout statement 626
Set Format statement 627
Set Graph statement 628
Set Handler statement 633
Set Layout statement 634
Set Legend statement 636

Set LibraryServiceInfo statement 639
Set Map statement 641
Set Map3D statement 677
Set Next Document statement 679
Set Paper Units statement 680
Set Path statement 681
Set PrismMap statement 682
Set ProgressBars statement 684
Set Redistricter statement 684
Set Resolution statement 686
Set Shade statement 687
Set Style statement 688
Set Table Datum statement 689
Set Table statement 689
Set Target statement 692
Set Window statement 692
Sgn() function 703
Shade statement 704
Sin() function 716
Space() function 717
SphericalArea() function 718
SphericalConnectObjects() function 719
SphericalDistance() function 719
SphericalObjectDistance() function 720
SphericalObjectLen() function 721
SphericalOffset() function 722
SphericalOffsetXY() function 723
SphericalPerimeter() function 724
Sqr() function 725
StatusBar statement 726
Stop statement 727
Str$() function 728
String$() function 729
StringCompare() function 730
StringCompareIntl() function 731
StringToDate() function 731
StringToDateTime function 733
StringToTime function 733
StyleAttr() function 734
StyleOverrideInfo( ) function 736
Symbol clause 740
SystemInfo() function 744
TableInfo() function 746
TableListInfo() function 751
TableListSelectionInfo() function 753
Tan() function 753
TempFileName$() function 754
Terminate Application statement 755
TextSize() function 756
Time() function 756
Timer() function 757
TriggerControl() function 759
TrueFileName$() function 760
UBound() function 762
UCase$() function 763
UnDim statement 763
UnitAbbr$() function 764
UnitName$() function 765
Unlink statement 766
Update statement 766
Update Window statement 767
URL clause 768
USNGToPoint(string) 769
Val() function 770
Weekday() function 771
WFS Refresh Table statement 772
WindowID() function 776
WindowInfo() function 777
Year() function 786

MapInfo 3.0 symbols 740–744
MAPINFOW.ABB file 295
MapperInfo( ) function 402
math functions
absolute value 39
arc-cosine 39
arc-sine 82
arc-tangent 84
area of object 79
area of overlap 80
converting strings to numbers 770
cosine 154
distance 261
exponentiation 279
logarithms 386
maximum value 407
minimum value 417
rounding off a number 297, 347, 530
sign 703
sine 716
square root 725
tangent 753

Max( ) aggregate function 555–556
Maximum( ) function 407
MBR( ) function 408
memo fields 76, 124, 127
menu
commands, executing 534

Menu Bar statement 409
MenuitemInfoByHandler( ) function 409
MenuitemInfoByID( ) function 411
menus, customizing
adding hierarchical menus 66
adding menu items 62–67
altering menu items 68–70
creating checkable menu items 187
creating new menus 185–190

