



MapInfo MapBasic v. 8.0

Reference Guide

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 MapInfo Corporation, One Global View, Troy, New York 12180-8399.

© 2005 MapInfo Corporation. All rights reserved. MapInfo, MapInfo Professional, MapBasic, StreetPro and the MapInfo logo are trademarks of MapInfo Corporation and/or its affiliates.

MapInfo Corporate Headquarters:

Voice: (518) 285-6000

Fax: (518) 285-6060

Sales Info Hotline: (800) 327-8627

Government Sales Hotline: (800) 619-2333

Technical Support Hotline: (518) 285-7283

Technical Support Fax: (518) 285-6080

Contact information for North American offices is located at: http://www.mapinfo.com/company/company_profile/index.cfm.

Contact information for worldwide offices is located at: http://www.mapinfo.com/company/company_profile/worldwide_offices.cfm.

Contact information for European and Middle East offices is located at: <http://www.mapinfo.co.uk>.

Contact information for Asia Pacific offices is located at: <http://www.mapinfo.com.au>.

Adobe Acrobat® is a registered trademark of Adobe Systems Incorporated in the United States.

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.

libtiff © 1988-1995 Sam Leffler, copyright © Silicon Graphics, Inc.

libgeotiff © 1995 Niles D. Ritter.

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

HIL - Halo Image Library™ © 1993, Media Cybernetics Inc. Halo Imaging Library is a trademark of Media Cybernetics, Inc.

Portions thereof LEAD Technologies, Inc. © 1991-2005. All Rights Reserved.

Portions © 1993-2005 Ken Martin, Will Schroeder, Bill Lorensen. All Rights Reserved.

Blue Marble © 1993-2005

ECW by ER Mapper © 1993-2005

VM Grid by Northwood Technologies, Inc., a Marconi Company © 1995-2004™.

Portions © 2005 Earth Resource Mapping, Ltd. All Rights Reserved.

MrSID, MrSID Decompressor and the MrSID logo are trademarks of LizardTech, Inc. used under license. Portions of this computer program are (c) 1995-1998 LizardTech and/or the university of California or are protected by US patent nos. 5,710,835; 5,130,701; 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.

Universal Translator by Safe Software, Inc. © 2004.

Crystal Reports ® is proprietary trademark of Crystal Decisions. All Rights Reserved.

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 2005

Table of Contents

Chapter 1: New and Enhanced MapBasic Statements and Functions.....	5
Enhanced MapBasic Functions and Statements.....	22
Enabling Transparent Patterns on Same Layer.....	32
Export Windows to Additional Formats	32
Chapter 2: Introduction	33
Language Overview	34
MapBasic Fundamentals	34
Variables	34
Looping and Branching	35
Output and Printing	35
Procedures (Main and Subs).....	35
Error Handling	35
Functions.....	35
Custom Functions	36
Data-Conversion Functions	36
Date and Time Functions.....	36
Math Functions	37
String Functions.....	37
Working With Tables.....	38
Creating and Modifying Tables	38
Querying Tables.....	38
Working With Remote Data	39
Working With Files (Other Than Tables).....	40
File Input/Output	40
File and Directory Names	40
Working With Maps and Graphical Objects	41
Creating Map Objects	41
Modifying Map Objects	41
Querying Map Objects.....	42
Working With Object Styles	42
Working With Map Windows	43
Creating the User Interface	43
ButtonPads (ToolBars).....	43
Dialog Boxes	44
Menus	44
Windows	44
System Event Handlers	45

Communicating With Other Applications	45
DDE (Dynamic Data Exchange; Windows Only)	45
Integrated Mapping.	45
Special Statements and Functions	46
A – Z Reference	46
Appendix A: Character Code Table	586
Appendix B: Summary of Operators	588
Numeric Operators	589
Comparison Operators	590
Logical Operators	590
Geographical Operators	591
Precedence	592
Automatic Type Conversions	592
Appendix C: MapBasic Definitions File	594
Index	615

New and Enhanced MapBasic Statements and Functions

These are the new statements and functions available for the MapInfo Professional 8.0 product.

Sections in this Appendix:

- ♦ **New MapBasic Functions and Statements 6**
- ♦ **Enhanced MapBasic Functions and Statements 22**

New MapBasic Functions and Statements

CartesianConnectObjects() function

Purpose

Returns an object representing the shortest or longest distance between two objects.

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.

Returns

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.

`CartesianClosestPoints()` 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.

CartesianObjectDistance() function

Purpose

Returns the distance between two objects.

Syntax

```
CartesianObjectDistance(object1, object2, unit_name)
```

object1 and *object2* are object expressions.

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

Returns

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.

ConnectObjects() function**Purpose**

Returns an object representing the shortest or longest distance between two objects.

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.

Returns

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.

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.

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 optional parameter representing 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 represents a table of objects that you want to find farthest distances from.

fromvar represents a MapBasic variable representing an object that you want to find the farthest distances from.

totable represents a table of objects that you want to find farthest distances to.

intotable represents a table to place the results into.

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. The entire Ignore clause is optional, as are the Min and Max subclauses within it (e.g., only a Min or only a Max, or both may occur).

Normally, if one object is contained within another object, the distance between the objects is zero. For example, if the From table is WorldCaps and the To table 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.

The Data clause can be used to mark which *fromtable* object and which *totable* object the result came from.

Description

Every object in the *fromtable* is considered. For each object in the *fromtable*, the farthest object in the *totable* is found. If *N* is present, then the *N* farthest objects in *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 present, 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 (i.e., if *N* is greater than 1), then objects of the same distance will fill subsequent slots. If the tie exists at the second farthest object, and 3 objects are requested, then the object 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 (i.e., a distance of 100 should only occur in the first pass and never in the second pass).

Data Clause

```
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 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). 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
```

It is currently not possible to fill in a column in the *intotable* with the distance. However, this can be easily accomplished after the Nearest operation is completed by using the **TABLE > UPDATE COLUMN...** functionality from the menu or by using the Update MapBasic statement.

See Also

Nearest statement, ObjectDistance() function, ConnectObjects() 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.

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 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.

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. The entire Ignore clause is optional, as are the Min and Max subclauses within it (e.g., only a Min or only a Max, or both may occur).

Normally, if one object is contained within another object, the distance between the objects is zero. For example, if the From table is WorldCaps and the To table 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.

The Data clause can be used to mark which *fromtable* object and which *totable* object the result came from.

Description

Every object in the *fromtable* is considered. For each object in the *fromtable*, the nearest object in the *totable* is found. If *N* is present, 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 present, 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 (i.e., 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.

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 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 (i.e., a distance of 100 should only occur in the first pass and never in the second pass).

Data Clause

```
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 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). 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
```

It is currently not possible to fill in a column in the *intotable* with the distance. However, this can be easily accomplished after the Nearest operation is completed by using the **TABLE > UPDATE COLUMN...** functionality from the menu or by using the `Update MapBasic` 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 selection To atm Into result Data where=buisness,address=address
```

If we wanted to find the closest five ATM machines to our current location:

```
Nearest 5 From 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 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 customer To coastline Into result Data
who=customer.name,where=customer.address,coast_loc=coastline.county,type=coastli
ne.designation
```

If we don't care about customer locations which are greater than 30 miles from any coastline:

```
Nearest From customer To coastline Into result Ignore Max 30 Units "mi" Data
who=customer.name,where=customer.address,coast_loc=coastline.county,type=coastli
ne.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 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, **ObjectDistance() function**, **ConnectObjects() function**

ObjectDistance() function

Purpose

Returns the distance between two objects.

Syntax

```
ObjectDistance(object1, object2, unit_name)
```

object1 and *object2* are object expressions.

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

Returns

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.

ObjectNodeM() function

Purpose

Returns the m-value of a specific node in a region, polyline or multipoint object.

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 *object* does not support m values or m-value for this node is not defined, then, error is set.

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

ObjectNodeZ () function**Purpose**

Returns the z-coordinate 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

Server Create Workspace statement

Purpose

Creates a new workspace in the database (Oracle 9i or later).

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 GARYG in the database.

```
Dim hdbc As Integer
hdbc = Server_Connect("ORAINET", "SRVR=TROJNY;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=TROJNY;UID=MIUSER;PWD=MIUSER")
Server hdbc Create Workspace "MBPROG" Description "MapBasic project" Parent
"MIUSER"
```

See also

[Server Remove Workspace statement](#), [Server Versioning 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).

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=TROJNY;UID=MIUSER;PWD=MIUSER")
Server hdbc Remove Workspace "MIUSER"
```

See also

[Server Create Workspace statement](#)

Server Versioning statement

Purpose

Version-enable or disable 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.

Syntax

```
Server ConnectionNumber Versioning
{
  ON
    [History {SRV_WM_HIST_NONE|SRV_WM_HIST_OVERWRITE|SRV_WM_HIST_NO_OVERWRITE}]
  | 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.

When version-enabling a table (ON), *History* is an optional parameter.

History clause specifies how to track modifications to *ServerTableName*, i.e., 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. 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.

When disabling a version-enabled table (OFF), *Force* is an optional parameter.

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).

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 identifies the rows to be merged into the parent workspace. The clause itself should omit the WHERE keyword.

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 following 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.

Interactive

In the event of a conflict, MapInfo displays the Conflict Resolution dialog box. 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.

Note: Due to a system limitation, this option is not available if the server is Oracle9i.

Automatic StopOnConflict

In the event of a conflict, MapInfo will stop here. (This is also the default behavior if the statement does not include an Interactive clause or an Automatic clause.)

Automatic RevertToBase

In the event of a conflict, MapInfo 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 parameter must be followed by UseParent or UseCurrent.)

Automatic UseCurrent

In the event of a conflict, MapInfo uses the child workspace values.

Automatic UseParent

In the event of a conflict, MapInfo uses the parent workspace values.

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.

Examples

The following example merge changes to USA where MI_PRINX=5 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).

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.

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 following 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.

Interactive

In the event of a conflict, MapInfo displays the Conflict Resolution dialog box. The conflicts will be resolved one by one based on user choices. After all the conflicts are resolved, the table is refreshed from its parent based on the user's choices.

Note: Due to a system limitation, this option is not available if the server is Oracle9i.

Automatic StopOnConflict

In the event of a conflict, MapInfo will stop here. (This is also the default behavior if the statement does not include an Interactive clause or an Automatic clause.)

Automatic RevertToBase

In the event of a conflict, MapInfo 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.)

Automatic UseCurrent

In the event of a conflict, MapInfo uses the child workspace values.

Automatic UseParent

In the event of a conflict, MapInfo uses the parent workspace values.

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.

Examples

The following example refreshes MIUSER by applying changes made to USA where MI_PRINX=5 in its parent workspace.

```
Server Workspace Refresh
Table "GWMUSA2"
Where "MI_PRINX = 60"
Automatic UseParent
```

See also

[Server Workspace Merge statement](#)

SphericalConnectObjects() function

Purpose

Returns an object representing the shortest or longest distance between two objects.

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.

Returns

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.

SphericalObjectDistance() function

Purpose

Returns the distance between two objects.

Syntax

```
SphericalObjectDistance(object1, object2, unit_name)
```

object1 and *object2* are object expressions.

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

Returns

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.

Enhanced MapBasic Functions and Statements

Add Cartographic Frame statement

```
[ 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
[ Using
    [ Column { column | object [ FromMapCatalog { On | Off }]] ]
...

```

The syntax indicates that if you specify `Using Column object`, there is a new `FromMapCatalog` clause you can use that is only applicable to live access tables.

`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` (i.e., 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 then the behavior is to revert to the default behavior for live tables, which is to get the default styles from the MapCatalog (`FromMapCatalog ON`).

Examples

Creating on live access table that supports per record styles with map styles:

```
Create Cartographic Legend From Window 168811024
Scrollbars On
Portrait Style Size Large
Default Frame
Title "# Legend"
Font ("Arial",0,10,0)
Default Frame Style "%"
Font ("Arial",0,8,0)
Frame From Layer 1
Title "nyalbak Legend"
Using column object FromMapCatalog OFF label default
```

Creating on live access table with MapCatalog:

```
Create Cartographic Legend From Window 168811024
Scrollbars On
Portrait Style Size Large
Default Frame
Title "# Legend"
Font ("Arial",0,10,0)
Default Frame Style "%"
Font ("Arial",0,8,0)
Frame From Layer 1
Title "tony_nyalbak Legend"
Using column object FromMapCatalog ON label default
```

Creating on live access table with MapCatalog:

```
Create Cartographic Legend From Window 168811024
Scrollbars On
Portrait Style Size Large
Default Frame Title "# Legend"
Font ("Arial",0,10,0)
Default Frame Style "%"
Font ("Arial",0,8,0)
Frame From Layer 1 Title "nyalbak Legend"
Using column class label default
```

Workspace Behavior

When you save to a workspace, the new `FromMapCatalog OFF` clause is written to the workspace when specified. This requires the workspace to be 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.

Alter Object statement

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
  }
}
```

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

Create Cartographic Legend statement

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 } ] } ]
...

```

The syntax indicates that if you specify Using Column object, there is a new FromMapCatalog clause you can use that is only applicable to live access tables.

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 (i.e., 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).

Create Collection statement

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_polygons is the number of polygons inside the Collection object.

num_sections specifies how many sections the multi-section polyline will contain.

Create Pline statement

Syntax

```
Create Pline
  [ Into { Window window_id | Variable var_name } ]
    [ Multiple num_sections ]
    num_points
    ( x1, y1 ) ( x2, y2 ) [...]
    [ Pen ... ]
    [ Smooth ]
```

num_sections specifies how many sections the multi-section polyline will contain.

Create Region statement

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 ) ]
```

num_polygons specifies the number of polygons that will make up the region (zero or more).

Commit Table statement

Here is the syntax with the new `ConvertObjects` keyword in **bold**:

```
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
    } ]
    [ CoordSys... ]
    [ Version version ] ]
  [ { Interactive | Automatic commit_keyword } ]
  [ConvertObjects {ON | OFF | INTERACTIVE }]
```

ExtractNodes() function

```
ExtractNodes( object, polygon_index, begin_node, end_node, b_region )
```

`polygon_index` is an Integer value, 1 or larger: for region objects. This indicates which polygon (for regions) or section (for polylines) to query.

Import statement

Syntax

```
Import file_name
  [ Type "GML21" ]
  [ Layer layer_name ]
  [ Into table_name ]
  [ Overwrite ]
  [ Coordsys clause ]
```

file_name is the name of the GML 2.1 file to import.

Type is "GML21" for GML 2.1 files.

layer_name is the name of the GML layer.

table_name is the MapInfo table name.

`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.

Note: If the Coordsys clause does not match the projection of the GML file, your data may not import correctly. The coordsys must match the coordsys of the data in the GML file. It will not transform the data from one projection to another.

Example

```
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
```

The following functions have been updated for this release.

ObjectGeography() function

attribute setting	Return value (Float)
OBJ_GEO_POINTZ	z-value of a Point object.
OBJ_GEO_POINTM	m-value of a Point object.

If object does not support z/m values or z/m-value for this node is not defined, then an error is thrown.

ObjectInfo() function

Syntax

```
ObjectInfo( object, attribute )
```

object is an Object expression

attribute is an integer code specifying which type of information should be returned.

Return value

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 *N*th polygon of a region or the *N*th section of a polyline.

Note: 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.

attribute setting	Return value
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. Return empty string if units are not specified.
OBJ_INFO_HAS_Z(14)	Logical, indicating whether object has Z values.
OBJ_INFO_HAS_M(15)	Logical, indicating whether object has M values.

ObjectNodeX() function

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).

ObjectNodeY() function

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).

Register Table statement

Syntax

```
Register Table source_file
.
.
.
    Type "ODBC" [ Cache { On | OFF } ]
    Connection { Handle ConnectionNumber | ConnectionString }
    Toolkit toolkitname
    Table SQLQuery
    [Versioned {Off | On}]
    [Workspace WorkspaceName]
    [ParentWorkspace ParentWorkspaceName]
    ...
```

Versioned indicates if the table to be opened is an version-enabled (ON) table or not (OFF).

WorkspaceName is the name of the current workspace in which the table will be operated. The name is case sensitive.

ParentWorkspaceName is the name of parent workspace of the current workspace.

Note: In order to have this statement be effective, the table has to be version-enabled, that is, *Versioned* is set ON.

Examples

The following example create a tab file and then open the tab file.

```
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
```

Note: **INTERACTIVE** is not a valid parameter to use when registering SHP files.

See Also

Server Create Workspace statement

Set Cartographic Legend statement

Syntax

```
Set Cartographic Legend
  [ Window window_id ]
  [Refresh]
  [Portrait | Landscape]
  [Columns number_of_columns | Lines number_of_lines]
  ...
```

number_of_columns specifies the width of the legend.

number_of_lines specifies the height of the legend.

Set Legend statement

Purpose

The Set Legend command is used to provide custom ordering of legend categories or items. The new syntax is in **bold**.

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 | Fontsize}]
  [ Columns number_of_columns ]
  [ Ascending { On | Off } | Order { Ascending | Descending | Custom } ]
  [ Ranges { [Font . . . ]
    [ Range { range_identifier | default } ]
    range_title [ Display { On | Off } ] }
  [ , . . . ]
]
[ , . . . ]

```

There are four new clauses: Order, Range, Style Size, and Columns. When you want custom order, include `Order Custom` in the MapBasic statement as well as a range identifier for each category in the theme. The order of ranges dictates the order of categories in the legend. 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 Order and Range clauses will increase the workspace version to 8.0. 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 a new way to specify legend label order of ascending or descending as well as new custom order. However, the original Ascending { On | Off } clause is still available for backwards compatibility. You can use either the new Order clause, or the old 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.

Examples

The example workspace below needs the following shade statement:

```
shade 1 with Province_Name values
  "Alberta" Brush (2,16711680,16777215) Pen (1,2,0) ,
  "British Columbia" Brush (2,65280,16777215) Pen (1,2,0) ,
  "Manitoba" Brush (2,255,16777215) Pen (1,2,0) ,
  "New Brunswick" Brush (2,16711935,16777215) Pen (1,2,0) ,
  "Newfoundland" Brush (2,16776960,16777215) Pen (1,2,0) ,
  "Northwest Territories" Brush (2,65535,16777215) Pen (1,2,0) ,
  "Nova Scotia" Brush (2,8388608,16777215) Pen (1,2,0) ,
  "Nunavut" Brush (2,32768,16777215) Pen (1,2,0) ,
  "Ontario" Brush (2,128,16777215) Pen (1,2,0) ,
  "Prince Edward Island" Brush (2,8388736,16777215) Pen (1,2,0) ,
  "Quebec" Brush (2,8421376,16777215) Pen (1,2,0) ,
  "Saskatchewan" Brush (2,32896,16777215) Pen (1,2,0) ,
  "Yukon Territory" Brush (2,16744576,16777215) Pen (1,2,0)
default Brush (1,0,16777215) Pen (1,2,0) # color 1 #
```

The Set Legend statement includes the Order Custom tokens and a Range identifier for each category. The Range identifier is the same string found in the shade statement and the order of ranges is what is displayed in the Legend. (New information is in **bold**.)

```
set legend
  layer 1
    display on
    shades on
    symbols off
    lines off
    count on
    title auto Font ("Arial",0,9,0)
    subtitle auto Font ("Arial",0,8,0)
    order custom
    ranges Font ("Arial",0,8,0)
      range "Prince Edward Island" auto display on ,
      range "Northwest Territories" auto display on ,
      range "British Columbia" auto display on ,
      range "Yukon Territory" auto display on ,
      range "New Brunswick" auto display on ,
      range "Newfoundland" auto display on ,
      range "Saskatchewan" auto display on ,
      range "Nova Scotia" auto display on ,
      range "Manitoba" auto display on ,
      range "Nunavut" auto display on ,
      range "Ontario" auto display on ,
      range "Quebec" auto display on ,
      range "Alberta" auto display on ,
      range default auto display off
```

Enabling Transparent Patterns on Same Layer

In order to facilitate a multi-thematic analysis on a particular layer, transparent patterns are necessary. To facilitate this, the Shade statement and the Set Shade statement now have the addition of a `Style Replace` clause for use with for Range and Individual Value themes. The syntax for the new clause is as follows:

```
{Style Replace { On | Off } }
```

`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.

Export Windows to Additional Formats

The Save Window statement now supports three additional formats for image export. The new values for `type` include: "TIFFG4", "TIFFLZW", and "GIF".