disabling shortcut menus 190
querying menu item status 409–411
redefining the menu bar 67–68, 190–192
removing menu items 62–67
showing/hiding the menu bar 409
merging objects. ***See*** **combining objects**
messages
displaying in a Note dialog box 423
displaying on the status bar 726
opening the Message window 475
printing to the Message window 490
metadata
code example 414
keys 413–414
managing in tables 412–414
reading keys 325
statement 412
metric units
area 608
distance 624
MGRSToPoint( ) function 415
MICloseContent( ) procedure 794
MICloseFtpConnection( ) procedure 794
MICloseFtpFileFind( ) procedure 794
MICloseHttpConnection( ) procedure 795
MICloseHttpFile( ) procedure 795
MICloseSession( ) procedure 796
MICreateSession( ) function 796
MICreateSessionFull( ) function 797
Microsoft Access tables
connection string attributes 568
Mid$( ) function 416
MidByte$( ) function 417
MIErrorDlg( ) function 798
MIF files
exporting 279
importing 337–342
MIFindFtpFile( ) function 800
MIFindNextFtpFile( ) function 801
MIGetContent( ) function 801
MIGetContentBuffer( ) function 803
MIGetContentLen( ) function 803
MIGetContentString( ) function 804
MIGetContentToFile( ) function 804
MIGetContentType( ) function 805
MIGetCurrentFtpDirectory( ) function 806
MIGetErrorCode( ) function 806
MIGetErrorMessage( ) function 807
MIGetFileURL( ) function 807
MIGetFtpConnection( ) function 808
MIGetFtpFile( ) function 809
MIGetFtpFileFind( ) function 810
MIGetFtpFileName( ) procedure 811
MIGetHttpConnection( ) function 811
MIIsFtpDirectory( ) function 812
MIIsFtpDots( ) function 813
military grid reference format 403, 670
Min( ) aggregate function 555–556
minimizing MapInfo
Set Window statement 692
suppressing progress bars 684
minimum bounding rectangle
of an object 408
of entire table 748
Minimum( ) function 417
Minute function 418
MIOpenRequest( ) function 813
MIOpenRequestFull( ) function 814
MIParseURL( ) function 816
MIPutFtpFile( ) function 817
MIQueryInfo( ) function 818
MIQueryInfoStatusCode( ) function 819
MISaveContent( ) function 820
MISendRequest( ) function 820
MISendSimpleRequest( ) function 821
MISetCurrentFtpDirectory( ) function 822
MISetSessionTimeout( ) function 822
mixed case, converting to 499
MIXmlAttributeListDestroy( ) procedure 825
MIXmlDocumentCreate( ) function 825
MIXmlDocumentDestroy( ) procedure 826
MIXmlDocumentGetNamespaces( ) function 826
MIXmlDocumentGetRootNode( ) function 827
MIXmlDocumentLoad( ) function 827
MIXmlDocumentLoadXML( ) function 828
MIXmlDocumentLoadXMLString( ) function 829
MIXmlDocumentSetProperty( ) function 830
MIXmlGetAttributeList( ) function 831
MIXmlGetChildList( ) function 831
MIXmlGetNextAttribute( ) function 832
MIXmlGetNextNode( ) function 833
MIXmlNodeDestroy( ) procedure 833
MIXmlNodeGetAttributeValue( ) function 834
MIXmlNodeGetFirstChild( ) function 834
MIXmlNodeGetName( ) function 835
MIXmlNodeGetParent( ) function 836
MIXmlNodeGetText( ) function 836
MIXmlNodeGetValue( ) function 837
MIXmlNodeListDestroy( ) procedure 837
MIXmlSCDestroy( ) procedure 838
MIXmlSCGetLength( ) function 838
MIXmlSCGetNamespace( ) function 839
MIXmlSelectNodes( ) function 839
MIXmlSelectSingleNode( ) function 840
Mod operator 845
modal dialog boxes 250

modifying an object. *See* specific object type
arc, ellipse, frame, line, point, polyline, rectangle, region, rounded rectangle, text 38
Month( ) function 418–419
most-recently-used list (File menu) 187
mouse actions 625
mouse cursor
customizing shape of 55
displaying coordinates of 643
moving
an object 457–458
MRU list (File menu) 187
MultiListBox controls 137
multipoint objects
combining 118
creating 192–194
inserting nodes 75–76

N

natural break
thematic ranges 205
Nearest statement 420
network file sharing 626
nodes
adding 74, 459, 477
displaying 648, 655, 658
extracting a range of nodes from an object 282
maximum number per object 200, 210
querying number of nodes 435
querying x/y coordinates 443–444
removing 74
Noselect keyword 555
note statement 423
null handling 587
NumAllWindows( ) function 423
number of characters in a string 380
NumberToDate( ) function 424–425
NumberToDateTime function 425
NumCols( ) function 426
numeric operators 845
NumTables( ) function 427
NumWindows( ) function 427

O

object variables 258
ObjectDistance( ) function 431
ObjectGeography( ) function 431
ObjectInfo( ) function 433
ObjectLen( ) function 438
ObjectNodeHasM( ) function 439
ObjectNodeHasZ( ) function 440
ObjectNodeM( ) function 441
ObjectNodeX( ) function 443
ObjectNodeY( ) function 444
ObjectNodeZ( ) function 445
objects
moving within input table 457–458
Objects Check statement 446
Objects Clean statement 447
Objects Combine statement 449
Objects Disaggregate statement 450
Objects Enclose statement 452
Objects Erase statement 453
Objects Intersect statement 455
Objects Move statement 457
Objects Offset statement 458
Objects Overlay statement 459
Objects Pline statement 460
Objects Snap statement 461
Objects Split statement 463
objects, copying
offset by distance 465–466
to offset location 458–459
objects, creating
arcs 159
by buffering 92, 194–199
by combining objects 118
by intersecting objects 476
circles 169–171, 173
convex hull 195
ellipses 169–171, 173
frames 174–176
lines 180–181
map labels 85
multipoint 192–194
points 200–201
polylines 199
rectangles 207
regions 209–211
rounded rectangles 212
text 219–221
voronoi polygons 196
objects, modifying
adding nodes 74, 459
combining 118, 449–450
converting to polylines 145
converting to regions 146
erasing entire object 248
erasing part of an object 273, 453–456
moving nodes 70–76
removing nodes 74
resolution of converted objects 686
rotating 528
rotating around specified point 529

setting the target object 692
snap setting 461–463
splitting 463–465
objects, querying
area 79
boundary gaps 446
boundary overlap 446
centroid 104–106
content of a text object 435
coordinates 431, 443–444
HotLink support 123
length 438
minimum bounding rectangle 408
number of nodes 435
number of polygons in a region 435
number of sections in a polyline 435
overlap, area of 80
overlap, proportion of 499
perimeter 485
points of intersection 348
styles 433–438
type of object 434–438
ODBC connection 129
ODBC tables
changing object styles in mappable tables 596
Offset( ) function 465–466
OffsetXY( ) function 466
OKButton clause 132
OLE Automation
handling button event 163
handling menu event 189
OnError statement 467
Open Connection statement 469
Open File statement 470
Open Report statement 472
Open Table statement 473
Open Window statement 475
opening windows
Browse statement 86–87
Create Redistricter statement 208
Graph statement 323, 330
Layout statement 374
Map statement 396
Open Window statement 475
OpenStreetMap tile server 219
operating environment, determining 744–746
operators
automatic type conversions 847
summary of 844
optimizing performance
animation layers 50
screen updates 625
table editing 689–691
Oracle
databases, connection string attributes 568–570
Oracle8i databases
connection string attributes 570
Order By clause
sorting rows 558
ordering layers 667–668
Overlap( ) function 476
overlaps
checking in regions 446–447
cleaning 447
snapping nodes 461–463
OverlayNodes( ) function 477

P

Pack Table statement 478
page layout, opening 374
paper units of measure 680
papersize attribute 781
parallel labels 652, 662, 664
parent windows
reparenting dialog boxes 607
reparenting document windows 679–680
partialsegments option 653, 662, 664
PathToDirectory$( ) function 479
PathToFileName$( ) function 480
PathToTableName$( ) function 480
pattern matching 382
peek requests 518
Pen clause 481
pen styles
creating 394
modifying an object's style 71–72
Pen clause defined 481
querying an object's style 433–438
querying parts of 733–736
reading current border style 223
reading current line style 226
reading current style 226
setting current style 688–689
pen variables 258
PenPicker controls 140
PenWidthToPoints( ) function 484
percent complete dialog box 496–498
performance, improving
animation layers 50
screen updates 625
table editing 689–691
Perimeter( ) function 485
per-object styles 578
PICT files
creating 540–542

importing 337–342
pie charts
in graph windows 323, 330–331
in thematic maps 712–714
platform, determining 744–746
PNG files
creating 540–542
point objects
creating 200–201
modifying 70–74
querying the symbol style 433–438
storing in a new row 344–346
storing in an existing row 766
point styles. ***See*** **symbol styles**
PointsToPenWidth( ) function 486
PointToMGRS$( ) function 487
PointToUSNG$( ) function 488
polygon draw mode 56
polyline objects
adding/removing nodes 74
Cartesian length of 100
converting objects to polylines 145
creating 199
creating cutter objects 172
determining length of 438
extracting a range of nodes from 282
modifying the pen style 71–72
querying the pen style 433–438
storing in a new row 344–346
storing in an existing row 766
PopupMenu controls 142
positioning the row cursor 286–288
precedence of operators 847
Preferences dialog box(es) 535
preventing user from closing windows 699
Print # statement 491
Print statement 490
printer settings 692–702
overriding default printer 701
printing
attributes 781
PrintWin statement 492
prism maps
creating 202–204
properties 493–495
setting 682–683
PrismMapInfo( ) function 493
procedures, creating
Call statement 94–96
Declare Sub statement 244
Exit Sub statement 278
Sub...End Sub statement 739
procedures, special
EndHandler 270
ForegroundTaskSwitchHandler 302
Main 388–389
RemoteMapGenHandler 516
RemoteMsgHandler 517
SelChangedHandler 549
ToolHandler 757–759
WinChangedHandler 774
WinClosedHandler 775
WinFocusChangedHandler 784
ProgramDirectory$( ) function 496
progress bars, hiding 684
ProgressBar statement 496
projections
changing a table's projection 124–129
copying from a table or window 147–151
querying a table's CoordSys 747
querying a window's CoordSys 404
setting the current MapBasic CoordSys 619
setting within an application 110
Proper$( ) function 499
proportionate aggregates
Proportion Avg( ) 44–48
Proportion Sum( ) 44–48
Proportion WtAvg( ) 44–48
ProportionOverlap( ) function 499
PSD files
creating 540–542
Put statement 500