Examples

```
save window frontwindow() as "untitled.gif" type "gif"
save window frontwindow() as "untitled.tif" type "tiffg4"
save window frontwindow() as "untitled.tif" type "tiffllzw"
```

TableInfo() function

attribute code	TableInfo() returns
TAB_INFO_SUPPORT_MZ	Logical result: TRUE if table supports M and Z values.
TAB_INFO_Z_UNIT_SET	Logical result: TRUE is unit is set for Z-values.
TAB_INFO_Z_UNIT	String result: indicates distance units used for Z-values. Return empty string if units are not specified.

Introduction

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*.

In this chapter...

♦ Type Conventions	34
♦ Language Overview	34
♦ MapBasic Fundamentals	34
♦ Functions	35
♦ Working With Tables	38
♦ Working With Files (Other Than Tables)	40
♦ Working With Maps and Graphical Objects	41
♦ Creating the User Interface	43
♦ Communicating With Other Applications	45
♦ Special Statements and Functions	46
♦ A – Z Reference	46

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, Integer, Pen, Object	Non-bold words with the first letter capitalized are usually special procedure names or variable types.
<i>table, handler, window_id</i>	Italicized words represent parameters to MapBasic statements. When you construct a MapBasic statement, you must supply an appropriate expression for each parameter.
[<i>window_id</i>], [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 this font (Courier).

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 an MGRS coordinate:	PointToMGRS\$()
Convert a MGRS coordinate to a point object:	MGRSToPoint()

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:	NumberToDate(), StringToDate()

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 an MGRS coordinate:	PointToMGRS\$()
Convert a MGRS coordinate to a point object:	MGRSToPoint()

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:	Commit 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 Commit
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 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:	ObjectsDisaggregate statement, Objects Snap statement, Objects Clean statement

Querying Map Objects

Return calculated values:	Area(), Perimeter(), Distance(), ObjectLen(), Overlap(), AreaOverlap(), ProportionOverlap()
Return coordinate values:	ObjectGeography(), MBR(), ObjectNodeX(), ObjectNodeY(), 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, Unit-Abbr\$(), 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 StyleTextSize()
Return part of a style:	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(), LayerInfo()
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()

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 ButtonPads As Default

Dialog Boxes

Display a standard dialog:	Ask(), Note, ProgressBar, FileOpenDlg(), FileSaveAsDlg(), GetSeamlessSheet()
Display a custom dialog:	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:	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
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(), Command-Info(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

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

A – Z Reference

The next 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 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.

See Also

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.

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.

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.

Syntax

```
Acos ( num_expr )
```

num_expr is a numeric expression between one and minus 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 minus one, the expression *num_expr* should represent a value no larger than one and no smaller than minus 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.

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()** and **WindowId()** functions.

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, e.g., `MakePen(width, pattern, color)`. If a default border pen is defined, then it will 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*.

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** or previous 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 be 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** (i.e., 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 then 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.

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**; 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** and **Update** statements.

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, the 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

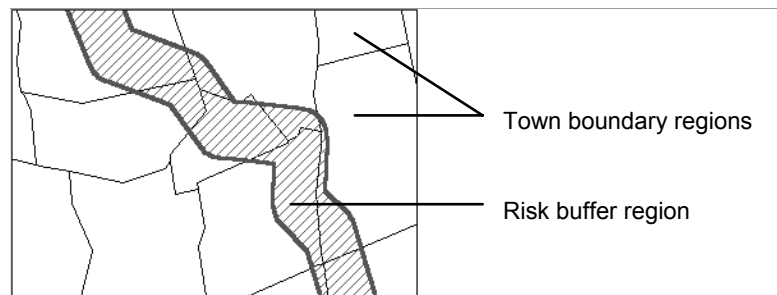
If you specify a **From** clause, you can calculate values for the new column by aggregating data from the second table. To perform data aggregation, specify a **Set To** clause that includes an aggregate function.

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 <i>weight_col</i> value have more of an impact than rows having a small <i>weight_col</i> 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

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 Towns table, containing town boundary regions and demographic information (e.g., population) about each town. 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.



Given these two tables, you might want to calculate the population that lives within the risk region. If half of a town's area falls within the risk region, you will consider half of that town's population to be at risk; if a third of a town's area falls within the risk region, you will consider a third of that town'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 towns
    Set To Proportion Sum(town_pop)
    Where Intersects
```

For each town that is at least partly within the risk region, MapInfo Professional adds some or all of the town's town_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 very 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 (e.g., 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 Towns table contains a column, median_age, that indicates the median age in each town.

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

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

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

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.

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.

See Also

[Alter Table statement](#), [Update statement](#)

Add Map statement

Purpose

Adds another layer to a Map window.

Syntax

```
Add Map
  [ Window window_id ]
  [ Auto ]
  Layer table [ , table ... ]
  [ Animate ]
```

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

Description

The **Add Map** statement adds one or more open tables to a Map window. MapInfo Professional then automatically redraws the Map window, unless you have suppressed redraws through a **Set Event Processing Off** statement or **Set Map...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()** and **WindowID()** functions. 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 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 *table* layer will be drawn last. You can then use the **Set Map** statement to alter the order of layers in the Map window.

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 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.

Note: 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
```

If the **Add Map** statement specifies two or more layers and it includes the **Animate** keyword, the first layer named becomes the animation layer, and the remaining layers are added to the map as conventional layers.

To terminate the animation layer processing, issue a **Remove Map ... 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
```

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
```


Note: 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 **Open Window 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 } ]
  [ 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 ODBC

button_id is a custom button's unique identification number

handler_num is an Integer handler code (e.g., M_TOOLS_RULER) from MENU.DEF

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)

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 (e.g., M_FILE_OPEN); 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** syntax, see **Create ButtonPad**.

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

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 *n* specifies the icon to appear on the button; *n* can be one of the standard MapInfo icon codes listed in ICONS.DEF (e.g., 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.

Cursor *n* specifies the shape the mouse cursor should adopt whenever the user chooses a ToolButton tool; *cursor_code* is a code (e.g., 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.

dm_code specifies whether the user can click and drag, or only click with the tool; *dm_code* is a code (e.g., DM_CUSTOM_LINE) 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.

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.

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); **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** statements 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, the parameter *n* must refer to one of the icon codes listed in ICONS.DEF (e.g., MI_ICON_RULER).

Note: 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 (e.g., MI_CURSOR_CROSSHAIR or MI_CURSOR_ARROW). The procedure for specifying a custom cursor is similar to the procedure for specifying a custom icon.

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.

<i>dm_code parameter</i>	Description
DM_CUSTOM_POINT	The user cannot drag while using the custom tool.
DM_CUSTOM_LINE	As the user drags, a line connects the cursor with the location where the user clicked.
DM_CUSTOM_RECT	As the user drags, a rectangular marquee appears.
DM_CUSTOM_CIRCLE	As the user drags, a circular marquee appears.
DM_CUSTOM_ELLIPSE	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.
DM_CUSTOM_POLYGON	The user may draw a polygon. To retrieve the object drawn by the user, use the function call: Command-Info(CMD_INFO_CUSTOM_OBJ)
DM_CUSTOM_POLYLINE	The user may draw a polyline. To retrieve the object drawn by the user, use the function call: Command-Info(CMD_INFO_CUSTOM_OBJ)

All of the draw modes except for DM_CUSTOM_POINT 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**.

Note: 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
```

...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

```

Note: The **Separator** keyword, which inserts space between the last button on the Tools ButtonPad and the new "+" button.

See Also

Alter Button statement, **ButtonPadInfo() function**, **Create ButtonPad 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.)

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()** and **WindowId()** functions.

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 Legend** 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.

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

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; accordingly, the **Alter Control** statement should only be issued while a dialog is active (i.e. from within a handler procedure that is called by one of the dialog controls). If there are two or more nested dialogs on the screen, the **Alter Control** statement only affects controls within the topmost dialog.

The *id_num* specifies which dialog 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 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 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.

Note: 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 containing two check-boxes, 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
```

See Also

Dialog statement, Dialog Preserve statement, ReadControlValue() function

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 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 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.

Note: Use this statement to modify 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’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 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.

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

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), 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*.

Check box: 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. 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 **SystemInfo()** to determine the MapInfo Professional version number.

See Also

Alter Control statement, Dialog statement

Alter Menu statement

Purpose

Adds or removes items from an existing menu.

Syntax1

```
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 (e.g., "File").

menu_id is a standard Integer menu ID from 1 to 22; 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** statements.

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.

Syntax2

```
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 22; 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. Note that if the application is running on a non-English language version of MapInfo, and if the menu names have been translated, the **Alter Menu** statement must specify the translated version of the menu name.

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 22 (one represents the File menu).

The following table lists the names and ID numbers of all standard MapInfo Professional menus.

Note: Menus 16 through 22 are shortcut menus, which appear if the user clicks with the right mouse button. Shortcut menus are only available on Windows.

Menu Name	Description
"File"	File menu (can also be referred to as ID 1)
"Edit"	Edit menu (ID 2)
"Objects"	Objects menu (ID 14)
"Query"	Query menu (ID 3)
"Table"	Table menu (ID 15)
"Options"	Options menu (ID 5)
"Window"	Window menu (ID 6)
"Help"	Help menu (ID 7)
"Browse"	Browse menu (ID 8). Ordinarily, this only appears when a Browser window is the active window. See WinSpecific, below.
"Map"	Map menu (ID 9). Ordinarily, this menu is only available when a Map window is active.
"Graph"	Graph menu (ID 11). Available when a Graph window is active.
"Layout"	Layout menu (ID 10). Available when a Layout window is active.
"Redistrict"	Redistrict menu (ID 13). Available when a Districts Browser is active.
"MapBasic"	MapBasic menu (ID 12). Available when the MapBasic window is active.
"Tools"	Tools menu (ID 4). A menu used by MapBasic utilities such as ScaleBar.
"WinSpecific"	The generic name for the window-specific menu, which changes dynamically depending on which type of window is the active window.
"Raster"	The hierarchical menu located on the Table menu.
"Maintenance"	The hierarchical menu located on the Table menu.
"Default-Shortcut"	The default shortcut menu. This menu appears if the user right-clicks on a window that does not have its own shortcut menu defined. (ID16)

Menu Name	Description
"Mapper-Shortcut"	The Map window shortcut menu. (ID 17)
"Browser-Shortcut"	The Browse window shortcut menu. (ID 18)
"Layout-Shortcut"	The Layout window shortcut menu. (ID 19)
"Grapher-Shortcut"	The Graph window shortcut menu. (ID 20)
"CmdShort-cut"	The MapBasic window shortcut menu. (ID 21)
"Redistrict-Shortcut"	The Redistricting shortcut menu; available when the Districts Browser is active. (ID 22)

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** statements.

```
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. *winspecific* removes all menu bar items that are window specific such as mappers, browsers, layouts, etc.

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)

Note: 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 the **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; 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 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** or **Alter Menu**)

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” (i.e. if the **Create Menu** 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.

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.

<i>obj_info_code</i> value	Result of Alter Object
OBJ_INFO_PEN	Resets object's Pen style; <i>new_info_value</i> must be a Pen expression
OBJ_INFO_BRUSH	Resets object's Brush style; <i>new_info_value</i> must be a Brush expression
OBJ_INFO_TEXTFONT	Resets a Text object's Font style; <i>new_info_value</i> must be a Font expression
OBJ_INFO_SYMBOL	Resets a Point object's Symbol style; <i>new_info_value</i> must be a Symbol expression
OBJ_INFO_SMOOTH	Resets a Polyline object's smoothed/unsmoothed setting; <i>new_info_value</i> must be a Logical expression
OBJ_INFO_FRAMEWIN	Changes which window is displayed in a Layout frame; <i>new_info_value</i> must be an Integer window ID
OBJ_INFO_FRAMETITLE	Changes the title of a Frame object; <i>new_info_value</i> must be a String
OBJ_INFO_TEXTSTRING	Changes the text string that comprises a Text object; <i>new_info_value</i> must be a String expression
OBJ_INFO_TEXTSPACING	Changes a Text object's line spacing; <i>new_info_value</i> must be a Float value of 1, 1.5, or 2
OBJ_INFO_TEXTJUSTIFY	Changes a Text object's alignment; <i>new_info_value</i> must be 0 for left-justified, 1 for center-justified, or 2 for right-justified
OBJ_INFO_TEXTARROW	Changes a Text object's label line setting; <i>new_info_value</i> 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
```

Note: 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.

<i>attribute setting</i>	Result of Alter Object
OBJ_GEO_MINX	alters object's minimum bounding rectangle
OBJ_GEO_MINY	alters object's MBR
OBJ_GEO_MAXX	alters object's MBR; does not apply to Point objects
OBJ_GEO_MAXY	alters object's MBR; does not apply to Point objects
OBJ_GEO_ARCBEGANGLE	alters beginning angle of an Arc object
OBJ_GEO_ARCENDANGLE	alters ending angle of an Arc object
OBJ_GEO_LINEBEGX	alters a Line object's starting node
OBJ_GEO_LINEBEGY	alters a Line object's starting node
OBJ_GEO_LINEENDX	alters a Line object's ending node
OBJ_GEO_LINEENDY	alters a Line object's ending node
OBJ_GEO_POINTX	alters a Point object's x coordinate
OBJ_GEO_POINTY	alters a Point object's y coordinate
OBJ_GEO_ROUNDRAIUS	alters the diameter of the circle that defines the rounded corner of a Rounded Rectangle object
OBJ_GEO_TEXTLINEX	alters x coordinate of the end of a Text object's label line
OBJ_GEO_TEXTLINEY	alters y coordinate of the end of a Text object's label line
OBJ_GEO_TEXTANGLE	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** and **Create Region** statements. **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** or **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 (i.e., 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(), CentroidX(), and CentroidY() functions.

OBJ_GEO_CENTROID is defined in mapbasic.def.

Multipoint Objects and Collections

The **Alter Object** statement has been extended to support the following new 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
```

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

Support has also been added to the Alter Object statement that allows you to insert and delete 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)
```

Note: 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 - Multipoint object variable

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

node_num - 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 - Multipoint object variable

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

node_num - number of a node to be changed

(*x,y*) - new coordinates of 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
```

Note: 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.

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.

Note: 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**.

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.

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](#)

Area() function

Purpose

Returns the geographical area of an Object.

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 the **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.

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.

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.

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.

Note: 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.

Syntax

```
Asin ( num_expr )
```

num_expr is a numeric expression from one to minus 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 minus one, the expression *num_expr* should represent a value no larger than one and no smaller than minus one.

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, asking the user a yes or no (OK or Cancel) question.

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 *prompt* string can be up to 300 characters long.

The dialog 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 (e.g., by pressing the Escape 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 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.

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.

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** clause 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.

Syntax

`Beep`

Description

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

Browse statement

Purpose

Opens a new Browser window.

Syntax

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

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

table is the name of an open table

unitname is a String representing the name of a paper unit (e.g., "mm")

x , *y* specifies the position of the upper left corner of the Browser, in paper units

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 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. Conversely, if the **Browse** statement includes the **Min** keyword, the Browser window is minimized immediately; note that certain hardware platforms do not support minimized windows.

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](#)

Brush clause

Purpose

Specifies a fill style for graphic objects.

Syntax

```
Brush brush_expr
```

brush_expr is a Brush expression, such as `MakeBrush(pattern, fgcolor, bgcolor)`

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**, 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**), 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.

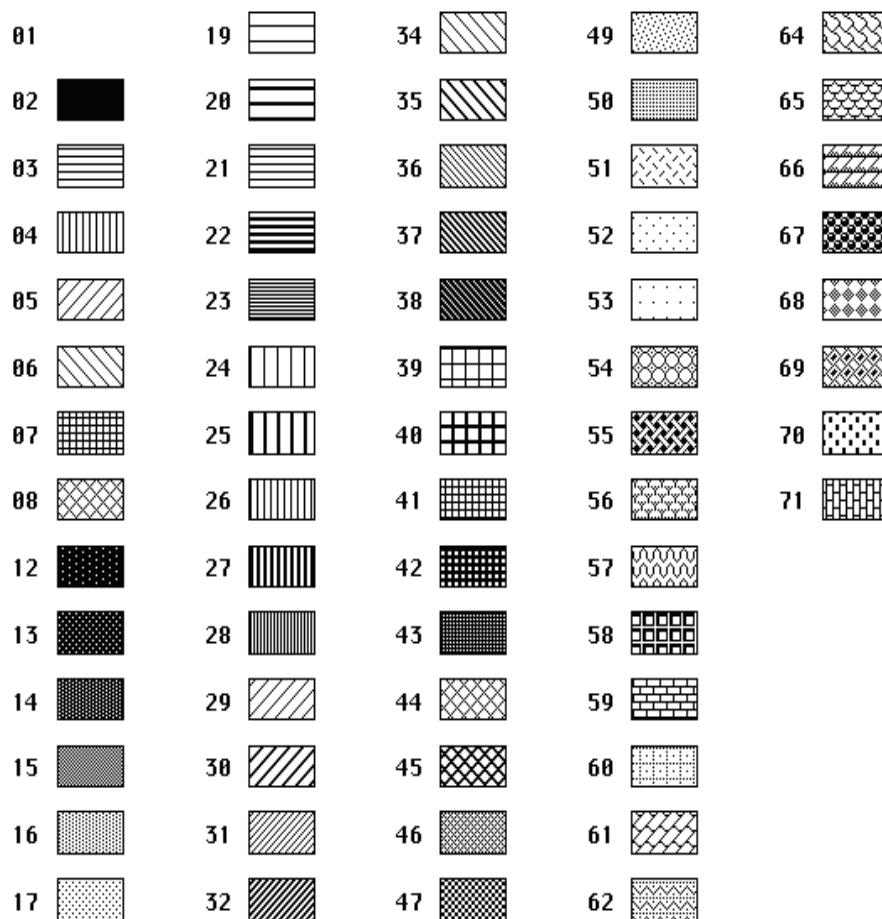
The following table summarizes the three components (pattern, foreground color, background color) that define a Brush:

Component	Description
pattern	Integer value from 1 to 8 or from 12 to 71; see table below.
foreground color	Integer RGB color value; see the 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.

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

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

**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).

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 As Buffer** statement. 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.

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

<i>attribute code</i>	ButtonPadInfo() returns:
BTNPAD_INFO_FLOATING	Logical: TRUE means the pad is floating, FALSE means the pad is docked.
BTNPAD_INFO_NBTNS	Smallint: The number of buttons on the pad.
BTNPAD_INFO_WIDTH	Smallint: The width of the pad, expressed as a number of buttons (not including separators).
BTNPAD_INFO_WINID	Integer: The window ID of the specified pad.
BTNPAD_INFO_X	A number indicating the x-position of the upper-left corner of the pad. If pad is docked, this is an Integer, zero or greater; if pad is floating, this is a Float value, in paper units such as inches.
BTNPAD_INFO_Y	A number indicating the y-position of the upper-left corner of the pad.

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** statement). Alternately, a program running under MapInfo Professional for Windows 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** 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.

Syntax

```
CartesianArea( expr, unit_name )
```

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 SQL **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).

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 As Buffer** statement. 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.

The **CartesianBuffer()** function will calculate 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.

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.

Returns

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.

`CartesianClosestPoints()` 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.

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 the **Set Distance Units** statement for the list of available unit names.

The **CartesianDistance()** 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 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.

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.

Syntax

```
CartesianObjectDistance(object1, object2, unit_name)
```

object1 and *object2* are object expressions.

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

Returns

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.

Syntax

```
CartesianObjectLen( 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.

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, and

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 or Perimeter, 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.

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, and

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 or Perimeter, 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.

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.

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. 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 SQL **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.

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.

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 SQL **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()** and **CentroidY()** functions, 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.

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.

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 **Open File**, 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 (i.e. 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)
"PackedEUC.Japanese"	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"	
"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

Character Set	Comments
"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

Note: 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...As** statement.

MapBasic 2.x CharSet Syntax

MapBasic version 2.x supported three character sets: "XASCII", "ANSI" and "MAC". Older programs that refer to those three character-set names will still compile and run in later versions of MapBasic; however, continued use of the 2.x-era character set names is discouraged.

CharSet "XASCII" specifies the same character set as **CharSet "CodePage437"**.

CharSet "MAC" specifies the same character set as **CharSet "MacRoman"**.

When a program runs on Windows, **CharSet "ANSI"** specifies whatever character set Windows is currently using. Example: When reading a file created by a DOS application, you should specify the "CodePage437" character set, as shown in the following example.

```
Open File "parcel.txt"
  For INPUT As #1
    CharSet "CodePage437"
```

See Also

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

ChooseProjection\$() function

Purpose

Displays the Choose Projection dialog and returns the coordinate system selected by the user.

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 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 is displayed. If false, then the dialog is not displayed and the default boundary is used.

Description

This function displays the Choose Projection dialog 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.

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.

Note: 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 error 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.

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 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.

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

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 **Close All**.

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 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** statement before the **Close Table** statement. To determine whether a table has unsaved edits, call the **TableInfo**(*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.

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()** and **WindowID()** functions.

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).

The following table lists the available *window_spec* values:

Window name	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
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.

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.

<i>attribute</i> setting	ColumnInfo() returns:
COL_INFO_NAME	String identifying the column name
COL_INFO_NUM	SmallInt indicating the number of the column
COL_INFO_TYPE	SmallInt indicating the column type (see table below)
COL_INFO_WIDTH	SmallInt indicating the column width; applies to Character or Decimal columns only
COL_INFO_DECPLACES	SmallInt indicating the number of decimal places in a Decimal column
COL_INFO_INDEXED	Logical value indicating if column is indexed
COL_INFO_EDITABLE	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:	Type of column indicated:
COL_TYPE_CHAR	Character
COL_TYPE_DECIMAL	Fixed-point decimal
COL_TYPE_FLOAT	Floating-point decimal
COL_TYPE_INTEGER	Integer (4-byte)
COL_TYPE_SMALLINT	Small Integer (2-byte)
COL_TYPE_DATE	Date
COL_TYPE_LOGICAL	Logical (TRUE or FALSE)
COL_TYPE_GRAPHIC	special column type Obj; this represents the graphical objects attached to the table

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 error generated if the specified table is not available

ERR_FCN_ARG_RANGE 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

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()** MapBasic 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, Polyline, or Arcs) and produce Polyline 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.

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:

<i>attribute</i> code	CommandInfo(<i>attribute</i>) returns:
CMD_INFO_DLG_OK	Logical value: TRUE if the user dismissed a custom dialog box by clicking OK; FALSE if user canceled by clicking Cancel, pressing Esc, etc. (This call is only valid following a Dialog statement.)
CMD_INFO_STATUS	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:

<i>attribute</i> code	CommandInfo(<i>attribute</i>) returns:
CMD_INFO_MENUITEM	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	Logical value: TRUE if the user double-clicked on a ListBox or MultiListBox control within a custom dialog. 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**, **WinChangedHandler** and **WinClosedHandler**. From within **SelChangedHandler**:

<i>attribute</i> code	CommandInfo(attribute) returns:
CMD_INFO_SELTYPE	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	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	Logical value: TRUE if the user interrupted a selection by pressing Esc, FALSE otherwise.

From within **RemoteMsgHandler**, **RemoteQueryHandler()**, or **RemoteMapGenHandler**:

CMD_INFO_MSG	String value, representing the execute string or the item name sent to MapInfo Professional by a client program. For details, see RemoteMsgHandler , RemoteQueryHandler() , or RemoteMapGenHandler .
--------------	--

From within **WinChangedHandler** or **WinClosedHandler**:

CMD_INFO_WIN	Integer value, representing the ID of the window that changed or the window that closed. For details, see WinChangedHandler or WinClosedHandler .
--------------	---

From within **ForegroundTaskSwitchHandler**:

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:

<i>attribute</i> code	CommandInfo(<i>attribute</i>) returns:
CMD_INFO_FIND_RC	Integer value, indicating whether the Find statement found a match.
CMD_INFO_FIND_ROWID	Integer value, indicating the Row ID number of the row that was found.
CMD_INFO_X or CMD_INFO_Y	Floating-point number, indicating x- or y-coordinates of the location that was found.

Within a Custom ToolButton's Handler Procedure

Within a custom ToolButton's handler procedure, you can specify any of these codes:

<i>attribute</i> code	CommandInfo(<i>attribute</i>) returns:
CMD_INFO_X	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 leftmost 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.
CMD_INFO_Y	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	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	y-coordinate of the spot where the user released the mouse button.
CMD_INFO_SHIFT	Logical value: TRUE if the user held down the Shift key while clicking.
CMD_INFO_CTRL	Logical value: TRUE if the user held down the Ctrl key while clicking.

<i>attribute code</i>	CommandInfo(<i>attribute</i>) returns:
CMD_INFO_TOOLBTN	Integer value, representing the ID of the button the user clicked.
CMD_INFO_CUSTOM_O BJ	Object value: a polyline or polygon drawn by the user. Applies to drawing modes DM_CUSTOM_POLYLINE or DM_CUSTOM_POLYGON.

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:

<i>attribute code</i>	CommandInfo(<i>attribute</i>) returns:
CMD_INFO_HL_WINDOW_ID	Id of map or browser window.
CMD_INFO_HL_TABLE_NAME	Name of table associated with the map layer or browser
CMD_INFO_HL_ROWID	Id of the table row corresponding to the map object or browser row.
CMD_INFO_HL_LAYER_ID	Layer id, if the program was launched from a map window.
CMD_INFO_HL_FILE_NAME	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.

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
  [ CoordSys... ]
  [ Version version ] ]
[ { Interactive | Automatic commit_keyword } ]
  [ ConvertObjects { ON | OFF | INTERACTIVE } ]
```

table is the name of the table you are saving.

filespec is a file specification (optionally including directory path). This is where the MapInfo .TAB file is saved.

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 4.0. If the database in which the table is being created already exists, the specified database version is ignored

char_set is the name of a character set; see the separate **CharSet** discussion.

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 file.

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.

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.

tablename is the name of the table as you want it to appear in the database.

CoordSys is a coordinate system clause; see the separate **CoordSys** discussion.

version is 100 (to create a table that can be read by versions of MapInfo Professional) or 300 (MapInfo Professional3.0 format) for non-Access tables. For Access tables, version is 410.

commit_keyword is one of the following keywords: **NoCollision**, **ApplyUpdates**, **DiscardUpdates**

Description

If no **As** clause is specified, the **Commit** statement saves any pending edits to the table. This is analogous to the user choosing File > Save.

A **Commit** 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. By default, the type of the new table is **NATIVE**, but can also be saved as DBF.

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 the discussion of the **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 statements in MapBasic can be saved. The **Commit Table** statement will create 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.

Note: 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. After a successful Commit Table Interactive statement, MapInfo Professional displays a dialog allowing the user to refresh.

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. Unsupported object types will not be saved to the destination table, but the rest of the attributes will be saved. The supported databases include Oracle, SQL Server, IIS (Informix Universal Server), and Microsoft Access. However, to save a table with a spatial geometry/object, (including saving a point-only table) the SpatialWare/Blade is required for SQL Server and IUS, 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** 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 hodb as integer
hodb = server_connect("ODBC", "dlg=1")
Open table "C:\MapInfo\USA"
Commit Table USA
as "c:\temp\as\USA"
Type ODBC Connection hodb Table "USA"
```

See Also

Rollback statement

ConnectObjects() function

Purpose

Returns an object representing the shortest or longest distance between two objects.

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.

Returns

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**).

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** statements 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.

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 units

w specifies the width of the button in dialog units; default width is 40

h specifies the height of the button in dialog units; default height is 18

control_ID is an Integer; cannot be the same as the ID of another control in the dialog

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 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. Similarly, the user chooses a **CancelButton** control to “choose Cancel” and dismiss the dialog. Each dialog 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, whether the control is 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.

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 units

w specifies the width of the control in dialog units

control_ID is an Integer; cannot be the same as the ID of another control in the dialog

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 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 **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 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 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

Disable grays out the control initially

Hide initially hides the control

Description

If a Dialog statement includes a Control DocumentWindow clause, the dialog includes a document window control that can be re-parented using Set Next Document.

Example

The following example draws a legend in a dialog:

```
Control DocumentWindow
  ID ID_LEGENDWINDOW
  Position 160, 20
  Width 120 Height 150
```

The dialog 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.

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 units.

w specifies the width of the control in dialog units.

h specifies the height of the control in dialog 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.

initial_value is a String or a numeric expression that appears in the box initially.

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 *str_value* expression contains embedded **Chr\$(10)** values, the text appears formatted when the dialog 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.

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 units

w specifies the width of the control in dialog units

h specifies the height of the control in dialog units

control_ID is an Integer; cannot be the same as the ID of another control in the dialog

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.

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 units

w specifies the width of the control in dialog units; default width is 80

h specifies the height of the control in dialog units; default height is 70

control_ID is an Integer; cannot be the same as the ID of another control in the dialog

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 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 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 (for example, assign a handler to the OKButton control). Within the handler procedure, set up a loop to call **ReadControlValue()** repeatedly.

The first call to **ReadControlValue()** returns the number of the first selected item; the second call to **ReadControlValue()** returns the number of the second selected item; etc. When **ReadControlValue()** returns zero, you have exhausted the list of selected items. If the first call to **ReadControlValue()** 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 as if the user had clicked on a list item and then clicked OK.

To see an example, click here: [Letting the user double-click](#)

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

```

See Also

Alter Control statement, Dialog statement, ReadControlValue() 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 units

w specifies the control's width, in dialog units; default width is 20

h specifies the control's height, in dialog units; default height is 20

control_ID is an Integer; cannot be the same as the ID of another control in the dialog

handler is the name of a handler procedure; if the user clicks on the Picker control, and then clicks OK on the style dialog 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 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

Control PopupMenu clause
Purpose

Part of a **Dialog** statement; adds a popup menu control to the dialog.

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 units

w specifies the control's width, in dialog units; default width is 80

control_ID is an Integer; cannot be the same as the ID of another control in the dialog

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 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 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.

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.

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 units

control_ID is an Integer; cannot be the same as the ID of another control in the dialog

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 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 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. 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.

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 units

w specifies the control's width, in dialog units

h specifies the control's height, in dialog units

control_ID is an Integer; cannot be the same as the ID of another control in the dialog

title_string is a text string to appear in the dialog 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.

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.

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.

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.

Syntax 1

```
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

```
CoordSys Nonearth
[ Affine Units unitname, A, B C, D, E, F ]
Units unitname
Bounds ( minx, miny) ( maxx, maxy)
```

Syntax 3

```
CoordSys Layout Units paperunitname
```

Syntax 4

```
CoordSys Table tablename
```

Syntax 5

```
CoordSys Window window_id
```

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

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

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

tablename is the name of an open table

window_id is an Integer window identifier corresponding to a Map or Layout window

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 syntax 1 (above) 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 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.

Use syntax 3 (**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. The following **Set CoordSys** statement assigns a Layout window's coordinate system, using inches as the unit of measure:

```
Set CoordSys Layout Units "in"
```

Use syntax 4 (**CoordSys Table**) to refer to the coordinate system in which a table has been saved.

Use syntax 5 (**CoordSys Window**) to refer to the coordinate system already in use in a window.

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.

Versions of MapInfo Professional prior to MapInfo Professional 4.1.2 do not recognize the affine transformation constants in the **CoordSys** clause, Mapinfo.prj, or any MAP file. If a MAP file is created using an affine transformation, older versions of MapInfo Professional will use the base coordinate system instead of the derived coordinate system.

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.

Note: 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, Units, and Origin Latitude parameters, while the formula for a Transverse Mercator projection uses the Datum, Units, 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** 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
Commit Table world As "RWORLD.TAB"
CoordSys Earth Projection 12, 12, "m", 0.
```

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
```

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
```

See Also

Commit Table statement, Set CoordSys statement, Set Map statement

Cos() function

Purpose

Returns the cosine of a number.

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()** will be between one and minus 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. (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 = 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 Arc statement

Purpose

Creates an arc object.

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, see 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; see the **Pen** discussion for more details. 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).

See Also

Insert statement, Pen clause, Update statement

Create ButtonPad statement

Purpose

Creates a ButtonPad (toolbar).

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 } ]
```

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 ODBC

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 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)

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.

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. 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** and **Alter ButtonPad** statements.

Each toolbar can be hidden. To create a toolbar in the hidden state, include the **Hide** keyword. Each toolbar can be floating or fixed to the top of the screen ("docked"). A floating toolbar resembles a window, such as the Info tool window. To create a fixed toolbar, include the keyword **Fixed**. To create a floating toolbar, include the keyword **Float**. 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**.

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 <i>MapBasic User Guide</i> .
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*.

Example

```
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
```

See Also

[Alter Button statement](#), [Alter ButtonPad statement](#)

Create ButtonPads As Default statement

Purpose

Restore standard ButtonPads (for example, the Main ButtonPad) to their default state.

Syntax

```
Create ButtonPads As Default
```

Description

This statement destroys any custom ButtonPads and returns MapInfo Professional's standard ButtonPads (Main, Drawing, and Tools) to their default states.

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 Cartographic Legend statement

Purpose

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.

The previous MapInfo Professional map legend was a single floating window that only displayed thematic legends for the active map window and was shared by all map windows. The new legend window will replace the current legend window; however, the current legend window and its functionality will still be available programmatically through existing MapBasic statements (i.e., **Create Legend**, **Set Legend**, etc....)

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()** and **WindowId()** functions.

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.

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 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, or the object column (meaning that legend styles are based on the unique styles in the mapfile). The default is 'object'.

label is either 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.

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

Description

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 **Frame**.

Behind 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 work space 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 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 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** clause and appropriate **Line**, **Region**, or **Symbol** clause. Multipoint objects are treated as Point objects.

Collection objects are treated separately. When we create Legend based on object types, we draw Point symbols first, then Lines, then Regions. Collection objects are drawn last. Inside collection objects we draw point, then line and then region samples.

If a **Column** is defined, it must be 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'.

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** (i.e., 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 then 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, it is either 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 will be searched for metadata values for Title, Subtitle, Column and Label. If no metadata value exists for 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 will create 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.

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.

Note: 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 will use whatever Brush style is currently selected. To create a circle object with a specific Brush, you could issue the **Set Style** statement before calling **CreateCircle()**. Alternately, instead of calling **CreateCircle()**, you could issue a **Create Ellipse** statement, which has optional Pen and Brush clauses.

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 (through an **Insert** statement).

Note: Before creating objects on a Layout window, you must issue a **Set CoordSys Layout** statement.

Error Conditions

ERR_FCN_ARG_RANGE error 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 will be 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, 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

Combine points, linear objects and closed objects into a single object. The collection object displays in the Browser as a single record.

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 - 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.

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

Given a set of Target objects, and a set of polylines as a selection object, this statement will produce 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.

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 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 will return 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.

Note: 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.

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

The **Pen** clause specifies a line style

The **Brush** clause specifies a 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 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 lat/long 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, see 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; see the **Pen** discussion for more details. 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). Similarly, the optional **Brush** clause specifies a fill style; see the **Brush** discussion for more details.

See Also

Brush clause, CreateCircle() function, Insert statement, Pen clause, Update statement

Create Frame statement

Purpose

Creates a new frame in a Layout window.

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

The **Pen** clause specifies a line style

The **Brush** clause specifies a 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

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 or zed 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 won't be 200 by 200, but it will be at least that wide and high.

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 at By Percent [
    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 coordsys as the source table. Refer to the **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:value inflection 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:value inflection 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 name:value pair.

parameter_value is a numeric expression, specifying the value part of a name:value pair.

By Percent is a string expression, specifying the name part of a name:value pair.

Round is a numeric expression, specifying the value part of a name:value pair.

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.

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.

Note: MapInfo Professional cannot create an index if the table has unsaved edits. Use the **Commit** 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](#)

Create Legend statement

Purpose

Creates a new theme legend window tied to the specified Map window.

For 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*, Chapter 11.

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 **Set Window**).

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

CreateLine() function

Purpose

Returns an Object value representing a line.

Syntax

```
CreateLine( x1 , y1, x2 , y2 )
```

x1 is a Float value, indicating the x-position (for example, Longitude) 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.

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, see the **Set Paper Units** statement.

Note: 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 the **Pen** discussion 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.

Syntax

```
Create Map
  For table
    [ CoordSys... ] Using from_table]
```

table is the name of an open table

CoordSys... is a CoordSys clause

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 **Commit 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 the 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 will be 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 the separate discussion of the **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.

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 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 ranges from 0-360 degrees and rotates the maps around the center point of the grid. The vertical angle in the dialog 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, ViewPlane Normal and Clipping Range (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.

`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.

Units specifies the units the grid values are in. Do not specify this for unitless grids (i.e., grids generated using temperature or density). This option needs to be specified at creation time. You cannot change them later with Set Map3D or the Properties dialog.

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 will be displayed on the grid.

The creation will fail 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.

Syntax 1

```
Create Menu newmenuname [ ID menu_id ] 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

`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*, below. 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 the discussion of the **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.

Note: 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; a 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 **Alter Menu Item**. 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 Deletions"
! ... ^ ...	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.

Note: The codes in the following tables must appear at the end of a menu item name.

Windows Accelerator Code	Effect
<i>/W {letter %number}</i>	Defines a Windows shortcut key which can be activated by pressing the appropriate key. Examples: "Zap /WZ" or "Zap /W%120"
<i>/W# {letter %number}</i>	Defines a Windows shortcut key which also requires the shift key. Examples: "Zap /W#Z" or "Zap /W#%120"
<i>/W@ {letter %number}</i>	Defines a Windows shortcut key which also requires the Alt key. Examples: "Zap /W@Z" or "Zap /W@%120"
<i>/W^ {letter %number}</i>	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.

Note: 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"	Windows only. 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). The remainder of the string contains a comma-separated list of the values returned from relevant **CommandInfo()** 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, 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 function Chr\$(9) 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.

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 the **Alter Menu** 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

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 - number of points inside Multipoint object.

x y specifies the location of the point

The **Symbol** clause specifies a symbol style.

Note: 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 1 (MapInfo 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 the RGB() function.

size is an Integer point size, from 1 to 48.

Syntax 2 (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 the 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 3 (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 the 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 4

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

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 or Voronoi operation.

Syntax

```
Create Object As { Buffer | Union | Intersect | Merge | ConvexHull | Voronoi }
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 } ]
```

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

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 is the name of a distance unit (for example, "km" for kilometers)

smoothness is an Integer from 2 to 100, indicating the number of segments per circle in a Buffer operation

column is the name of a column in the table

Description

The **Create Object** statement creates one or more new region objects, by performing a geographic operation (**Buffer**, **Merge**, **Intersect**, **Union**, **ConvexHull** or **Voronoi**) 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.

Note: If you specify a **Group By** clause to perform data aggregation, you must store the results to a table rather than a variable.

The keyword which follows the **As** keyword dictates what type of objects will be created. Specify **Buffer** to generate buffer regions; see below for details. 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).

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.

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. Objects Combine will delete the input and insert a new combined object. Create Objects As Union will only insert the new combined object, it will not delete the input objects. Combining using a Target and potentially different tables is only available with Objects Combine. 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.

Use **Type** is the method used to calculate the buffer width around the object. It can either be **Spherical** or **Cartesian**. Note that if the Coordsys of the intotable is NonEarth, then the calculations will be performed using **Cartesian** methods regardless of the option chosen, and if the Coordsys of the intotable is Latitude/Longitude, then calculations will be performed using **Spherical** methods regardless of the option chosen.

Convex Hull Geographic Operation for the Create Object statement

```
Create Object As { Buffer | Union | Intersect | Merge | ConvexHull }
```

The **Create Object** statement creates one or more new region objects, by performing a geographic operation (**Buffer**, **Merge**, **Intersect**, **Union**, or **ConvexHull**) on one or more existing objects.

The **ConvexHull** operator will create 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 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.

The points used to construct the convex hull will be any nodes from Regions, Polylines, or Points in the From table. 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.

Buffering

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 will be interpreted in MapBasic’s current distance unit. By default, MapBasic uses miles as the distance unit; to change this unit, see the **Set Distance Units** statement.

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 will be 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 will be used regardless of the **Type** setting. If the data is in a NonEarth Projection, the Cartesian calculations will be used regardless of the **Type** setting.

The *smoothness* parameter 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, meaning that there will be 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.

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 will group the Parcel regions, then perform one merge operation for each group. Thus, the Zones table will end up with one region object for each group of objects in the Parcels table. Each group will consist of all parcels having the same value in the zone_id column.

Following the **Create Object** statement, the parcelcount column in the Zones table will indicate how many parcels were merged to produce that zone. The zonevalue column in the Zones table will indicate 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 creates a region object, representing a quarter-mile buffer around whatever objects are currently selected. The buffer object will be stored in the Object variable, corridor. A subsequent **Update** 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
```

The next example shows a multi-object convex hull using the Create Object As statement.