Q

quantiled ranges 205

R

RadioGroup controls 143
random file i/o
closing files 113
opening files 471
reading data 320–321
writing data 500
random numbers
Randomize statement 501
Rnd( ) function 526
Randomize statement 501
ranged thematic maps 204–708
RasterTableInfo( ) function 502
ReadControlValue( ) function 504
realtime applications 50–51
rectangle objects
Cartesian area of 96
Cartesian perimeter of 103
creating 207, 212

determining area of 79
determining perimeter of 485
modifying 70–74
querying the pen or brush style 433–438
storing in a new row 344–346
storing in an existing row 766
ReDim statement 507
Redistricting windows
closing 115
modifying 684, 692–702
opening 208
region objects
adding/removing nodes 74
Cartesian area of 96
Cartesian perimeter of 103
checking for data errors 446
converting objects to regions 146
creating 209–211
creating convex hull objects 146
determining area of 79
determining perimeter of 485
extracting a range of nodes from 282
modifying the pen or brush style 71–72
querying the pen or brush style 433–438
returning a buffer region 97
setting a centroid 74
storing in a new row 344–346
storing in an existing row 766
regional settings 627–628
Register Table statement 508
relational joins 553–555
Relief Shade statement 515
Reload Symbols statement 516
remote databases
creating new tables 579–581
refreshing linked tables 594
retrieving active database connection info 575
shutting down server connection 582
RemoteMapGenHandler procedure 516
RemoteMsgHandler procedure 517
RemoteQueryHandler function 518
Remove Cartographic Frame statement 519
Remove Map statement 520
removing
buttons 52–57
menu items 62–67
nodes 74
Rename File statement 521
Rename Table statement 522
reports
creating 211
loading 472
Reproject statement 523
reserved words 259
resizing
arrays 507–508
Resume statement 523
retrieving
column information
Server_ColumnInfo( ) 564
data source information
Server_DriverInfo( ) 583
number of columns in a results set
Server_NumCols( ) 592
number of toolkits
Server_NumDrivers( ) 593
records from an open table
Fetch statement 286
rows from a results set
Server Fetch 586–588
retrying on file access 626
returning
a connection number
Server_Connect 567–574
a coordinate system
ChooseProjection$( ) function 110
a date
FormatDate$( ) function 305
a pen width for a point size
PointsToPenWidth( ) function 486
a point size for a pen width
PenWidthToPoints( ) function 484
ODBC connection handle
Server_GetodbcHConn( ) function 588
ODBC statement handle
Server_GetodbcHStmt( ) function 589
pen width units
IsPenWidthPixels( ) function 352
RGB( ) function 524
Right$( ) function 525
Rnd( ) function 526
Rollback statement 527
Rotate( ) function 528
RotateAtPoint( ) function 529
rotated
map labels 652, 662, 664
symbols 393, 742
Round( ) function 530
rounded rectangle objects
Cartesian area of 96
Cartesian perimeter of 103
creating 212
modifying 70–74
querying the pen or brush style 433–438
storing in a new row 344–346
storing in an existing row 766

rounding off a number
Fix( ) function 297
Format$( ) function 303
Int( ) function 347
Round( ) function 530
RowID
after Find operations 122
with SelChangedHandler 121
rows in a Browser, positioning 609
rows in a table
deleting rows 248
end-of-table condition 271
inserting new rows 344–346
packing (purging deleted rows) 478
positioning the row cursor 286–288
selecting rows that satisfy criteria 553–555
updating existing rows 766
RPC (Remote Procedure Calls) 244–245
RTrim$( ) function 531
Ruler tool
closing Ruler window 115
modifying Ruler window 692–702
opening Ruler window 475
Run Application statement 531
Run Command statement 532
Run Menu Command statement 534
Run Program statement 536
runtime errors, trapping. ***See*** **error handling**