```
create object as convex hull from state_caps into table dump_table
```

See Also

Buffer() function, ConvexHull() function, Objects Combine statement, Objects Erase statement, Objects Intersect statement

Create Pline statement

Purpose

Creates a polyline object.

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 it will not be known 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 **Alter Object** to add detail to the polyline object. See the discussion of the **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 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; see **Set CoordSys** for more information). 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, see 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 the **Pen** discussion for more details. 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). **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**, **Update statement**

CreatePoint() function

Purpose

Returns an Object value representing a point.

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).

Note: If you need to create objects on a Layout window, you must first issue a **Set CoordSys Layout** 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. Note that MapInfo for DOS pointfiles store coordinates in *millionths* of degrees, not whole degrees. Also, most MapInfo for DOS pointfiles store longitude coordinates in the “NorthWest quadrant,” meaning that longitudes increase as you move westward. Thus, to perform the **Update** operation on a MapInfo for DOS pointfile, you would need to divide the Xcoord and Ycoord fields by one million, and multiply the Xcoord field by negative one:

```
Update sites
Set obj = CreatePoint (-xcoord/1000000,ycoord/1000000)
```

See Also

Create Point statement, Insert statement, Update statement

Create Point statement

Purpose

Creates a point object.

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, see the **Set Paper Units** statement.

Note: 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 the **Symbol** discussion 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).

See Also

CreatePoint() function, **Insert statement**, **Symbol clause**, **Update statement**

Create PrismMap statement

Purpose

Creates a Prism map.

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 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.

Camera specifies the camera position and orientation.

angle is an angle measurement in degrees. The horizontal angle in the dialog ranges from 0-360 degrees and rotates the maps around the center point of the grid. The vertical angle in the dialog 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, ViewPlane Normal and Clipping Range (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.

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").

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.

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.

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 **Quantile Using** expression.

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** 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.

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

The **Pen** clause specifies a line style

The **Brush** clause specifies a fill style

Description

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, see the **Set Paper Units** statement.

Note: 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 the **Pen** discussion 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). Similarly, the optional **Brush** clause specifies a fill style; see the **Brush** discussion for more details.

See Also

Brush clause, Create RoundRect statement, Insert statement, Pen clause, Update statement

Create Redistrict statement

Purpose

Begins a redistricting session.

Syntax

```
Create Redistrict source_table By district_column
  With
    [ Count ]
    [ , Brush ] [ , Symbol ] [ , Pen ]
    [ , { Sum | Percent } ( expr ) ]
    [ , { Sum | Percent } ( expr ) . . . ]
    [ 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

the **Count** keyword specifies that the Districts Browser will show a count of the objects belonging to each district

the **Brush** keyword specifies that the Districts Browser will show each district's fill style

the **Symbol** keyword specifies that the Districts Browser will show each district's symbol style

the **Pen** keyword specifies that the Districts Browser will show each district's line style

expr is a numeric column expression

the **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 complete **Symbol** or **Pen** clauses. 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.

See Also

[Set Redistricter statement](#)

Create Region statement

Purpose

Creates a region object.

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

The **Pen** clause specifies a line style

The **Brush** clause specifies a 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 **Alter Object** 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 **Alter Object** to add the variable numbers of nodes. See **Alter Object** 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, see the **Set Paper Units** statement.

Note: 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 the **Pen** discussion 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). Similarly, the optional **Brush** clause specifies a fill style; see the **Brush** discussion 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:

Syntax

```
Create Report From Table tablename [Into reportfilespec] [Interactive]
```

tablename is an open table in MapInfo

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.

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

The **Pen** clause specifies a line style

The **Brush** clause specifies a 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, see the **Set Paper Units** statement.

Note: 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 object's outline; see the **Pen** discussion 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). Similarly, the optional **Brush** clause specifies a fill style; see the **Brush** discussion 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.

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

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 the discussion of the **Create Ranges** statement.

See Also

[Create Ranges statement](#), [Set Shade statement](#), [Shade statement](#)

Create Table statement

Purpose

Creates a new table.

Syntax

```
Create Table table
( column columntype [ , . . . ] ) | Using from_table {
  [ File filespec ]
  [ { Type NATIVE |
    Type DBF [ CharSet char_set ] |
    Type {Access | ODBC} database_filespec [ Version version ]
    Table tablename
    [ Password pwd ] [ CharSet char_set ]
  } ]
  [ Version version ]
```

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.

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.

char_set is the name of a character set; see the separate **CharSet** discussion.

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 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 4.0. 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.

version is 100 (to create a table that can be read by versions of MapInfo Professional) or 300 (MapInfo Professional 3.0 format). Does not apply when creating an Access table; the version of the Access table is handled by DAO.

columnntype is the data type associated with the column. Each *columnntype* is defined as follows:

```
Char( width ) |  
Float |  
Integer |  
SmallInt |  
Decimal( width , decplaces ) |  
Date |  
Logical
```

width indicates how large each field should be (does not apply to all field types). Char fields can have a width of up to 254 characters.

decplaces indicates the number of decimal places to use in a Decimal field.

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 can't be used and will produce an error. The column structure of the new table being created will be identical to this table.

The optional *filespec* clause specifies where to create the new table. If no *filespec* 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. See the **CharSet** clause discussion for more information.

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 **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.

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
```

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.

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 50, 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.

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 ) ]
[ 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

text_angle is a Float value indicating the angle of rotation for the text object (in degrees)

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.

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\$() or Srt\$() functions.

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](#)

CurrentBorderPen() function

Purpose

Returns the current border pen style currently in use.

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, 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).

To extract specific attributes of the Pen style (such as the color), call the **StyleAttr()** function. For more information about Pen settings, see the 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.

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. 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 **Create Region**, 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** or **Shade**).

To extract specific Brush attributes (such as the color), call **StyleAttr()**.

For more information about Brush settings, see the **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.

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 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**).

To extract specific attributes of the Font style (such as the color), call the **StyleAttr()** function.

For more information about Font settings, see the **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.

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, 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).

To extract specific attributes of the Pen style (such as the color), call the **StyleAttr()** function. For more information about Pen settings, see the 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.

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 **Create Line**, 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 **Set Map**).

To extract specific attributes of the Pen style (such as the color), call the **StyleAttr()** function.

For more information about Pen settings, see the **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.

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. 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 setting as a parameter (such as **Set Map** or **Shade**).

To extract specific attributes of the Symbol style (such as the color), call the **StyleAttr()** function.

For more information about Symbol settings, see the **Symbol** clause.

Example

```
Dim sym_user_symbol As Symbol
sym_user_symbol = CurrentSymbol( )
```

See Also

MakeSymbol() function, **Set Style statement**, **StyleAttr() function**, **Symbol clause**

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.

Syntax

```
DateWindow(context)
```

context is a SmallInt that can either be DATE_WIN_CURPROG or DATE_WIN_SESSION.

Description

This depends on which context is passed. If *context* is DATE_WIN_SESSION, then the current session setting in effect is returned. If *context* is DATE_WIN_CURPROG, then the current MapBasic program's local setting is returned, if a program is not running the session setting is returned.

MBX's compiled before v5.5 will still convert 2-digit years to the current century (5.0 and earlier behavior). To get the new behavior, they must be recompiled with MapBasic v5.5 or later.

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.

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 error generated if not running on Windows

ERR_NO_RESPONSE_FROM_APP 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** statements (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** or **DDETerminateAll** statements.

Note: 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 (i.e. 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\$() calls can return Excel system information, such as a list of the names of the worksheets in use; DDE-Execute statements can send commands for Excel to execute
DDEInitiate("Excel" , <i>wks</i>)	If <i>wks</i> is the name of an Excel document in use, subsequent DDEPoke statements can store values in the worksheet, and DDERequest\$() calls can read information from the worksheet
DDEInitiate("MapInfo" , "System")	DDERequest\$() calls can provide system information, such as a list of the MapBasic applications currently in use by MapInfo Professional.
DDEInitiate("MapInfo" , <i>mbx</i>)	If <i>mbx</i> is the name of a MapBasic application in use, DDE-Poke statements can assign values to global variables in the specified application, and DDERequest\$() 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 sub procedure named RemoteMsgHandler).

Error Conditions

ERR_CMD_NOT_SUPPORTED error generated if not running on Windows

ERR_INVALID_CHANNEL 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

```

Note: 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 **DDEInitiate()**

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 error generated if not running on Windows

ERR_INVALID_CHANNEL 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 **DDEInitiate()**

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

Note: 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.

<i>Topic name</i>	<i>item names to use with DDERequest</i>
"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
<i>mbx</i> (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 error generated if not running on Windows

ERR_INVALID_CHANNEL error if the specified channel number is invalid

ERR_CANT_INITIATE_LINK 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 **DDEInitiate()**

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** or **DDETerminateAll** statements.

Note: 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 error generated if not running on Windows

ERR_INVALID_CHANNEL 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** or **DDETerminateAll** statements.

Error Conditions

ERR_CMD_NOT_SUPPORTED 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**.

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 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**.

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**.

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 token 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 the discussion of the **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.

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 thousands 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 **Val()** to determine the numeric value of a string. Before calling **Val()**, this example calls **DeformatNumber\$()** to remove thousands separators from the string. (The string that you pass to **Val()** cannot contain thousands 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.

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  
Commit 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  
Commit 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

h specifies the height of the dialog, in dialog units (8 dialog 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'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 appears centered

handler is the name of a procedure to call before the user is allowed to use the dialog; this procedure is typically used to issue **Alter Control statements**

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 the dialog. 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 dialogs, see the *MapBasic User Guide*.

Anything that can appear on a dialog is known as a *control*. Each dialog must contain at least one control (for example, an OKButton control). Individual control clauses are discussed in separate entries (for example, see **Control CheckBox** for a discussion of check-box controls). As a general rule, every dialog should include an OKButton control and/or a CancelButton control, so that the user has a way of dismissing the dialog.

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), use one of the following statements or functions: **Ask()**, **Note**, **ProgressBar**, **FileOpenDlg()**, **FileSaveAsDlg()**, or **GetSeamlessSheet()**.

For an introduction to the concepts behind MapBasic dialog boxes, see the *MapBasic User Guide*.

Sizes and Positions of Dialogs and Dialog Controls

Within the **Dialog** statement, sizes and positions are stated in terms of dialog units. A width of four dialog units equals the width of one character, and a height of eight dialog units equals the height of one character. Thus, if a dialog 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. To place a control at the upper-left corner of a dialog, 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, with subsequent control clauses appearing further down in the dialog.

Terminating a Dialog

After a MapBasic program issues a **Dialog** statement, the user will continue interacting with the dialog until one of four things happens:

- The user clicks the OKButton control (if the dialog has one);
- The user clicks the CancelButton control (if the dialog has one);
- The user clicks a control with a handler that issues a **Dialog Remove** statement; or
- The user otherwise dismisses the dialog (for example, by pressing Esc on a dialog that has a CancelButton).

To force a dialog 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 **CommandInfo()** to determine whether the user clicked OK or Cancel to dismiss the dialog. 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.

Note: MapInfo Professional only updates the variable if the user clicks OK. Also, MapInfo Professional only updates the variable after the dialog terminates.

To read a control's value from within a handler procedure, call **ReadControlValue()**.

Specifying Hotkeys for Controls

When a MapBasic application runs on MapInfo, dialogs can assign hotkeys to the various controls. A hotkey is a convenience allowing the user to choose a dialog 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 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. The focus moves from control to control according to the dialog'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 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 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 control.

You cannot issue this statement from the MapBasic window.

Description

The **Dialog Preserve** statement allows the user to resume using a custom dialog (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 asking a question such as “Do you want to lose your changes?” If the user chooses “No” on the confirmation dialog, the application should reactivate the original dialog. 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 control. You cannot issue this statement from the MapBasic window.

Description

The **Dialog Remove** statement removes the dialog created by the most recent **Dialog** statement. A dialog disappears automatically after the user clicks on an OKButton control or a CancelButton control. Use the **Dialog Remove** statement (within a dialog control's handler routine) to remove the dialog before the user clicks OK or Cancel. This is useful, for example, if you have a dialog with a ListBox control, and you want the dialog to come down if the user double-clicks an item in the list.

Note: Dialog Remove signals to remove the dialog after the handler sub procedure returns. It does not remove the dialog instantaneously.

Example

The following procedure is part of the sample program NIEWS.MB. It handles the ListBox control in the Named Views dialog. When the user single-clicks a list item, this handler procedure enables various buttons on the dialog. When the user double-clicks a list item, this handler uses a **Dialog Remove** statement to dismiss the dialog.

Note: 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.

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 -2,147,483,647 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 * <i>length</i>	Fixed-length character string (where <i>length</i> dictates the length of the string, in bytes, up to 32768 bytes); fixed-length strings are padded with trailing blanks
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
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** construct or within a **Function...End Function** construct, 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 (i.e. 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	Commit	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.

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 the **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 ... Else** 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 ... Else** 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** or **Exit Do**, 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.

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.)

Note: MapInfo Professional cannot drop an index if the table has unsaved edits. Use the **Commit** 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**.

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.

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.

Note: 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.

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.

Note: 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 the discussion of the 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

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.

Note: 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 NIEWS.MB (Named Views).

Error Conditions

ERR_FILEMGR_NOTOPEN 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.

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 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**

Erase() function**Purpose**

Returns an object created by erasing part of another object.

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.

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` 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()** and **ObjectInfo()**, take an Object expression as a parameter. Any such function will generate the `ERR_FCN_OBJ_FETCH_FAILED` 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` error. Since the `ERR_FCN_OBJ_FETCH_FAILED` 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` and `ERR_FP_MATH_LIB_RANGE` - which can occur as a result of an invalid numeric parameter. These errors might be generated by calls to any of the following functions: **Asin()**, **Acos()**, **Atn()**, **Cos()**, **Exp()**, **Log()**, **Sin()**, **Sqr()**, or **Tan()**.

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.

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.

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 the discussion of the **Err()** function for more information.

See Also

Err() function, **Error statement**, **OnError statement**

Exit Do statement

Purpose

Exits a **Do** loop 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** loop 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** loop. 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** construct.

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** definition.

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** procedure.

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 procedure calls 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.

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.

Note: 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.

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 filename 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, "WindowsLatin1"; see the separate **CharSet** discussion 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 "DBF"	dBASE file format. Note: Map objects are not exported when you specify DBF format.
Type "ASCII"	Text file format. Note: Map objects are not exported when you specify ASCII format.
Type "CSV"	Comma-delimited text file format. Note: 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:

Preserve AttributeData

Include this clause if you want to export the table's tabular data as attribute data in the DXF file.

Preserve MultiPolygonRgns As Blocks

Include this 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.

Binary or ASCII [DecimalPlaces *decimal_places*]

Include the **Binary** keyword to export into a binary DXF file; or, include the **ASCII** clause 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).

Version 12 or Version 13

This 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.

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 **ObjectInfo()**.

Error Conditions

ERR_FCN_ARG_RANGE 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.

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 optional parameter representing 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 represents a table of objects that you want to find farthest distances from.

fromvar represents a MapBasic variable representing an object that you want to find the farthest distances from.

totable represents a table of objects that you want to find farthest distances to.

intotable represents a table to place the results into.

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. The entire Ignore clause is optional, as are the Min and Max subclauses within it (e.g., only a Min or only a Max, or both may occur).

Normally, if one object is contained within another object, the distance between the objects is zero. For example, if the From table is WorldCaps and the To table 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.

The Data clause can be used to mark which *fromtable* object and which *totable* object the result came from.

Description

Every object in the *fromtable* is considered. For each object in the *fromtable*, the farthest object in the *totable* is found. If *N* is present, then the *N* farthest objects in *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 present, 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 (i.e., if *N* is greater than 1), then objects of the same distance will fill subsequent slots. If the tie exists at the second farthest object, and 3 objects are requested, then the object 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 (i.e., a distance of 100 should only occur in the first pass and never in the second pass).

Data Clause

```
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 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).

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
```

It is currently not possible to fill in a column in the *intotable* with the distance. However, this can be easily accomplished after the Nearest operation is completed by using the **TABLE > UPDATE COLUMN...** functionality from the menu or by using the Update MapBasic 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 (i.e., which row is the current row).

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.

Note: 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
<i>table.column</i>	World.Country
<i>table.col#</i>	World.col1
<i>table.col(number)</i>	World.col(variable_name)

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.

Note: 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, **EOT()** 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.coll + " has negative net migration"
```

See Also

EOT() function, **Open Table statement**

FileAttr() function

Purpose

Returns information about an open file.

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:

<i>attribute</i> parameter	Return Value
FILE_ATTR_MODE	Small Integer, indicating the mode in which the file was opened. Return value will be one of these: <ul style="list-style-type: none">• MODE_INPUT• MODE_OUTPUT• MODE_APPEND• MODE_RANDOM• MODE_BINARY
FILE_ATTR_FILESIZE	Integer, indicating the file size in bytes.

Error Conditions

ERR_FILEMGR_NOTOPEN error 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.

Syntax

```
FileExists( filespec )
```

filespec is a string that specifies the file path and name.

Return Value

Logical: TRUE if the file already exists

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, and returns the name of the file the user selected.

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, 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

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 similar to the one that displays when the user chooses File > Open.

To choose a file from the list that appears in the dialog, 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, and lets the user choose a filename. If your application then needs to actually open the file chosen by the user, the application must issue a statement such as **Open Table**. If you want your application to display an Open dialog, and then you want MapInfo Professional to automatically open the selected file, you can issue a statement such as **Run Menu Command M_FILE_OPEN** or **Run Menu Command 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 appears. If the *path* parameter is blank (a null string), the dialog will present a list of files in the current working directory.

The *filename* parameter specifies the default filename 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. Some *filetype* settings have special meaning; for example, if the *filetype* parameter is "TAB", the dialog will present a list of MapInfo tables, and if the *filetype* parameter is "WOR", the dialog will present a list of MapInfo workspace files.

There are also a variety of other three-character *filetype* values, summarized in the table below. If you specify one of the special *type* values from the table below, the dialog will include a control that lets the user choose between seeing a list of table files or a list of all files ("*. *").

<i>type</i> 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
"XLS"	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 DOS file extension; in other words, specifying a *filetype* parameter of "WOR" tells MapBasic to display a list of files having the DOS ".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 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()** call specifies a *filetype* parameter which is not listed in the table of file extensions above, the dialog would appear 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, and returns the name of the file the user entered.

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 should let the user choose

prompt is a String title that appears at the top of the dialog

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, similar to the dialog 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. Since each filename 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 appears. If the *path* parameter is blank (a null string), the dialog presents a list of files in the current directory.

The *filename* parameter specifies the default filename 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. To display a dialog that lists workspaces, specify the string "WOR" as the *filetype* parameter; to display a dialog that lists table names, specify the string "TAB." See the discussion of the **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, and lets the user choose a filename to save. To save data under the filename chosen by the user, issue a statement such as **Commit Table As**.

See Also

Commit Table statement, **FileOpenDlg() function**

Find statement

Purpose

Finds a location in a mappable table.

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 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. If the **Find** statement results in an approximate match, the function call returns a value greater than one. 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
xx1	Exact match
xx2	A substitution from the abbreviations file used
xx3 (-)	Exact match not found
xx4 (-)	No object name specified; match not found
xx5 (+)	The user chose a name from the Interactive dialog
x1x	Side of street undetermined
x2x (+ / -)	Address number was within min/max range
x3x (+ / -)	Address number was not within min/max range
x4x (+ / -)	Address number was not specified
x5x (-)	Streets do not intersect
x6x (-)	The row matched does not have a map object
x7x (+)	The user chose an address number from the Interactive dialog
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

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 the 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 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, and the user enters text in the dialog 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

Find Using statement

Purpose

Dictates which table(s) and column(s) should be searched in subsequent **Find** operations.

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. 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.

value specifies the Offset value (the distance back from the street). 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.

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.

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.

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.

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 **Create Text**, 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()** or **MakeFont()**) which returns a Font value:

```
Font MakeFont("Helvetica", 1, 12, BLACK, WHITE)
```

With some MapBasic statements (for example, **Set Legend**), 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
fontname	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 , <i>italic</i> , and <u>underline</u> . See table below for details.
size	Integer value representing a point size. A point size of twelve is one-sixth of an inch tall.

Component	Description
foreground color	Integer RGB color value, representing the color of the text. See the RGB() 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
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** statement provides loop control. This statement requires a numeric variable (identified by the *var_name* parameter). A **For** 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** 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** 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 **CommandInfo()** 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.

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.

<i>pattern</i> character	Role in formatting results:
#	The result will include one or more digits from the value.
	If the <i>pattern</i> 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 <i>pattern</i> 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 <i>pattern</i> 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.
, (comma)	The <i>pattern</i> 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 produce 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 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.

Note: 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.

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.

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

FrontWindow() function

Purpose

Returns the Integer identifier of the active window.

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** 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** 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** 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 ^ 0.333333333333
End Function
```

See Also

Declare Function statement, Declare Sub statement, Sub...End Sub 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.

Note: 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

GetFolderPath\$() function

Purpose

Return the path of a special MapInfo Professional or Windows folder.

Syntax

```
GetFolderPath$( folder_id )
```

folder_id is one of the following values:

```
FOLDER_MI_APPDATA  
FOLDER_MI_LOCAL_APPDATA  
FOLDER_MI_PREFERENCE  
FOLDER_MI_COMMON_APPDATA  
FOLDER_APPDATA  
FOLDER_LOCAL_APPDATA  
FOLDER_COMMON_APPDATA  
FOLDER_COMMON_DOCS  
FOLDER_MYDOCS  
FOLDER_MYPICS
```

Return Value

String

Description

Given the id of a special MapInfo or Windows folder, GetFolderPath\$() function returns the path of the folder. An example of a special Windows folder is the My Documents folder. An example of a special MapInfo folder is the preference folder; the default location to which MapInfo Professional writes out the preference file.

The location of many of these folders varies between versions of Windows. They can also vary depending on which user is logged in. Note that FOLDER_MI_APPDATA, FOLDER_MI_LOCAL_APPDATA and FOLDER_MI_COMMON_APPDATA may not exist. Before attempting to access those folders, test for their existence by using FileExists(). FOLDER_MI_PREFERENCE always exists

Ids beginning in FOLDER_MI return the path for folders specific to MapInfo Professional. The rest of the ids return the path for Windows folders 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 sub
```

See Also

LocateFile\$() function

GetMetadata\$() function**Purpose**

Retrieves metadata from a table.

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 the **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

GetSeamlessSheet() function**Purpose**

Prompts the user to select one sheet from a seamless table, and then returns the name of the chosen sheet.

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**

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 (i.e. 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** and **Exit For** statements 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.

Syntax (5.5 and later)

```
Graph
    label_column , expr [ , ... ]
From table
    [ Position ( x , y ) [ Units paperunits ] ]
    [ Width width [ Units paperunits ] ]
    [ Height 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

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

Syntax (pre-version 5.5)

```
Graph
    label_column , expr [ , ... ]
From table
    [ Position ( x , y ) [ Units paperunits ] ]
    [ Width width [ Units paperunits ] ]
    [ Height height [ Units paperunits ] ]
    [ Min | Max ]
```

label_column is the name of the column to use for labelling the y-axis

expr is an expression providing values to be graphed

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

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 it 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 the **Series In Columns** is included, then the graph series are based on the table columns. Otherwise the series are based on the table rows.

Graph commands in workspaces or programs that were created prior to version 5.5 will still create a 5.0 graph window. When a 5.0 graph window is active in MapInfo Professional 5.5 or later, the 5.0 graph menu will be also be active, so the user can modify the graph using the 5.0 editing dialogs. The Create Graph wizard will always created a 5.5 or later version graph window.

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** statements can re-configure the specifics of the graph (for example, the graph rotation, graph type, title, etc.).

MapInfo Professional 's Window > Graph dialog 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 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

HomeDirectory\$() function

Purpose

Returns a string indicating the user's home directory path.

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. This is the directory containing Windows system files, such as SYSTEM.INI and WIN.INI. In a networked environment, each user has a private Windows directory, to allow each user to have a unique configuration.

Example

```
Dim s_home_dir As String  
s_home_dir = HomeDirectory$( )
```

See Also

[ApplicationDirectory\\$\(\) function](#), [ProgramDirectory\\$\(\) function](#), [SystemInfo\(\) function](#)

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 Elself 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 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 Elself 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 Elself 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 statement is more elegant than an If statement with many Elself 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.

Syntax 1 (for MIF/MID files, PICT files, or MapInfo for DOS files)

```
Import file_name
[ Type file_type ]
[ Into table_name ]
[ Overwrite ]
```

Syntax 2 (for DXF files)

```
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 . . . ]
```

Syntax 3 (for GML files)

```
Import file_name
[ Type "GML" ]
[ Layer layer_name ]
[ Into table_name ]
[ Style Auto [ On | Off ] ]
```

Syntax 4 (for GML 2.1 files)

```
Import file_name
[ Type "GML21" ]
[ Layer layer_name ]
[ Into table_name ]
[ Overwrite ]
[ Coordsys clause ]
```

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, GML GML21, or PICT)

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

Layer *layer_name* is a String representing the name of a layer in the GML file.

Description

The Import statement creates a new MapInfo table by importing the contents of an existing file.

Note: To create a MapInfo table based on a spreadsheet or database file, use the Register Table statement, not the Import statement.

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 filename. For example, on Windows, if you import the text file "WORLD.MIF", the new table's default name is "WORLD.TAB".

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).
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
Type "GML21"	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.

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.

Import Options for DXF Files

If you import a DXF file, the Import statement can include the following DXF-specific clauses.

Note: 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 . . .

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

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 file 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.

Importing GML Files

file_name is the name of the GML 2.1 file to import.

Type is "GML21" for GML 2.1 files.

layer_name is the name of the GML layer.

table_name is the MapInfo table name.

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.

Note: If the *Coordsys* clause does not match the projection of the GML file, your data may not import correctly. The *coordsys* must match the *coordsys* of the data in the GML file. It will not transform the data from one projection to another.

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
```

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
```

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
```

See Also

Export statement

Include statement

Purpose

Incorporates the contents of a separate text file as part of a MapBasic program.

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 Open File

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 #.

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, EOF() returns FALSE; if the end-of-file was reached before the input was completed, EOF() returns TRUE.

For an example of the Input # statement, see the sample program NIEWS (Named Views).

The following data types are not available with the Input # statement: Alias, Pen, Brush, Font, Symbol, and Object.

See Also

EOF() function, Line Input statement, Open File statement, Write # statement

Insert statement**Purpose**

Appends new rows to an open table.

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 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.

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.

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 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.

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() 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.

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.

IsPenWidthPixels() function

Purpose

The IsPenWidthPixels function determines if a pen width is in pixels or in points.

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 StylAttr() 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](#)

Kill statement

Purpose

Deletes a file.

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.

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 Labelinfo().

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.

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 one of these functions: LabelFindFirst(), LabelFindNext(), or LabelFindByID().
2. If the function you called in step 1 did not return FALSE, you can query the current label by calling Labelinfo().

To continue querying additional labels, return to step 1.

Example

For an example, see Labelinfo().

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.

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 LabelFindFirst() to begin querying labels, you can call LabelFindNext() to advance to the next label in the same layer.

Example

For an example, see Labelinfo().

See Also

[LabelFindByID\(\) function](#), [LabelFindFirst\(\) function](#), [Labelinfo\(\) function](#)

Labelinfo() function**Purpose**

Returns information about a label in a map.

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.

Note: Labels are different than text objects. To query a text object, call functions such as ObjectInfo() or ObjectGeography().

Before calling Labelinfo(), you must initialize MapBasic's internal label pointer by calling LabelFindFirst(), LabelFindNext(), or LabelFindByID(). See example below.

The attribute parameter must be one of the codes from the following table; codes are defined in MAPBASIC.DEF.

attribute code	Labelinfo() Return Value
LABEL_INFO_ANCHORX	Float value, indicating the x coordinate of the label's anchor location.
LABEL_INFO_ANCHORY	Float value, indicating the y coordinate of the label's anchor location.
LABEL_INFO_DRAWN	Logical value; TRUE if label is currently visible.
LABEL_INFO_EDIT	Logical value; TRUE if label has been edited.
LABEL_INFO_EDIT_ANCHOR	Logical value; TRUE if label has been moved.
LABEL_INFO_EDIT_ANGLE	Logical value; TRUE if label's rotation angle has been modified.
LABEL_INFO_EDIT_FONT	Logical value; TRUE if label's font has been modified.
LABEL_INFO_EDIT_OFFSET	Logical value; TRUE if label's offset has been modified.
LABEL_INFO_EDIT_PEN	Logical value; TRUE if callout line's Pen style has been modified.
LABEL_INFO_EDIT_POSITION	Logical value; TRUE if label's position (relative to anchor) has been modified.
LABEL_INFO_EDIT_TEXT	Logical value; TRUE if label's text has been modified.
LABEL_INFO_EDIT_TEXTARROW	Logical value; TRUE if label's text arrow setting has been modified.
LABEL_INFO_EDIT_TEXTLINE	Logical value; TRUE if callout line has been moved.
LABEL_INFO_EDIT_VISIBILITY	Logical value; TRUE if label visibility has been set to OFF.
LABEL_INFO_OBJECT	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.

attribute code	Labelinfo() Return Value
LABEL_INFO_OFFSET	Integer value between 0 and 50, indicating the distance (in points) the label is offset from its anchor location.
LABEL_INFO_POSITION	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: <ul style="list-style-type: none"> • 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_ROWID	Integer value, representing the ID number of the row that owns this label; returns zero if no label exists.
LABEL_INFO_SELECT	Logical value; TRUE if label is selected.
LABEL_INFO_TABLE	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.

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

LayerInfo() function

Purpose

Returns information about a layer in a Map window.

Syntax

```
LayerInfo( map_window_id , layer_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 MapperInfo()

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() 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 (0 is the cosmetic 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.

<i>attribute code</i>	LayerInfo() Return Value
LAYER_INFO_NAME	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).
LAYER_INFO_EDITABLE	Logical value; TRUE if the layer is editable.
LAYER_INFO_LBL_PARTIALSEGS	Logical value; TRUE if the Label Partial Objects check box is selected for this layer.
LAYER_INFO_SELECTABLE	Logical value; TRUE if the layer is selectable.
LAYER_INFO_PATH	String value representing the full directory path of the table associated with the map layer.
LAYER_INFO_ZOOM_LAYERED	Logical; TRUE if zoom-layering is enabled.
LAYER_INFO_ZOOM_MIN	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.)

<i>attribute code</i>	LayerInfo() Return Value
LAYER_INFO_ZOOM_MAX	Float value, indicating the maximum zoom value at which the layer displays.
LAYER_INFO_COSMETIC	Logical; TRUE if this is the Cosmetic layer.
LAYER_INFO_DISPLAY	SmallInt, indicating how and whether this layer is displayed; return value will be one of these values: <ul style="list-style-type: none"> • LAYER_INFO_DISPLAY_OFF (the layer is not displayed); • LAYER_INFO_DISPLAY_GRAPHIC (objects in this layer appear in their “default” style—the style saved in the table); • LAYER_INFO_DISPLAY_GLOBAL (objects in this layer are displayed with a “style override” specified in Layer Control); • LAYER_INFO_DISPLAY_VALUE (objects in this layer appear as thematic shading)
LAYER_INFO_OVR_LINE	Pen style used for displaying linear objects.
LAYER_INFO_OVR_PEN	Pen style used for displaying the borders of filled objects.
LAYER_INFO_OVR_BRUSH	Brush style used for displaying filled objects.
LAYER_INFO_OVR_SYMBOL	Symbol style used for displaying point objects.
LAYER_INFO_OVR_FONT	Font style used for displaying text objects.
LAYER_INFO_LBL_CURFONT	For applications compiled with MapBasic 4.0 or later, this query always returns false. 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).
LAYER_INFO_LBL_FONT	Font style used in labels.
LAYER_INFO_LBL_EXPR	String value: the expression used in labels.
LAYER_INFO_LBL_LT	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: <ul style="list-style-type: none"> • LAYER_INFO_LBL_LT_NONE (no line) • LAYER_INFO_LBL_LT_SIMPLE (simple line) • LAYER_INFO_LBL_LT_ARROW (line with an arrow-head)
LAYER_INFO_LBL_PARALLEL	Logical value: TRUE if layer is set for parallel labels.

<i>attribute code</i>	LayerInfo() Return Value
LAYER_INFO_LBL_POS	Smallint value, indicating label position. Return value will match one of these values (T=Top, B=Bottom, C=Center, R=Right, L=Left): <ul style="list-style-type: none"> • LAYER_INFO_LBL_POS_TL • LAYER_INFO_LBL_POS_TC • LAYER_INFO_LBL_POS_TR • LAYER_INFO_LBL_POS_CL • LAYER_INFO_LBL_POS_CC • LAYER_INFO_LBL_POS_CR • LAYER_INFO_LBL_POS_BL • LAYER_INFO_LBL_POS_BC • LAYER_INFO_LBL_POS_BR
LAYER_INFO_LBL_VISIBILITY	Smallint value, indicating whether labels are visible; see the Visibility clause of the Set Map statement. Return value will be one of these values: <ul style="list-style-type: none"> • LAYER_INFO_LBL_VIS_ON (labels always visible) • LAYER_INFO_LBL_VIS_OFF (labels never visible) • LAYER_INFO_LBL_VIS_ZOOM (labels visible when in zoom range)
LAYER_INFO_LBL_ZOOM_MIN	Float value, indicating the minimum zoom distance for this layer's labels.
LAYER_INFO_LBL_ZOOM_MAX	Float value, indicating the maximum zoom distance for this layer's labels.
LAYER_INFO_LBL_AUTODISPLAY	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	Logical value; TRUE if overlapping labels are allowed.
LAYER_INFO_LBL_DUPLICATES	Logical value; TRUE if duplicate labels are allowed.
LAYER_INFO_LBL_OFFSET	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	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	Logical value; TRUE if the Label Partial Segments check box is checked for this layer.
<i>attribute code</i>	LayerInfo() Return Value
LAYER_INFO_ARROWS	Logical value; TRUE if layer displays direction arrows on linear objects.
LAYER_INFO_NODES	Logical value; TRUE if layer displays object nodes.
LAYER_INFO_CENTROIDS	Logical value; TRUE if layer displays object centroids.
LAYER_INFO_SELECTABLE	Logical value; TRUE if the layer is selectable.

<i>attribute code</i>	LayerInfo() Return Value
LAYER_INFO_PATH	String value representing the full directory path of the table associated with the map layer.
LAYER_INFO_TYPE	SmallInt value, indicating this layer's file type: <ul style="list-style-type: none"> • LAYER_INFO_TYPE_NORMAL for a normal layer; • LAYER_INFO_TYPE_COSMETIC for the Cosmetic layer; • LAYER_INFO_TYPE_IMAGE for a raster image layer; • LAYER_INFO_TYPE_THEMATIC for a thematic layer. • LAYER_INFO_TYPE_GRID for a grid image layer. • LAYER_INFO_TYPE_WMS for a layer from a Web Service Map.
LAYER_HOTLINK_EXPR	Returns the layer's Hotlink filename expression.
LAYER_HOTLINK_MODE	Returns the layer's Hotlink mode, one of the following predefined values: <ul style="list-style-type: none"> • HOTLINK_MODE_LABEL • HOTLINK_MODE_OBJ • HOTLINK_MODE_BOTH
LAYER_HOTLINK_RELATIVE	Returns True if the relative path option is on, False otherwise.

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

MapperInfo() function

Layout statement

Purpose

Opens a new layout window.

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 the 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 workspace.

```
Layout Position (0, 0) Width 2 Height 4
```

See Also

[Open Window statement](#)

LCase\$() function

Purpose

Returns a lower-case equivalent of a string.

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.

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.

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	LegendFrameInfo() Return Value
FRAME_INFO_TYPE	Returns one of the following predefined constant indicating frame type: <ul style="list-style-type: none"> FRAME_TYPE_STYLE FRAME_TYPE_THEME
FRAME_INFO_MAP_LAYER_ID	Returns the id of the layer to which the frame corresponds.
FRAME_INFO_REFRESHABLE	Returns true if the frame was created without the Norefresh keyword. Always returns true for theme frames.
FRAME_INFO_POS_X	Returns the distance of the frames upper left corner from the left edge of the legend canvas (in paper units).
FRAME_INFO_POS_Y	Returns the distance of the frame's upper left corner from the top edge of the legend canvas (in paper units).
FRAME_INFO_WIDTH	Returns the width of the frame (in paper units).
FRAME_INFO_HEIGHT	Returns the height of the frame (in paper units).

Attribute codes	LegendFrameInfo() Return Value
FRAME_INFO_TITLE	Returns the title of a style frame or theme frame.
FRAME_INFO_TITLE_FONT	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	Returns the subtitle of a style frame or theme frame.
FRAME_INFO_SUBTITLE_FONT	Same as title font.
FRAME_INFO_BORDER_PEN	Returns the pen used to draw the border.
FRAME_INFO_NUM_STYLES	Returns the number of styles in a frame. Zero if theme frame.
FRAME_INFO_VISIBLE	Returns true if the frame is visible (theme frames can be invisible).
FRAME_INFO_COLUMN	Returns the legend attribute column name as a string if there is one. Returns an empty string for a theme frame.
FRAME_INFO_LABEL	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.

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	LegendInfo() Return Value
LEGEND_INFO_MAP_ID	Returns the ID of the parent map window (can also get this value by issuing WindowInfo() with the WIN_INFO_TABLE code).
LEGEND_INFO_ORIENTATION	Returns predefined value to indicate the layout of the legend: <ul style="list-style-type: none"> • ORIENTATION_PORTRAIT • ORIENTATION_LANDSCAPE • ORIENTATION_CUSTOM
LEGEND_INFO_NUM_FRAMES	Returns the number of frames in the legend.
LEGEND_INFO_STYLE_SAMPLE_SIZE	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.

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	LegendStyleInfo() Return Values
LEGEND_STYLE_INFO_TEXT	Returns the text of the style.
LEGEND_STYLE_INFO_FONT	Returns the font of the style.
LEGEND_STYLE_INFO_OBJ	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.

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

Like() function**Purpose**

Returns TRUE or FALSE to indicate whether a string satisfies pattern-matching criteria.

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 wild-card 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 Input 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.

Syntax

```
LocateFile$( file_id )
```

file_id is one of the following values

Value	Description
LOCATE_PREF_FILE	preference file (mapinfow.prf)
LOCATE_DEF_WOR	default workspace file (mapinfow.wor)
LOCATE_CLR_FILE	color file (mapinfow.clr)
LOCATE_PEN_FILE	pen file (mapinfow.pen)
LOCATE_FNT_FILE	symbol file (mapinfow.fnt)
LOCATE_ABB_FILE	abbreviation file (mapinfow.abb)
LOCATE_PRJ_FILE	projection file (mapinfow.prj)
LOCATE_MNU_FILE	menu file (mapinfow.mnu)
LOCATE_CUSTSYMB_DIR	custom symbol directory (custsymb)
LOCATE_THMTMPLT_DIR	theme template directory (thmtmpl)
LOCATE_GRAPH_DIR	graph support directory (GraphSupport)

Returns

String

Description

Given the ID of a MapInfo application data file, this function returns the location where MapInfo Professional found that file. In versions prior to 6.5 these files were, for the most part, installed into the program directory (same directory as mapinfow.exe). As of 6.5, 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 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.

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.

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 statements) 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 (i.e. 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.

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 the 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 the description of the 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](#), [StyleAttr\(\) function](#)

MakeCustomSymbol() function

Purpose

Returns a Symbol value based on a bitmap file.

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 the **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 the description of the Symbol clause for information about other symbol types.

The following table describes how the customstyle argument controls the symbol's style:

<i>customstyle</i> value	Symbol Style
0	Both the Show Background setting and the Apply Color setting are off; the symbol appears in its default state. 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.

Example

```
Include "mapbasic.def"
Dim sym_marker As Symbol
sym_marker = MakeCustomSymbol("CAR1-64.BMP", BLUE, 18, 0)
```

See Also

CurrentSymbol() function, **MakeFontSymbol() function**, **MakeSymbol() function**, **StyleAttr() function**, **Symbol clause**

MakeFont() function

Purpose

Returns a Font value.

Syntax

```
MakeFont( fontname, style, size, forecolor, backcolor )
```

fontname is a text string specifying a font (for example, "Helv"). 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 the 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 or Set Style).

See the description of the Font clause for more information about Font settings.

Example

```
Include "mapbasic.def"  
Dim big_title As Font  
big_title = MakeFont("Helvetica", 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.

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 the **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 the description of the Symbol clause for information about other symbol types.

The following table describes how the fontstyle argument controls the symbol's style:

<i>fontstyle</i> 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.

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 the **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, Create Polyline, Set Style, or Set Map).

See the description of the 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

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.

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 the **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 setting as a parameter (such as Create Point, Set Map, Set Style, or Shade).

To create a symbol from a character in a TrueType font, call MakeFontSymbol().

To create a symbol from a bitmap file, call MakeCustomSymbol().

See the description of the 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.

Syntax

```
Map From table [ , table ... ]  
  [ Position ( x, y ) [ Units paperunits ] ]  
  [ Width window_width [ Units paperunits ] ]  
  [ Height window_height [ Units paperunits ] ]  
  [ { Min | Max } ]
```

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 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 statements.

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, 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 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.

Example

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 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
```

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.