S

Save File statement 538
Save MWS statement 538
Save Window statement 540
Save Workspace statement 542
saving
changes to a table 124–129
linked tables 127
work to the database, Server Commit 566
scale of a map
determining 403
displaying 645
scope of variables
global 328
local 257–261
scroll bars
showing/hiding 700
scrolling
automatically 700
seamless tables
determine if table is seamless 749
prompt user to choose a sheet 326–328
turn seamless behavior on/off 691
SearchInfo( ) function 543
searching for map objects
at a point 546
processing search results 543–546
within a rectangle 547
SearchPoint( ) function 546
SearchRect( ) function 547–548
Second function 548
seconds, elapsed 757
Seek statement 549
Seek( ) function 548
SelChangedHandler procedure 549
Select statement 551
selectable map layers 648
selection
handling selection-changed event 121, 549
interrupted by Esc key 550
querying current selection 560
Select statement 551
SelectionInfo( ) function 560
self-intersections
checking in regions 446–447
sequential file i/o
closing files 113
opening files 471
reading data 343, 383
writing data 491, 785
Server Begin Transaction statement 561
Server Bind Column statement 562
Server Close statement 563
Server Commit statement 566
Server Create Map statement 576
Server Create Style statement 578
Server Create Table statement 579
Server Create Workspace statement 581
Server Disconnect statement 582
Server DriverInfo( ) function 583
Server Fetch statement 586
Server Link Table statement 590
Server Refresh statement 594
Server Remove Workspace statement 595
Server Rollback statement 595
Server Set Map statement 596
Server Versioning statement 597
Server Workspace Merge statement 599
Server Workspace Refresh statement 601
Server_ColumnInfo( ) function 564
Server_Connect( ) function 567
Server_ConnectInfo( ) function 575
Server_EOT( ) function 584
Server_Execute function 585
Server_GetODBCHConn( ) function 588
Server_GetODBCHStmt( ) function 589

Server_NumCols( ) function 592
Server_NumDrivers( ) function 593
server_string, defined 585
SessionInfo( ) function 603
Set Adornment statement 604
Set Application Window statement 607
Set Area Units statement 608
Set Browse statement 609
Set Cartographic Legend statement 611
Set Command Info statement 613
Set Connection GeoCode statement 614
Set Connection Isogram statement 617
Set CoordSys statement 619
Set Date Window statement 620
Set Datum Transform Version statement 621
Set Digitizer statement 622
Set Distance Units statement 624
Set Drag Threshold statement 625
Set Event Processing statement 625
Set File Timeout statement 626
Set Format statement 627
Set Graph statement 628
Set Handler statement 633
Set Layout statement 634
Set Legend statement 636
Set LibraryServiceInfo statement 639
Set Map statement 641
Set Map3D statement 677
Set Next Document statement 679
Set Paper Units statement 680
Set Path statement 681
Set PrismMap statement 682
Set ProgressBars statement 684
Set Redistricter statement 684
Set Resolution statement 686
Set Shade statement 687
Set Style statement 688
Set Table Datum statement 689
Set Table statement 689
Set Target statement 692
Set Window statement 692
Sgn( ) function 703
shade statement 704
shadow text 298–300
shapefiles 513
Shift key
- detecting shift-click 122
- effect on drawing tools 55–56
- selecting multiple list items 137–139

shortcut menus
- disabling 190
- example 67

Show/Hide menu commands 188
showing
- ButtonPads 52–57
- dialog box controls 58
- menu bar 409

shutting down the connection
- Server Disconnect 582

simulating a menu selection 534
Sin( ) function 716
small integer variables 257
smart redraw 645
snap tolerance
- controlling 701–702

snapping nodes 461–463
sorting rows in a table 558
sounds, beeping 86
Space$( ) function 717
spaces
- trimming from a string 387

spaces, trimming from a string 531
speed, improving
- animation layers 50
- screen updates 625
- table editing 689–691

SphericalArea( ) function 718
SphericalConnectObjects( ) function 719
SphericalDistance( ) function 719
SphericalObjectDistance( ) function 720
SphericalObjectLen( ) function 721
SphericalOffset( ) function 722
SphericalOffsetXY( ) function 723
SphericalPerimeter( ) function 724
splitting objects 463–465
spreadsheets
- using as tables 508–515