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.

Returns

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	Return Value
MAP3D_INFO_SCALE	Float result representing the 3DMaps scale factor.
MAP3D_INFO_RESOLUTION_X	Integer result representing the X resolution of the grid(s) in the 3DMap window.
MAP3D_INFO_RESOLUTION_Y	Integer result representing the Y resolution of the grid(s) in the 3DMap window.
MAP3D_INFO_BACKGROUND	Integer result representing the background color, see the RGB function.

Attribute	Return Value
MAP3D_INFO_UNITS	String representing the map's abbreviated area unit name, for example, "mi" for miles.
MAP3D_INFO_LIGHT_X	Float result representing the X coordinate of the Light in the scene.
MAP3D_INFO_LIGHT_Y	Float result representing the Y coordinate of the Light in the scene.
MAP3D_INFO_LIGHT_Z	Float result representing the Z coordinate of the Light in the scene.
MAP3D_INFO_LIGHT_COLOR	Integer result representing the Light color, see the RGB function.
MAP3D_INFO_CAMERA_X	Float result representing the X coordinate of the Camera in the scene.
MAP3D_INFO_CAMERA_Y	Float result representing the Y coordinate of the Camera in the scene.
MAP3D_INFO_CAMERA_Z	Float result representing the Z coordinate of the Camera in the scene.
MAP3D_INFO_CAMERA_FOCAL_X	Float result representing the X coordinate of the Cameras FocalPoint in the scene.
MAP3D_INFO_CAMERA_FOCAL_Y	Float result representing the Y coordinate of the Cameras FocalPoint in the scene.
MAP3D_INFO_CAMERA_FOCAL_Z	Float result representing the Z coordinate of the Cameras FocalPoint in the scene.
MAP3D_INFO_CAMERA_VU_1	Float result representing the first value of the ViewUp Unit Normal Vector
MAP3D_INFO_CAMERA_VU_2	Float result representing the second value of the ViewUp Unit Normal Vector.
MAP3D_INFO_CAMERA_VU_3	Float result representing the third value of the ViewUp Unit Normal Vector.
MAP3D_INFO_CAMERA_VPN_1	Float result representing the first value of the View-Plane Unit Normal Vector.
MAP3D_INFO_CAMERA_VPN_2	Float result representing the second value of the View-Plane Unit Normal Vector.
MAP3D_INFO_CAMERA_VPN_3	Float result representing the third value of the View-Plane Unit Normal Vector.
MAP3D_INFO_CAMERA_CLIP_NEAR	Float result representing the cameras near clipping plane.
MAP3D_INFO_CAMERA_CLIP_FAR	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.

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

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.

<i>attribute setting</i>	MapperInfo() Return Value
MAPPER_INFO_AREAUNITS	String representing the map's abbreviated area unit name, for example, "sq mi" for square miles.
MAPPER_INFO_CENTERX	The x-coordinate of the Map window's center.
MAPPER_INFO_CENTERY	The y-coordinate of the Map window's center.
MAPPER_INFO_COORDSYS_CLAUSE	string result, indicating the window's CoordSys clause.
MAPPER_INFO_COORDSYS_NAME	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_DISPLAY	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: <ul style="list-style-type: none"> • MAPPER_INFO_DISPLAY_SCALE • MAPPER_INFO_DISPLAY_ZOOM • MAPPER_INFO_DISPLAY_POSITION
MAPPER_INFO_DISPLAY_DMS	A SmallInt that indicates whether the map displays coordinates in decimal degrees; degrees, minutes, seconds; o in the Military Reference system. Return value will be one of the following: <ul style="list-style-type: none"> • MAPPER_INFO_DISPLAY_DECIMAL for degrees decimal coordinates (0) • MAPPER_INFO_DISPLAY_DEGMINSEC for degrees, minutes, seconds coordinates (1) • MAPPER_INFO_DISPLAY_MGRS for Military Grid System coordinates (2)
MAPPER_INFO_DISTUNITS	String representing the map's abbreviated distance unit name, for example, "mi" for miles.

<i>attribute setting</i>	MapperInfo() Return Value
MAPPER_INFO_EDIT_LAYER	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_LAYERS	Returns number of layers in the Map window as a SmallInt (excludes the cosmetic layer).
MAPPER_INFO_MAXX	The largest x-coordinate shown in the window.
MAPPER_INFO_MAXY	The largest y-coordinate shown in the window.
MAPPER_INFO_MINX	The smallest x-coordinate shown in the window.
MAPPER_INFO_MINY	The smallest y-coordinate shown in the window.
MAPPER_INFO_NUM_THEMATIC	Small integer, indicating the number of thematic layers in this Map window.
MAPPER_INFO_SCALE	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_SCROLLBARS	Logical value indicating whether the Map window shows scrollbars.
MAPPER_INFO_XYUNITS	String representing the map's abbreviated coordinate unit name, for example, "degree". Small integer, indicating whether the map displays coordinates in decimal degrees, DMS (degrees, minutes, seconds), or Military Grid Reference System format. Return value will be one of these: <ul style="list-style-type: none"> • MAPPER_INFO_DISPLAY_DECIMAL • MAPPER_INFO_DISPLAY_DMS • MAPPER_INFO_DISPLAY_MGRS (Military Grid Reference System)
MAPPER_INFO_ZOOM	The Map window's current zoom value (i.e. the East-West distance currently displayed in the Map window), specified in MapBasic's current distance units; see Set Distance Units.
MAPPER_INFO_COORDSYS_CLAUSE_WITH_BOUNDS	String result, indicating the window's CoordSys clause including the bounds.
MAPPER_INFO_MOVE_DUPLICATE_NODES	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 move. The attribute.

<i>attribute setting</i>	MapperInfo() Return Value
MAPPER_INFO_DIST_CALC_TYPE	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 <ul style="list-style-type: none"> • MAPPER_INFO_DIST_SPHERICAL • MAPPER_INFO_DIST_CARTESIAN
MAPPER_INFO_CLIP_REGION	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_CLIP_TYPE	The type of clipping being implemented. Choices include: <ul style="list-style-type: none"> • MAPPER_INFO_CLIP_DISPLAY_ALL • MAPPER_INFO_CLIP_DISPLAY_POLYOBJ • MAPPER_INFO_CLIP_OVERLAY

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 it's 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 MapBasic statement.

Coordinate Value Returns

MapperInfo() does not return coordinates (i.e. 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 method is the method that has been the only option until MapInfo Professional 6.0. Using this method, the Overlap() function (Erase Outside) is used internally. Since the Overlap() function can't 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. never clipped.

The MAPPER_INFO_DISPLAY_ALL 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 uses the Windows Display to selectively perform clipping which mimics the functionality produced by MAPPER_INFO_CLIP_OVERLAY. 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 method. In general, this method should provide better performance than the MAPPER_INFO_CLIP_OVERLAY method.

Error Conditions

ERR_BAD_WINDOW error generated if parameter is not a valid window number

ERR_FCN_ARG_RANGE error generated if an argument is outside of the valid range

ERR_WANT_MAPPER_WIN error generated if window id is not a Map window

See Also

[LayerInfo\(\) function](#), [Set Distance Units statement](#), [Set Map statement](#)

Maximum() function

Purpose

Returns the larger of two numbers.

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 of another object.

Syntax

```
MBR( obj_expr )
```

obj_expr is an object expression

Return Value

Object (a rectangle)

Description

The MBR() function calculates the minimum bounding rectangle (or 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.

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; 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() instead of calling MenuItemInfoByHandler().

The attribute parameter is a numeric code (defined in MAPBASIC.DEF) from the following table:

<i>attribute setting</i>	<i>Return value</i>
MENUTITEM_INFO_ACCELERATOR	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 the Create Menu statement.
MENUTITEM_INFO_CHECKABLE	Logical: TRUE if this menu item is checkable (specified by the "!" prefix in the menu text)
MENUTITEM_INFO_CHECKED	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.
MENUTITEM_INFO_ENABLED	Logical: TRUE if the menu item is enabled
MENUTITEM_INFO_HANDLER	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.

<i>attribute setting</i>	Return value
MENUITEM_INFO_HELPMSG	String: the menu item's help message (as specified in the HelpMsg clause in Create Menu) or empty string if the menu item has no help message.
MENUITEM_INFO_ID	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.
MENUITEM_INFO_SHOWHIDEABLE	Logical: TRUE if this menu item has alternate menu text (i.e. 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 Create Menu or Alter Menu) and it has a caret (^) in the string.
MENUITEM_INFO_TEXT	String: the full text used (for example, in a Create Menu statement) to create the menu item.

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 MenuItemInfoByHandler() 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. For a listing of codes you can use, see MenuItemInfoByHandler().

See Also

MenuItemInfoByHandler() function

Metadata statement

Purpose

Manages a table's metadata.

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 MapInfo 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( ))
```

Note: 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.

Note: 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.

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.

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.

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.

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

Month() function**Purpose**

Returns the month component (1 - 12) of a date value.

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.

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 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.

`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. The entire `Ignore` clause is optional, as are the `Min` and `Max` subclauses within it (e.g., only a `Min` or only a `Max`, or both may occur).

Normally, if one object is contained within another object, the distance between the objects is zero. For example, if the *From* table is *WorldCaps* and the *To* table 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.

The `Data` clause can be used to mark which *fromtable* object and which *totable* object the result came from.

Description

Every object in the *fromtable* is considered. For each object in the *fromtable*, the nearest object in the *totable* is found. If *N* is present, 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 present, 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 (i.e., 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.

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 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 (i.e., a distance of 100 should only occur in the first pass and never in the second pass).

Data Clause

```
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 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). 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
```

It is currently not possible to fill in a column in the *intotable* with the distance. However, this can be easily accomplished after the Nearest operation is completed by using the **TABLE > UPDATE COLUMN...** functionality from the menu or by using the `Update MapBasic` 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 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 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 selection To atm Into result Ignore Max 5 Units "mi" Data
where=business,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 customer To coastline Into result Data
who=customer.name,where=customer.address,coast_loc=coastline.county,type=coastli
ne.designation
```

If we don't care about customer locations which are greater than 30 miles from any coastline:

```
Nearest From customer To coastline Into result Ignore Max 30 Units "mi" Data
who=customer.name,where=customer.address,coast_loc=coastline.county,type=coastli
ne.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 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.

Syntax

```
Note message
```

message is an expression to be displayed in a dialog

Description

The Note statement creates a dialog to display a message. The dialog contains an OK menu button; the message dialog 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. 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.

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.

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, 1995:

```
NumberToDate(19951231)
```

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(19950101)

' i_elapsed now contains the number of days
' since January 1, 1995
```

See Also

[StringToDate\(\) function](#)

NumCols() function

Purpose

Returns the number of columns in a specified table.

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.

Note: 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 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.

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 dialog box 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).

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.

Syntax

```
ObjectDistance(object1, object2, unit_name)
```

object1 and *object2* are object expressions.

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

Returns

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.

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.

<i>attribute setting</i>	Return value (Float)
OBJ_GEO_MINX	minimum x coordinate of an object's minimum bounding rectangle (MBR), unless object is a line; if object is a line, returns same value as OBJ_GEO_LINEBEGX.
OBJ_GEO_MINY	minimum y coordinate of object's MBR. For lines, returns OBJ_GEO_LINEBEGY value.
OBJ_GEO_MAXX	maximum x coordinate of object's MBR. Does not apply to Point objects. For lines, returns OBJ_GEO_LINEENDX value.
OBJ_GEO_MAXY	maximum y coordinate of the object's MBR. Does not apply to Point objects. For lines, returns OBJ_GEO_LINEENDY value.
OBJ_GEO_ARCBEGANGLE	beginning angle of an Arc object.
OBJ_GEO_ARCENDANGLE	ending angle of an Arc object.
OBJ_GEO_LINEBEGX	x coordinate of the starting node of a Line object.
OBJ_GEO_LINEBEGY	y coordinate of the starting node of a Line object.
OBJ_GEO_LINEENDX	x coordinate of the ending node of a Line object.
OBJ_GEO_LINEENDY	y coordinate of the ending node of a Line object.
OBJ_GEO_POINTX	x coordinate of a Point object.
OBJ_GEO_POINTY	y coordinate of a Point object.
OBJ_GEO_ROUNDRAIDUS	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	returns a point object for centroid of regions, collections, multi-points, and polylines. This is most commonly used with the Alter Object statement.
OBJ_GEO_TEXTLINEX	x coordinate of the end of a Text object's label line.
OBJ_GEO_TEXTLINEY	y coordinate of the end of a Text object's label line.
OBJ_GEO_TEXTANGLE	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 object's MBR.
OBJ_GEO_MAXX (3)	maximum x coordinate of object's MBR.
OBJ_GEO_MAXY (4)	maximum y coordinate of 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.

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 *N*th polygon of a region or the *N*th section of a polyline.

Note: 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.

<i>attribute</i> Setting	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, Rounded Rectangle.
OBJ_INFO_BRUSH (3)	Brush style is returned; this query is only valid for the following object types: Ellipse, Frame, Region, Rectangle, Rounded Rectangle.
OBJ_INFO_TEXTFONT (2)	Font style is returned; this query is only valid for Text objects. Note: 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 <i>N</i> th polygon of a region or the <i>N</i> th section of a polyline. Note: 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).

<i>attribute</i> Setting	Return Value
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.
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, FALSE - if object is empty.
OBJ_INFO_REGION (8)	Object value representing region part of a collection object. If collection object does not have a region, it returns empty region. This query is valid only for collection objects
OBJ_INFO_PLINE (9)	Object value representing polyline part of a collection object. If collection object does not have a polyline, it returns empty polyline object. This query is valid only for collection objects
OBJ_INFO_MPOINT (10)	Object value representing Multipoint part of a collection object. If collection object does not have a Multipoint, it returns 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. Return empty string if units are not specified.
OBJ_INFO_HAS_Z(14)	Logical, indicating whether object has Z values.
OBJ_INFO_HAS_M(15)	Logical, indicating whether 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.

Object type code	Corresponding object type
OBJ_TYPE_ARC	Arc object
OBJ_TYPE_ELLIPSE	Ellipse / circle objects
OBJ_TYPE_LINE	Line object
OBJ_TYPE_PLINE	Polyline object
OBJ_TYPE_POINT	Point object
OBJ_TYPE_FRAME	Layout window Frame object
OBJ_TYPE_REGION	Region object
OBJ_TYPE_RECT	Rectangle object
OBJ_TYPE_ROUNDRECT	Rounded rectangle object
OBJ_TYPE_TEXT	Text object
OBJ_TYPE_MULTIPPOINT	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.

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 the **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**

ObjectNodeM() function

Purpose

Returns the m-value of a specific node in a region, polyline or multipoint object.

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 object does not support m values or m-value for this node is not defined, then, error is set.

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

ObjectNodeX() function**Purpose**

Returns the x-coordinate of a specific node in a region or polyline object.

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 s a positive integer. 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 will return the value in the coordinate system currently in use by MapBasic; by default, MapBasic uses a longitude, latitude coordinate system. See the **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.

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 the description of the **ObjectNodeX()** function for more information.

Example

See the **ObjectNodeX()** function description.

See Also

Alter Object statement, **ObjectGeography() function**, **ObjectInfo() function**, **ObjectNodeX() function**, **Set CoordSys statement**

ObjectNodeZ() function**Purpose**

Returns the z-coordinate 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

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.

Syntax

```
Objects Check From tablename Into Table tablename
    [SelfInt [Symbol Clause] ]
    [Overlap [Pen Clause] [Brush Clause] ]
    [Gap Area [Unit Units] [Pen Clause] [Brush Clause] ] ]
```

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** and **Brush** clauses 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 perform the overlap parameter 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 (i.e., regions, rectangles, rounded rectangles, or ellipses).

Syntax

```
Objects Clean From tablename
    [Overlap]
    [Gap Area [Unit Units] ]
```

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.

Note: **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** clause. If the **Units** sub-clause is not present, then the **Gap Area** value will be 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** command. Any gaps that the **Objects Check** command flags will be removed with the **Objects Clean** command.

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's Objects > Combine command.

Syntax

```
Objects Combine
[ Into Target ]
[ Data column = expression [ , column = expression ... ] ]
```

column is the name of a column in the table being modified

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's Objects > Combine menu item. For an introduction to this operation, see the discussion of the Objects > Combine menu item in the *MapInfo Professional Reference*. To see a demonstration of the **Objects Combine** statement, run MapInfo, open MapInfo's 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 are not allowed and point and text objects are not allowed.

Into Target clause

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.

Data clause

The **Data** clause controls data aggregation. (For an introduction to data aggregation, see the description of the Objects > Combine operation in the *MapInfo Professional Reference*.) 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
<i>col_name</i> = <i>col_name</i>	The column contents are not altered.
<i>col_name</i> = <i>value</i>	MapBasic stores the hard-coded <i>value</i> in the column of the result object.
<i>col_name</i> = Sum(<i>col_name</i>)	Used only for numeric columns. The column in the result object contains the sum of the column values of all objects being combined.
<i>col_name</i> = Avg(<i>col_name</i>)	Used only for numeric columns. The column in the result object contains the average of column values of all objects in the group.
<i>col_name</i> =WtAvg(<i>colname</i> , <i>wtcolname</i>)	Used only for numeric columns. MapInfo Professional performs weighted averaging, averaging all of the <i>col_name</i> column values, and weighting the average calculation based on the contents of the <i>wt_colname</i> 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.

Syntax

```
Objects Disaggregate [IntoTable name]
  [ All | Collection ]
  [ Data column_name = expression
    [ , column_name = expression ... ]
```

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 3 polygons, and one of those polygons is a Hole in another polygon, then the output will be 2 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 Multipoint. 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** command, and will produce an error.

The **Collection** switch 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** switch, 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's Objects Enclose command.

Syntax

```
Objects Enclose
    [ Into Table tablename]
    [ Region ]
```

tablename 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 will not remove the original input objects. No data aggregation is done.

The optional **Region** clause will allow closed objects (regions, rectangles, rounded rectangles and ellipses) to be used as input to the **Objects Enclose** operation. 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 (i.e., points, multipoints, collections and text) will cause the operation to produce an error. If closed objects exist in the selection, and the **Region** switch is not present, then those objects will be ignored.

The Objects Enclose statement corresponds to MapInfo's Objects > Enclose menu item. For an introduction to this operation, see the discussion of the Objects > Enclose menu item in the *MapInfo Professional Reference*. 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 will place the objects created by this command into the table. Otherwise, the output objects will be placed in the same table that contains the input objects.

Example

This will select all the objects in a table called testfile, perform an Objects Enclose and store 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.

Syntax

```
Objects Erase Into Target
[ Data column_name = expression
[ , column_name = expression ... ]
```

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 Reference*.

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 statement **Set Target**.

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
<i>col_name</i> = <i>col_name</i>	MapBasic does not alter the value stored in the column.
<i>col_name</i> = <i>value</i>	MapBasic stores a specific value in the column. If it is a character column, the <i>value</i> can be a string; if it is a numeric column, the <i>value</i> can be a number.
<i>col_name</i> = Proportion(<i>col_name</i>)	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.

Objects, 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.

Syntax

```
Objects Intersect Into Target
[ Data column_name = expression
  [ , column_name = expression ... ] ]
```

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's Objects > Erase Outside menu item. For an introduction to using Objects > Erase Outside, see the *MapInfo Professional Reference*.

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
<i>col_name</i> = <i>col_name</i>	MapBasic does not alter the value stored in the column.
<i>col_name</i> = <i>value</i>	MapBasic stores a specific value in the column. If the column is a character column, the <i>value</i> can be a string; if the column is a numeric column, the <i>value</i> can be a number.
<i>col_name</i> = Proportion(<i>col_name</i>)	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 object(s) that *do not* overlap the current selection, while **Objects Erase** 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 the description of the **Objects Erase** statement.

Example

```
Objects Intersect Into Target
Data
    Field2=Proportion(Field2)
```

See Also

Create Object statement, **Overlap()** function

Object Move statement**Purpose**

Objects Move moves the objects obtained from the current selection within the input table.

Syntax

```
Objects Move
    Angle angle
    Distance distance
    [Units unit]
    [Type {Spherical | Cartesian}]
```

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, see the **Set Distance Units** statement.

The optional **Type** sub-clause lets you specify the type of distance calculation used to create the offset. If the **Spherical** type is used, 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 the **Cartesian** type is used, 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**

Objects Offset copies objects, obtained from the current selection, offset from the original objects.

Syntax

```
Objects Offset
  [Into Table intotable]
  Angle angle
  Distance distance
  [Units unit]
  [Type {Spherical | Cartesian}]
  [Data column = expression [, column = expression ...]]
```

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 Into Table, if the **Into** clause is present. Otherwise, the objects are placed into the same table as the input objects are obtained from (i.e., 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 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, see the **Set Distance Units** statement.

The optional **Type** sub-clause lets you specify the type of distance calculation used to create the offset. If the **Spherical** type is used, 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 the **Cartesian** type is used, 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.

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 statement **Set Target**. 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.

Syntax

```
Objects Pline Split At Node index
[Into Table name]
[Data column_name = expression
  [ , column_name = expression ... ] ]
```

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
<i>col_name</i> = <i>col_name</i>	Does not alter the value stored in the column.
<i>col_name</i> = <i>value</i>	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,FI
PS=FIPS,ISO_2=ISO_2,ISO_3=ISO_3,Pop_1994=Pop_1994,Pop_Grw_Rt=Pop_Grw_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 (i.e., polylines and arcs) or all closed (i.e., regions, rectangles, rounded rectangles, or ellipses). Mixed linear and closed objects can't be cleaned in one operation, and an error will result.

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] ] ]
```

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 will have the spike portion removed. The resulting cleaned object will replace the original input object. If any overlaps exist between the objects they will be 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 will consist of multiple polygons. One polygon will represent the non-overlapping portion, and a separate polygon will represent each overlapping section.

The **Node** and **Vector Tolerances** values will 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 will be interpreted in MapBasic's current distance unit. By default, MapBasic uses miles as the distance units; to change this unit, see 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 (i.e., 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 will reduce 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 will be interpreted in MapBasic's current distance unit.

The **Bend** tolerance is used to control how co-linear a series of nodes can be. Given 3 nodes, connect the 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 will be interpreted in MapBasic's current area unit. By default, MapBasic uses square miles as the area unit; to change this unit, see the **Set Area Units** statement.

Note: 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. An length and area calculations 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.

Syntax

```
Objects Split Into Target
[ Data column_name = expression
  [ , column_name = expression ... ]
```

Description

Use the **Objects Split** statement to split each of the target objects into multiple objects. Using **Objects Split** is equivalent to choosing MapInfo'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 statement **Set Target**.

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 and Objects Intersect (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
<i>col_name</i> = <i>col_name</i>	MapBasic does not alter the value stored in the column; each object resulting from the split operation retains the original column value.
<i>col_name</i> = <i>value</i>	MapBasic stores a specific value in the column. If the column is a character column, the <i>value</i> can be a string; if the column is a numeric column, the <i>value</i> can be a number. Each object resulting from the split operation retains the specified value.
<i>col_name</i> = Proportion(<i>col_name</i>)	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.

Syntax

```
Offset(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, and

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 or Perimeter, is the unit for the distance value. The DistanceType used is Spherical unless the Coordinate System is NonEarth. For NonEarth, **Cartesian** DistanceType 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

Syntax

```
OffsetXY(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, and

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* or *Perimeter*, is the unit for the distance values. The *DistanceType* used is *Spherical* unless the *Coordinate System* is *NonEarth*. For *NonEarth*, *Cartesian DistanceType* 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 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 will jump 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** or **End Program**) 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 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 will display an error dialog, then halt 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 this:

```
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**. 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.

Note: 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 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: 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**, **Put**)

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 the separate **CharSet** discussion

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**, **Fetch**, and **Select**) and another set of statements for using other files in general (for example, **Open File**, **Get**, **Put**, **Input #**, **Print #**).

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 **Input #** and **Line Input #** statements.

If you specify the **For Output** clause or the **For Append** clause, you can write to the file by issuing **Print #** and **Write #** statements.

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** and **Put** statements 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** and **Put** statements 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 the **CharSet** clause discussion 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 will run on only 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](#)

Open Report statement

Purpose

Loads a report into the Crystal Report Designer module.

Syntax

```
Open Report reportfilespec
```

reportfilespec is a full path and filename for an existing report file.

See Also

[Create Report From Table statement](#)

Open Table statement

Purpose

Opens a MapInfo Professional table for input/output.

Syntax

```
Open Table filespec [ As tablename ]
    [ Hide ] [ ReadOnly ] [ Interactive ] [ Password pwd ]
    [ NoIndex ] [ View Automatic ] [ DenyWrite ]
```

filespec specifies which MapInfo table to open

tablename is an "alias" name by which the table should be identified

pwd is the database-level password for the database, to be specified when database security is turned on. Applies to Access tables only.

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.

Note: 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, see the **Create Table** statement. The **Open Table** statement only applies to MapInfo tables; to use files that are in other formats, see the **Register Table** and **Open File** statements.

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 dialogs that display lists of open tables (for example, the File > Close dialog). 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 that if 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 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 1993 (for example, “C:\1993\SITES.TAB”) and another, perhaps newer copy of the table in a different directory (for example, “C:\1994\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 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 tab
```

See Also

Close Table statement, Create Table statement, Delete statement, Fetch statement, Insert statement, TableInfo() function, Update statement

Open Window statement

Purpose

Opens / shows a window.

Syntax

```
Open Window window_name
```

window_name is 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)

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**, **Graph**, **Browse**, **Layout**, and **Create Redistricter** statements).

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's Objects > Erase Outside command.

Syntax

```
Overlap( object1 , object2 )
```

object1 is an object; cannot be a point or text object

object2 is an object; 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.

Syntax

```
OverlayNodes ( input_object, overlay_object )
```

input_object is the object whose nodes will be included in the output object; may not be a point or text object

overlay_object is the object that will be intersected with *input_object*; 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 *input_object* intersects with *overlay_object*.

If the *input_object* was a closed object (region, rectangle, rounded rectangle or ellipse), **OverlayNodes()** returns a region object. If *input_object* was 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 will 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's Table > Maintenance > Pack Table command.

Syntax

```
Pack Table table { Graphic | Data | Graphic Data } [ Interactive ]
```

table is 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** statement.

Note: 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 (i.e. from the bottom of the Browser window), packing will not invalidate the custom labels.

Packing Access Tables

The **Pack Table** statement will save 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 will be lost during a pack.

Example

```
Pack Table parcels Data
```

See Also

[Open Table statement](#)

PathToDirectory\$() function**Purpose**

Given a file specification, return only the file's directory.

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 following 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**

Given a file specification, return just the filename (without the directory).

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 following 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

Given a complete file specification (such as "C:\MapInfo\Data\1995 Data.tab"), returns a string representing a table alias (such as "_1995_Data").

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 **TableInfo()** 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.

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 **Create Line**, 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, **CurrentPen()** or **MakePen()**) which returns a Pen value:

`Pen MakePen(1, 2, BLUE)`

With some MapBasic statements (for example, **Set Map**), 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).
pattern	Integer value from 1 to 118; see table below. Pattern 1 is invisible.
color	Integer RGB color value; see the RGB() function.

The available pen patterns appear in the next figure:

01		31		61		91	
02		32		62		92	
03		33		63		93	
04		34		64		94	
05		35		65		95	
06		36		66		96	
07		37		67		97	
08		38		68		98	
09		39		69		99	
10		40		70		100	
11		41		71		101	
12		42		72		102	
13		43		73		103	
14		44		74		104	
15		45		75		105	
16		46		76		106	
17		47		77		107	
18		48		78		108	
19		49		79		109	
20		50		80		110	
21		51		81		111	
22		52		82		112	
23		53		83		113	
24		54		84		114	
25		55		85		115	
26		56		86		116	
27		57		87		117	
28		58		88		118	
29		59		89			
30		60		90			

Example

```
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)
```

See Also

Alter Object statement, CreateLine() function, Create Pline statement, CurrentPen() function, MakePen() function, RGB() function, Set Style statement

PenWidthToPoints() function

Purpose

The **PenWidthToPoints** function returns a point size for a given pen width.

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**

PointsToPenWidth() function**Purpose**

The **PointsToPenWidth** function returns a pen width for a given point size.

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**

Perimeter() function

Purpose

Returns the perimeter of a graphical object.

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 the **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 SQL **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](#)

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.

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

Example

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))

Commit Table MGRSfile
Close Table MGRSfile
```

See Also**MGRSToPoint() function**

Print statement**Purpose**

Prints a prompt or a status message in the Message window.

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 (Helvetica, 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 ("Helv", 1, 10, BLUE)    ' Helvetica 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"
```

Note: 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, 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 **Write #** instead of **Print #**.

See Also

Line Input statement, Open File statement, Write # statement

PrintWin statement**Purpose**

Prints an existing window.

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** token 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()** and **WindowInfo()** functions for more information about obtaining window identifiers.

If you include the **Interactive** keyword, MapInfo Professional displays the Print dialog. If you omit the **Interactive** keyword, MapInfo Professional prints the window automatically, without displaying the dialog.

Example1

```
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.

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

Returns

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	Return Value
PRISMMAP_INFO_SCALE	Float result representing the PrismMaps scale factor.
PRISMMAP_INFO_BACKGROUND	Integer result representing the background color, see the RGB function.
PRISMMAP_INFO_LIGHT_X	Float result representing the X coordinate of the light in the scene.
PRISMMAP_INFO_LIGHT_Y	Float result representing the Y coordinate of the Light in the scene.
PRISMMAP_INFO_LIGHT_Z	Float result representing the Z coordinate of the Light in the scene.
PRISMMAP_INFO_LIGHT_COLOR	Integer result representing the Light color, see the RGB function.
PRISMMAP_INFO_CAMERA_X	Float result representing the X coordinate of the Camera in the scene.

Attribute	Return Value
PRISMMAP_INFO_CAMERA_Y	Float result representing the Y coordinate of the Camera in the scene.
PRISMMAP_INFO_CAMERA_Z	Float result representing the Z coordinate of the Camera in the scene.
PRISMMAP_INFO_CAMERA_FOCAL_X	Float result representing the X coordinate of the Cameras FocalPoint in the scene.
PRISMMAP_INFO_CAMERA_FOCAL_Y	Float result representing the Y coordinate of the Cameras FocalPoint in the scene.
PRISMMAP_INFO_CAMERA_FOCAL_Z	Float result representing the Z coordinate of the Camera's FocalPoint in the scene.
PRISMMAP_INFO_CAMERA_VU_1	Float result representing the first value of the ViewUp Unit Normal Vector.
PRISMMAP_INFO_CAMERA_VU_2	Float result representing the second value of the ViewUp Unit Normal Vector.
PRISMMAP_INFO_CAMERA_VU_3	Float result representing the third value of the ViewUp Unit Normal Vector.
PRISMMAP_INFO_CAMERA_VPN_1	Float result representing the first value of the View Plane Unit Normal Vector.
PRISMMAP_INFO_CAMERA_VPN_2	Float result representing the second value of the ViewPlane Unit Normal Vector.
PRISMMAP_INFO_CAMERA_VPN_3	Float result representing the third value of the ViewPlane Unit Normal Vector.
PRISMMAP_INFO_CAMERA_CLIP_NEAR	Float result representing the cameras near clipping plane.
PRISMMAP_INFO_CAMERA_CLIP_FAR	Float result representing the cameras far clipping plane.
PRISMMAP_INFO_INFOTIP_EXPR	String for Infotip. not previously documented.

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.

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 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

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 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).

The *status_message* parameter is a String value, such as "Processing data...", which is displayed in the dialog.

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 - 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, 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. Alternately, if the *handler* procedure assigns a value other than negative one to the ProgressBar variable:

```
ProgressBar = 50
```

then MapBasic re-displays the dialog'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.

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.

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.

Syntax

```
ProportionOverlap(object1, object2)  
object1 is the bottom object (not text or points)  
object2 is the top object (not text or points)
```

Return Value

A Float value equal to **Area(Overlap(*object1*,*object2*)) / Area(*object1*)**.

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.

Note: If the **Open File** statement specified a sequential access mode (OUTPUT or APPEND), use **Print #** or **Write #** 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.

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 **Rnd()**, 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 **Rnd()** 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

ReadControlValue() function

Purpose

Reads the current status of a control in the active dialog.

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. A **ReadControlValue()** function call is only valid while there is an active dialog; thus, you may only call the **ReadControlValue()** function from within a dialog 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.

Note: A dialog control does not have a unique ID unless you include an **ID** clause in the **Dialog** statement's **Control** clause. Some types of dialog controls have no readable values (for example, static text labels).

The chart 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 Parent 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
SymbolPicker	Symbol value
MultiListBox	<p>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.</p> <p>The first call to ReadControlValue will return 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.</p>

Error Conditions

ERR_FCN_ARG_RANGE error generated if an argument is outside of the valid range

ERR_INVALID_READ_CONTROL error generated if the ReadControlValue() function is called when no dialog is active.

Example

The following example creates a dialog 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 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** 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

Builds a MapInfo Professional table from a spreadsheet, database, text file, raster, or grid image.

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(...)]
  Type "XLS" [ Titles ] [ Range range_name ] [Interactive] |
  Type "Access" Table table_name [Password pwd] [CharSet char_set] }
Type ODBC
  Connection { Handle ConnectionNumber | ConnectionString }
  Toolkit toolkitname
  Cache { On | OFF }
  Table SQLQuery
  [Versioned {Off | On}]
  [Workspace WorkspaceName]
  [ParentWorkspace ParentWorkspaceName]
Type "GRID" | Type "RASTER"
  [ ControlPoints ( MapX1 , MapY1) ( RasterX1 , RasterY1),
    ( MapX2 , MapY2) ( RasterX2, RasterY2),
    ( MapX3 , MapY3) ( RasterX3, RasterY3)
  [, ... ]
  ]
  [ CoordSys ... ]
Type "SHAPEFILE" [ Charset char_set ] Coordsys...
  [ PersistentCache { On |Off } ]
  [Symbol...] [ Linestyle Pen(...)]
  [ Regionstyle Pen(...) Brush(...)] [ Interactive ]
  [ Into destination_file ]
```

source_file is a string that specifies 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 db.

char_set is the name of a character set; see the separate **CharSet** discussion.

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").

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.

ConnectionNumber is an integer value that identifies an existing connection to an ODBC database.

ConnectionString is a string used to connect to a database server. See the **Server Connect** function.

toolkitname is "ODBC" or "ORAINET."

SQLQuery is the SQL query used to define the MapInfo table.

ControlPoints are optional, but can be specified if the type is Grid or Raster. If the ControlPoints token 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.

For WMS and Shapefiles, the **CoordSys** clause is mandatory. The compiler will indicate an error if it is missing. For other Types, the **CoordSys** clause is optional, but it can be specified for the Grid or Raster Types. If **CoordSys** is specified, it will override and replace any coordinate system associated with the image. This is useful when registering a raster image that has an associated World file.

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 will be deleted after closing a table and will be generated each time the table is opened.

Symbol (...) clause specifies the symbol style to be used for a point object type created from a shapefile

Linestyle Pen (...) clause specifies the line style to be used for a line object type created from a shapefile

Regionstyle Pen (...) Brush(...) clause specifies the line style and fill style to be used for a region object type created from a shapefile

The **Interactive** keyword is optional, but can be specified if the type XLS, Grid, or Raster. If the **Interactive** keyword is specified for type Grid or Raster, the user will be prompted for any missing control point or projection information. If the Interactive keyword is not specified, a .TAB file will be generated without user input and will be created as though the user had selected "Display" when opening a raster image from the **File > Open** dialog.

Note: If the **Interactive** keyword is specified for type XLS, it instructs the interface to display the Set Field Properties window when importing Excel files.

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.

Versioned indicates if the table to be opened is an version-enabled (ON) table or not (OFF).

WorkspaceName is the name of the current workspace in which the table will be operated. The name is case sensitive.

ParentWorkspaceName is the name of parent workspace of the current workspace.

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.

Note: 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, XLS, Raster, Access, or Grid. The other information is necessary for preparing certain types of tables. If the type of file being registered is a grid, the coordsys string will be read from the grid file and a MapInfo .TAB file will be created. If a raster file is being registered, the .TAB file that is generated will be the same as if the user selected "Display" when opening a raster image from the File> Open dialog.

If the type of file being registered is a grid, the coordsys string will be read from the grid file and a MapInfo .TAB file will be 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 the **CharSet** clause discussion 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.

Note: Do not alter the counter column in any way. It must be exclusively maintained automatically by MapInfo.

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.

This 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 will 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 dialogue will be 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 SQL Select menu item or MapBasic) is done against 2 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 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.

Note: INTERACTIVE is not a valid parameter to use when registering SHP files.

Example1

```
Register Table "c:\mapinfo\data\rpt23.dbf"
Type "DBF"
Into "Report23"
```

```
Open Table "c:\mapinfo\data\Report23"
```

Example2

```
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 informix.can_caps"
CONNECTION
    DSN=ius_adak;UID=informix;PWD=informix;DATABASE=sw;HOST=adak;
    SERVER=adak_tli;SERVICE=sqlexec;PROTOCOL=onsoctcp;"
Into
    "D:\MI\odbc_cancaps.TAB"
Open Table "D:\MI\odbc_cancaps.TAB" Interactive
Map From odbc_cancaps
```

Example3

Registering a completely georeferenced raster image (the raster handler can return at least 3 control points and a projection)

```
Register Table "GeoRef.tif" type "raster" into "GeoRef.TAB"
```

Example4

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"
```

Example5

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"
```

Example6

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
```

Example7

The following example creates a tab file and then opens the tab file.

```
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
```

See Also

[Open Table statement](#), [Create Table statement](#), [Server Create Workspace statement](#)

Relief Shade statement

Purpose

Adds relief shade information to an open grid table.

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 the *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.

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.

Note: MapInfo 3.0 Symbols refers to the symbol set that came with MapInfo 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, 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 following 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 **CommandInfo()** 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. 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 function call:

```
CommandInfo (CMD_INFO_MSG)
```

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.

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.

Note: 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 **CommandInfo()** 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

The **Remove Cartographic Frame** statement allows you to remove cartographic frames from an existing cartographic legend created with the **Create Cartographic Legend** statement.

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()** and **WindowId()** functions.

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.

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.

Syntax

```
Remove Map [ Window window_id ]
           Layer map_layer [ , map_layer . . . ] [ Interactive ]
```

window_id is the Integer window identifier of a Map window; to obtain a window identifier, call **FrontWindow()** or **WindowID()**

map_layer specifies which map layer(s) to remove; see examples below

Description

The **Remove Map** statement removes one or more layers from a Map window. If no *window_id* is provided, the statement affects the topmost Map window.

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 <i>map_layer</i> parameter, the top map layer (other than the Cosmetic layer) is removed. Specify "1, 2" to remove the top two layers.
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 the 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 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.

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.

Syntax

```
Rename Table table As newtablespect
```

table is the name of an open table

newtablespect is the new name (and, optionally, path) for the table

Description

The **Rename Table** statement assigns a new name to an open table.

The *newtablespect* parameter specifies the table's new name. If *newtablespect* 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).

Note: **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 (**Commit** 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.

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.

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.

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.

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 the Revert command displays a dialog box.

Note: 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
    Commit 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.

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

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 will be 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.

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

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 will be 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.

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.

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.

Syntax

```
Run Application file
```

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

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.

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() call 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.

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'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 (i.e. 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 this...

```
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 Help > MapInfo Professional Tutorial on the Web command.

```
Run Menu Command M_HELP_MAPINFO_WWW_TUTORIAL
```

MapInfo's Preferences dialog is a special case. The Preferences dialog contains several buttons, each of which displays a sub-dialog. You can use Run Menu Command to invoke individual sub-dialogs. For example, the following statement displays the Map Window Preferences sub-dialog: Run Menu Command M_EDIT_PREFERENCES_MAP.

You can access invert selection using the following MapBasic command:

```
Run Menu Command M_QUERY_INVERTSELECT.
```

In version 6.0 and later, 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

Run Program statement

Purpose

Runs an executable program.

Syntax

```
Run Program program_spec
```

program_spec is a command string; this string 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.

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 statements allows you to save the current workspace as an XML-based MWS file for use with MapXtreme 2004 applications. These MWS files can be shared across platforms in ways that workspaces cannot.

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 2004 applications. When saving a workspace to MWS format, only the map windows and legends are saved. All other windows are discarded as MapXtreme 2004 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 2004 installation or with an application developed

using MapXtreme 2004. 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 2004.

Note: 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 MapXtreme2004, 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.

What is Saved in the MWS

The following information is included in the MWS workspace file:

- Tab files' name and alias
- Coordsys 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
- Label and label edit information
- Individual value themes
- Dot density themes
- Graduated symbol themes
- Bar themes
- Range themes
- Pie themes
- Grid themes as MapXtreme 2004 grid layers with a style override
- Themes and label expressions based upon a single attribute column.

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
- Any theme that is generated from a complex expression

Note: A complex expression includes any operator or multiple referenced tables.

- Any queries
- Export options
- Line direction arrows
- Whether object nodes are drawn or not
- Hot links for labels and objects

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.

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 **FrontWindow()** or **WindowID()**

filespec is a String representing the name of the file to create

filetype is a String representing a file format:

- "BMP" specifies Bitmap format;
- "WMF" specifies Windows Metafile format;
- "JPEG" specifies JPEG format;
- "JP2" specifies JPEG 2000 format
- "PNG" specifies Portable Network Graphics format;
- "TIFF" specifies TIFF format;
- "TIFFCMYK" specifies TIFF CMYK format
- "TIFFG4" specifies TIFFG4 format
- TIFFLZW" specifies TIFFLZW format
- "GIF" specifies GIF format
- "PSD" specifies Photoshop 3.0 format;
- "EMF" specifies Windows Enhanced Metafile format.

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. 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.

Specifying a Copyright Notice

To include a copyright notice on the bottom of the image, use the optional **Copyright** clause. See example below. To eliminate the default notice, specify a **Copyright** clause with an empty string ("").

Error Codes

Error number 408 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, MapInfo Corp."
```

See Also

Export statement

Save Workspace statement

Purpose

Creates a workspace file representing the current MapInfo Professional session.