SQL Select command 551–559
SQL Server
- databases connection string attributes 570–573

Sqr( ) function 725
starting other applications
- Run Application statement 531
- Run Program statement 536

StaticText controls 144
statistical calculations
- average 44, 555–556
- count 44, 555–556
- min/max 44, 555–556
- quantile 205
- standard deviation 205
- sum 44, 555–556
- weighted average 44, 556–557

Statistics window
- closing 115
- modifying 692–702

opening 475
Status Bar help 54
Status bar help 161
StatusBar statement 726
Stop statement 727
Str$( ) function 728
street address, finding 292–295
string concatenation
& operator 845
+ operator 845
string functions
capitalization 375, 499, 763
comparison 730–731
converting codes to strings 111
converting strings to codes 81
converting strings to dates 627–628, 731–732
converting strings to numbers 627–628, 770
converting values to strings 728
extracting part of a string 376, 416–417, 525
finding a substring within a string 346
formatting a number 247, 303–306, 627–628
formatting based on locale 627–628
length of string 380
locale settings 627–628
pattern matching 382
repeated strings 729
spaces 717
trimming spaces from end 531
trimming spaces from start 387
string variables 257
String$( ) function 729
StringCompare( ) function 730
StringCompareIntl( ) function 731
StringToDate( ) function 731
StringToDateTime function 733
StringToTime function 733
structures 761
style override
add for layer labels 659–662
enable/disable for layer 658–659
enable/disable layer labels 665–666
layers 654–657
modify for layers 657–658
modify layer labels 662–665
remove from layer 658–659
remove layer labels 665–666
StyleAttr( ) function 734
StyleOverrideInfo( ) function 736
sub procedures. ***See*** **procedures**
Sub...End Sub statement 739
subtotals, calculating 555–556
Sum( ) aggregate function 555–556
symbol
variables 258
Symbol clause 740
symbol styles
creating 390, 393, 395
modifying an object's style 71–72
querying an object's style 433–438
querying parts of 733–736
reading current style 227
reloading symbol sets 516
setting current style 688–689
Symbol clause defined 740
SYMBOL.MBX utility
custom symbols 516
SymbolPicker controls 140
SystemInfo( ) function 744

T

TAB files, storing metadata in 412–414
tab order 252
table names
determining _ in Browse or Graph window 780
determining from file 480
determining table name from number 746–751
special names for Cosmetic layers 780
special names for Layout windows 780
table structure
3DMap 183–185
adding/removing columns 76–78
determining how many columns 426, 746–751
making a table mappable 182
making an ODBC table mappable 576–578
TableInfo( ) function 746
TableListInfo( ) function 751
TableListSelectionInfo() function 753
tables, closing
Close All statement 112
Close Table statement 114
tables, copying 124–129
tables, creating
creating a new table 214
importing a file 337–342
on remote databases 579–581
using a spreadsheet or database 508–515
tables, deleting 267
tables, importing 337–342
tables, modifying
adding columns 42–48, 76–78
adding metadata 412–414
adding rows 344–346
creating an index 179
deleting a table's objects 266
deleting an index 266

deleting columns 76–78
deleting rows or objects 248
discarding changes 527
optimizing edit operations 689–691
packing 478
renaming 522
saving changes 124–129
setting a map's default view 649
setting a map's projection 124–129
setting to read-only 689–691
sorting rows 558
updating existing rows 766
tables, opening 473–475
tables, querying
column information 116–118
directory path 749
end-of-table condition 271
finding a map address 292–295
joining 553–555
metadata 325, 412–414
number of open tables 427
objects at a point 546
objects in a rectangle 547–548
positioning the row cursor 271, 286–288
SQL Select 551–559
table information 746–751
Tan( ) function 753
TempFileName$( ) function 754
temporary columns 42
Terminate Application statement 755
text files
See also file input/output, files 38
using as tables 508–515
text objects
creating 219–221
modifying 70–74
querying the font style or string 435
storing in a new row 344–346
storing in an existing row 766
text styles
See font styles
TextSize( ) function 756
thematic maps
bar chart maps 714–716
counting themes in a 3D map window 399–402
counting themes in a map window 402–407
creating arrays of ranges 204–206
creating arrays of styles 213–214
dot density maps 710–711
grid surface maps 176–179
modifying 687
pie chart maps 712–714
quantiled ranges 205
ranged maps 704–708
thinning objects 461–463
thousands separators 247, 306, 627–628
TIFF files
creating 540–542
time delay when user drags mouse 625
Time( ) function 756
Timer( ) function 757
ToolHandler procedure 757
tooltip help 54, 161
totals, calculating 555–556
transparent fill patterns 90
trapping errors. ***See*** **error handling**
TriggerControl( ) function 759
trigonometric functions
arc-cosine 39
arc-sine 82
arc-tangent 84
cosine 151, 154
sine 716
tangent 753
trimming spaces
from end of string 531
from start of string 387
TrueFileName$( ) function 760
TrueType fonts, using as symbols 393
TrueType symbols 740–744
Type statement 761

U

UBound( ) function 762
UCase$( ) function 763
unchecking
dialog box check boxes (custom) 58
dialog box check boxes (standard) 60–62
menu items 68–70
underlined text 298–300
UnDim statement 763
undo system, disabling 689–691
UnitAbbr$( ) function 764
UnitName$( ) function 765
units of measure
abbreviated names 764
area 608
distance 624
full names 765
paper 680
Unlink statement 766
unselecting 115
Update statement 766
Update Window statement 767
upper case, converting to 763

URL 639, 768
USNGToPoint(string) 769

V

Val( ) function 770
variable length strings 257
variables
arrays 258, 507–508, 762
custom types 761
global variables 328
initializing 261
list of types 257–258
local variables 257–261
reading another application's variables 328
restrictions on names 259
strings variables 259
undefining 763
version number
.MBX version 744
MapInfo version 745
voronoi polygons
Create Object statement 196

W

Weekday( ) function 771
weighted averages 556–557
Add Column statement 44
WFS Refresh Table statement 772
While...Wend statement 772
wildcard
matching 382
WinChangedHandler procedure 774
WinClosedHandler procedure 775
WindowID( ) function 776
WindowInfo( ) function 777
windows operating system, 16- v. 32-bit 745
windows, closing
Close Window statement 115
preventing user from closing windows 699
windows, modifying
adding map layers 48–51
browser windows 609
forcing windows to redraw 767
general window settings 692–702
graph windows 628–633
layout windows 634–636
legend window 636–639
map windows 641–671
redistrict windows 684–686
removing map layers 520
windows, opening
Browse statement 86–87
Create Redistricter statement 208
Graph statement 330
Layout statement 374
Map statement 396
Open Window statement 475
windows, printing
to a file 540–542
to an output device 492
windows, querying
3D map window settings 399–402
general window settings 777–784
ID of a window 776
ID of front window 309
map window settings 364–371, 402–407
number of document windows 427
total number of windows 423
WinFocusChangedHandler procedure 784
WKS files, opening 508–515
WMF files, creating 540–542
workspaces
loading 531
workspaces, saving
Save Workspace statement 542
Write # statement 785
WtAvg( ) aggregate function 556–557

X

XCMDs 245
XFCNs 240–241
XLS files, opening 508–515
XML Library
MIXmlAttributeListDestroy( ) procedure 825
MIXmlDocumentCreate( ) function 825
MIXmlDocumentDestroy( ) procedure 826
MIXmlDocumentGetNamespaces( ) function 826
MIXmlDocumentGetRootNode( ) function 827
MIXmlDocumentLoad( ) function 827
MIXmlDocumentLoadXML( ) function 828
MIXmlDocumentLoadXMLString( ) function 829
MIXmlDocumentSetProperty( ) function 830
MIXmlGetAttributeList( ) function 831
MIXmlGetChildList( ) function 831
MIXmlGetNextAttribute( ) function 832
MIXmlGetNextNode( ) function 833
MIXmlNodeDestroy( ) procedure 833
MIXmlNodeGetAttributeValue( ) function 834
MIXmlNodeGetFirstChild( ) function 834
MIXmlNodeGetName( ) function 835
MIXmlNodeGetParent( ) function 836
MIXmlNodeGetText( ) function 836
MIXmlNodeGetValue( ) function 837

MIXmlNodeListDestroy( ) procedure 837
MIXmlSCDestroy( ) procedure 838
MIXmlSCGetLength( ) function 838
MIXmlSCGetNamespace( ) function 839
MIXmlSelectNodes( ) function 839
MIXmlSelectSingleNode( ) function 840

Y

Year( ) function 786