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.

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()**.

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:

<i>attribute code</i>	SearchInfo() returns:
SEARCH_INFO_TABLE	String value: the name of the table containing this object. If an object is from a Cosmetic layer, this string has the form "CosmeticN" (where N is a number, 1 or larger).
SEARCH_INFO_ROW	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 error 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 **SearchPoint()**; if the user uses the rectangle tool, the program calls **SearchRect()**. 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
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 + ".coll1"
        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.

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 **SearchInfo()** 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 **CommandInfo()** to determine the x/y coordinates.

Example

For a code example, see **SearchInfo()**.

See Also

SearchInfo() function, **SearchRect() function**

SearchRect() function

Purpose

Searches for map objects within a rectangular area.

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.

Note: 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()** 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 ButtonPad** statement to create a custom ToolButton. Use DM_CUSTOM_RECT as the button's draw mode. Within the button's handler procedure, call **CommandInfo()** to determine the x/y coordinates.

Example

For a code example, see **SearchInfo()**.

See Also

SearchInfo() function, **SearchPoint() function**

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 *file* 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 error 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()** with one of the following codes:

<i>attribute code</i>	CommandInfo(attribute) returns:
CMD_INFO_SELTYPE	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	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	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 **Note**, **Print**, or **Dialog**.

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.

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.

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, an 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**.

Note: 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 QUERYn (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.

Note: 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.

Note: 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.

Note: 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
<i>Avg(column)</i>	Returns the average value of the specified column.
<i>Count(*)</i>	Returns the number of rows in the group. Specify * (asterisk) instead of column name.
<i>Max(column)</i>	Returns the largest value of the specified column for all rows in the group.
<i>Min(column)</i>	Returns the smallest value of the specified column for all rows in the group.
<i>Sum(column)</i>	Returns the sum of the column values for all rows in the group.
<i>WtAvg(column , weight_column)</i>	Returns the average of the <i>column</i> values, weighted. See below.

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:
<i>objectA</i> Contains <i>objectB</i>	first object contains the centroid of second object
<i>objectA</i> Contains Part <i>objectB</i>	first object contains part of second object
<i>objectA</i> Contains Entire <i>objectB</i>	first object contains all of second object
<i>objectA</i> Within <i>objectB</i>	first object's centroid is within the second object
<i>objectA</i> Partly Within <i>objectB</i>	part of the first object is within the second object
<i>objectA</i> Entirely Within <i>objectB</i>	the first object is entirely inside the second object
<i>objectA</i> Intersects <i>objectB</i>	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.

Note: 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.

<i>attribute setting</i>	SelectionInfo() Return Value
SEL_INFO_TABLENAME	String: The name of the table the selection was based on. Returns an empty string if no data currently selected.
SEL_INFO_SELNAME	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	Integer: The number of selected rows. Returns zero if no data currently selected.

Note: 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 error 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.

Syntax

```
Server ConnectionNumber Begin Transaction
```

ConnectionNumber is an integer value that identifies the specific connection.

Description

The **Server Begin Transaction** command 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 call to **Server Commit** is issued. Use the **Server Rollback** command 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.

Syntax

```
Server StatementNumber Bind Column n To Variable, StatusVariable
```

StatementNumber is an integer value that identifies information about an 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** command 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** operation 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	Condition
SRV_NULL_DATA	Returned when the column has no data for that row.
SRV_TRUNCATED_DATA	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_Eof(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.

Syntax

```
Server StatementNumber Close
```

StatementNumber is an integer value that identifies information about an SQL statement.

Description

The **Server Close** command 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 **Server Close** immediately after **Server_Execute** 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.

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	Server_ColumnInfo() returns:
SRV_COL_INFO_NAME	String result, the name identifying the column
SRV_COL_INFO_TYPE	Integer result, a code indicating the column type: <ul style="list-style-type: none"> • SRV_COL_TYPE_NONE • SRV_COL_TYPE_CHAR • SRV_COL_TYPE_DECIMAL • SRV_COL_TYPE_INTEGER • SRV_COL_TYPE_SMALLINT • SRV_COL_TYPE_DATE • SRV_COL_TYPE_LOGICAL • SRV_COL_TYPE_FLOAT • SRV_COL_TYPE_FIXED_LEN_STRING • SRV_COL_TYPE_BIN_STRING See Server Fetch for how MapInfo Professional interprets data types.
SRV_COL_INFO_SCALE	Integer result, indicating the number of digits to the right of the decimal for a SRV_COL_TYPE_DECIMAL column, or -1 for any other column type.
SRV_COL_INFO_PRECISION	Integer result, indicating the total number of digits for a SRV_COL_TYPE_DECIMAL column, or -1 for any other column type.

Attr	Server_ColumnInfo() returns:
SRV_COL_INFO_WIDTH	Integer result, indicating maximum number of characters in a column of type SRV_COL_TYPE_CHAR or SRV_COL_TYPE_FIXED_LEN_CHAR. 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_VALUE	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.
SRV_COL_INFO_STATUS	Integer result, indicating the status of the column value: SRV_NULL_DATA - Returned when the column has no data for that row. SRV_TRUNCATED_DATA - 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	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.

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** command. Then you must issue a new **Server Begin Transaction** statement following the **Server Commit** command 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.

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 **Server_NumDrivers** and **Server_DriverInfo()**.

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 nice connect dialog 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 data-base.

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.

UseCurrentSchema (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.

Oracle8i 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;USR=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 SQLDriverConnect 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, 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.

Informix 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 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 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 Informix is:

```
DSN=Informix TABLES;DB=PAYROLL
```

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. The default is 1, which makes the driver thread-safe. When using the driver with single-threaded applications, you may set this option to 0 to avoid additional processing required for ODBC thread safety standards.

CancelDetectInterval (CDI): A value in seconds that determines how often the driver checks whether a query has been canceled using SQLCancel. If the driver determines that SQLCancel has been issued, the query is canceled. This attribute determines whether long-running queries in threaded applications are canceled if the application issues a SQLCancel. If set to 0 (the initial default), queries are not canceled even if SQLCancel is issued.

For example, if CancelDetectInterval is set to 5, then for every pending request, the driver checks every five seconds to see whether the application has canceled execution of the query using SQLCancel.

Database (DB): Name of the database to which you want to connect.

DataSourceName (DSN): Identifies an Informix data source configuration in the system information. Examples include "Accounting" or "Informix-Serv1."

HostName (HOST): Name of the machine on which the Informix server resides.

LogonID (UID): Your user name as specified on the Informix server.

PortNumber (PORT): The port number of the server listener. There is no default value.

ServerName (SRVR): The name of the server running the Informix database.

TrimBlankFromIndexName (TBFIN): TrimBlankFromIndexName={0 | 1}. Specifies whether or not the leading space should be trimmed from a system-generated index name. This option is provided to address problems with applications that cannot process a leading space in index names. When set to 1 (the default), the driver trims the leading space. When set to 0, the driver does not trim the space.

Example

```
Dim hdbc As Integer
hdbc = Server_Connect("ODBC",
"DSN=Informix;SRV=IUSSrvr;USR=atsmipro;PWD=miproats")
```

See Also

[Server Disconnect statement](#)

Server_ConnectInfo() function

Purpose

Retrieves information about the active database connections.

Syntax

```
Server_ConnectInfo (ConnectionNo, Attr)
```

ConnectionNumber is the integer returned by Server_Connect 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	Server_ConnectInfo() returns:
SRV_CONNECT_INFO_DRIVER_NAME	String result, the name identifying the toolkit driver-name associated with this connection.
SRV_CONNECT_INFO_DB_NAME	String result, returning the database name.
SRV_CONNECT_INFO_SQL_USER_ID	String result, returning the name of the SQL user ID.
SRV_CONNECT_INFO_DS_NAME	String result, returning the data source name.
SRV_CONNECT_INFO_QUOTE_CHAR	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

```

Server Create Map statement**Purpose**

This function identifies the spatial information for a server table. It does not alter the table to add the spatial columns.

Syntax

```

Server ConnectionNumber Create Map
For linked_table
Type { MICODE columnname | XYINDEX columnname | SPATIALWARE }
CoordSys ...
[ MapBounds {Data|CoordSys|Values (x1, y1) (x2, y2)} ]
[ ObjectType { Point | Line | Region | ALL } ]
[ Symbol (...) ]
[ Linestyle Pen(...) ]
[ Regionstyle Pen(...) Brush(...) ]
[ Style Type style_number (0 or 1) [ 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.

x1, y1, x2, y2 define the coordinate system bounds. See the **MapBounds** clause discussion for more information.

CoordSys ... 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 be CoordSys.).

The Coordsys option 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.

The Values option lets you specify your own bounds values for the MapCatalog.

ObjectType clause specifies the type of object in the table: points, lines, regions, or all objects. If no object type clause is specified, the default is points.

Symbol (...) clause specifies the symbol style to be used for a point object type

Linestyle Pen (...) clause 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

StyleType sets per-row symbology. The **Column** token and argument must be present when the **Type** is set to 1 (one). When the *style_number* is set to zero the **Column** token 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 or Oracle SDO 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.

Note: 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	OracleSpatial
IUS_SW	SpatialWare IUS Blade
IUS_MM_SW >	MapInfo MapMarker Geocoding DataBlade for SpatialWare
IUS_MM_XY <columnname>	MapInfo MapMarker Geocoding DataBlade for XY
SPATIALWARE	SpatialWare for SQL Server
MICODE	XYINDEX

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
```

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.

Syntax

```
Server ConnectionNumber Set Map linked table...
  [ Style Type style_ number (0 or 1) [ 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. The **Column** token and argument need to be present when the **Type** is set to 1 (one). When the *style_number* is set to zero the **Column** token is ignored and the rendition columns in the MAPCATALOG are cleared.

Description

The **Column** token and argument need to be present when the **Type** is set to 1 (one). When the *style_number* is set to zero the **Column** token is ignored and the rendition columns in the MAPCATALOG are cleared.

In order to succeed, the map catalog 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:informix.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_symbology"
```

See Also

[Server_Connect\(\) function](#)

Server Create Table statement

Purpose

Creates a new table on a specified remote database.

Syntax

```
Server ConnectionNumber Create Table TableName(ColumnName ColumnType [,...])
  [KeyColumn ColumnName]
  [ObjectColumn ColumnName]
  [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.

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 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 Server Create Map 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, IUS, and Microsoft Access. However, to create a table with a spatial geometry/object column, SpatialWare/Blade is required for SQL Server and IUS, and the spatial option is required for Oracle.

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 or IUS with SpatialWare Blade:

```
dim hodbrc as integer
hodbrc = server_connect("ODBC", "dlg=1")
Server hodbrc 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 hodbrc as integer
hodbrc = server_connect("ORAINET", "SRVR=cygnus;UID=mipro;PWD=mipro")
Server hodbrc 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).

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 GARYG 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 **Server_Connect** with the remote data server.

Syntax

```
Server ConnectionNumber Disconnect
```

ConnectionNumber is an integer value that identifies the specific connection.

Description

The **Server Disconnect** function 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.

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	Server_DriverInfo() returns:
SRV_DRV_INFO_NAME	String result, the name identifying the toolkit. ODBC indicates an ODBC data source. ORAINET indicates an Oracle Spatial connection.
SRV_DRV_INFO_NAME_LIST	String result, returning all the toolkit names, separated by semicolons. i.e. ODBC, ORAINET. The <i>DriverNo</i> parameter is ignored.
SRV_DRV_DATA_SOURCE	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.

Syntax

```
Server_EOT(StatementNumber)
```

StatementNumber is the is the number of the fetch statement you are checking.

Return Value

Logical

Description

The **Server_EOT** function returns TRUE or FALSE indicating whether the previous 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 an SQL string to execute on a remote data server.

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 Fetch and Close operations, to a particular SQL statement.

You should perform a **Server Close** 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.

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.

Description

The **Server Fetch** command 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** command, 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** commands will halt and set the error code `ERR() = ERR_SRV_ESC` if the user presses Escape. This allows your MapBasic application using the Server Fetch commands 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 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**), 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.

How MapInfo Professional Interprets Data Types

Refer to the MapBasic *User Guide* information on how MapInfo Professional interprets data types.

Errors

The command “Server *n* Fetch Into *table*” will generate 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

```
' 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

```
' 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

Return the ODBC connection handle associated with the remote database connection.

Syntax

```
Server_GetodbcHConn( ConnectionNumber )
```

ConnectionNumber is the Integer returned by Server_Connect 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 statement.

Syntax

```
Server_GetodbcHStmt( StatementNumber )
```

StatementNumber is the integer returned by **Server_Execute()** that identifies the result set of the SQL statement executed.

Description

This function returns the ODBC statement handle associated with the MapBasic Server statement. 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.

Syntax 1

```
Server Link Table
  SQLQuery
Using ConnectionString
Into TableName
Toolkit Toolkitname
[ File FileSpec ]
[ ReadOnly ]
```

Syntax 2

```
Server ConnectionNumber Link Table
  SQLQuery
Into TableName
Toolkit toolkitname
[ File FileSpec ]
[ ReadOnly ]
```

ConnectionNumber is an integer value that identifies an existing connection.

SQLQuery is an 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. See the SQL Query section.

ConnectionString is a string used to connect to a database server. See the **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.

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 command will fail with linked tables. Linked tables cannot be packed. The Pack Table dialog 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.

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 obtained from the **ServerDriverInfo()** 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
```

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.

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.

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 the linked table with the remote database data. This command can only be run when no edits are pending against the linked table.

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 linked table. Not by using the MapBasic delete statement, but by erasing the files and recreating.
2. If a connection handle is stored with the TABLE structure, use it. Otherwise, reconnect using the connection string stored in the 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).

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.

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** function. You must have an open transaction initiated by **Server Begin Transaction** 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

This statement allows you to change the object styles for a mappable ODBC table. This updates the map catalog.

Syntax

```

Server ConnectionNumber Set Map linked_table
  [ ObjectType { Point | Line | Region } ]
  [ 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 regions, lines, or all objects, see Server Create Map statement for details

Symbol (...) clause specifies the symbol style to be used for a point object type.

Linestyle Pen (...) clause 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-enable or disable 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.

Syntax

```
Server ConnectionNumber Versioning
{
  ON
    [History {SRV_WM_HIST_NONE|SRV_WM_HIST_OVERWRITE|SRV_WM_HIST_NO_OVERWRITE}]
  | 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.

When version-enabling a table (ON), *History* is an optional parameter.

History clause specifies how to track modifications to *ServerTableName*, i.e., 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. 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.

When disabling a version-enabled table (OFF), *Force* is an optional parameter.

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).

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 identifies the rows to be merged into the parent workspace. The clause itself should omit the WHERE keyword.

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 following 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.

Interactive

In the event of a conflict, MapInfo displays the Conflict Resolution dialog box. 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.

Note: Due to a system limitation, this option is not available if the server is Oracle9i.

Automatic StopOnConflict

In the event of a conflict, MapInfo will stop here. (This is also the default behavior if the statement does not include an Interactive clause or an Automatic clause.)

Automatic RevertToBase

In the event of a conflict, MapInfo 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 parameter must be followed by UseParent or UseCurrent.)

Automatic UseCurrent

In the event of a conflict, MapInfo uses the child workspace values.

Automatic UseParent

In the event of a conflict, MapInfo uses the parent workspace values.

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.

Examples

The following example merge changes to USA where MI_PRINX=5 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).

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.

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 following 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.

Interactive

In the event of a conflict, MapInfo displays the Conflict Resolution dialog box. The conflicts will be resolved one by one based on user choices. After all the conflicts are resolved, the table is refreshed from its parent based on the user's choices.

Note: Due to a system limitation, this option is not available if the server is Oracle9i.

Automatic StopOnConflict

In the event of a conflict, MapInfo will stop here. (This is also the default behavior if the statement does not include an Interactive clause or an Automatic clause.)

Automatic RevertToBase

In the event of a conflict, MapInfo 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.)

Automatic UseCurrent

In the event of a conflict, MapInfo uses the child workspace values.

Automatic UseParent

In the event of a conflict, MapInfo uses the parent workspace values.

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.

Examples

The following example refreshes MIUSER by applying changes made to USA where MI_PRINX=5 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.

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	Return Value
SESSION_INFO_COORDSYS_CLAUSE	String result that indicates a session's CoordSys clause.
SESSION_INFO_DISTANCE_UNITS	String result that indicates a session's distance units.
SESSION_INFO_AREA_UNITS	String result that indicates a session's area units.
SESSION_INFO_PAPER_UNITS	String result that indicates a session's paper units.

Error Conditions

ERR_FCN_ARG_RANGE error generated if an argument is outside of the valid range

Example

```
Include "mapbasic.def"  
print SessionInfo(SESSION_INFO_COORDSYS_CLAUSE)
```

Set Application Window statement**Purpose**

Sets which window will be the parent of dialogs that are yet to be created.

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.

Note: 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.

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. 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
"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.

Syntax

```
Set Browse
  [ Window window_id ]
  [ Grid { On | Off } ]
  [ Row row_num ]
  [ Column column_num ]
```

window_id is the Integer window identifier of a Browser 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 **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.

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 Cartographic Legend statement

Purpose

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. (To change the size, position or title of the legend window, use the **Set Window** statement.)

Syntax

```
Set Cartographic Legend
  [ Window legend_window_id ]
  Redraw { On | Off }
```


or

```
Set Cartographic Legend
[ Window legend_window_id ]
[ Refresh ]
[ Portrait | Landscape ]
  [Columns number_of_columns | 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()** and **WindowId()** functions.

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.

number_of-columns specifies the width of the legend.

number_of_lines specifies the height of the legend.

Description

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 **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 Command Info statement

Purpose

Stores values in memory; other procedures can call **CommandInfo()** to retrieve the values.

Syntax

```
Set Command Info attribute To new_value
```

attribute is one of the codes used by **CommandInfo()**, such as CMD_INFO_ROWID.

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, 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 **CommandInfo()** will return 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 **CommandInfo()**. 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 **CommandInfo()**:

```
Set Command Info CMD_INFO_ROWID To 1
```

See Also

CommandInfo() function, Set Handler statement

Set CoordSys statement

Purpose

Sets the coordinate system used by MapBasic.

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 **CentroidX()** and **ObjectNodeX()**) 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 the discussion of the **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"
```

Note: 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.

Syntax

```
Set Date Window { nYear | Off }
```

nYear a SmallInt from 0 to 99 that specifies the year equal to and above which is the current century (20th) and below which is the next century (21st).

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** command from the MapBasic window will change the behavior of input, but will not update the System Preference that is saved when MapInfo Pro 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 Application interface.

When the **Set Date Window** command is run from within a MapBasic program (also as Run Command) only the program's local context will be updated with the new setting. The session and preference settings will remain unchanged. The program's local context will be 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.

MBX's compiled before v5.5 will still convert 2-digit years to the current century (5.0 and earlier behavior). To get the new behavior, they must be recompiled with MapBasic v5.5 or later.

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 Digitizer statement

Purpose

Establishes the coordinates of a paper map on a digitizing tablet; also turns Digitizer Mode on or off.

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

tablet# parameters specify tablet right-left positions corresponding to the *mapx#* values

tablety# parameters specify tablet up-down positions corresponding to the *mapy#* values

names are optional labels 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 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 the **CoordSys** clause discussion.

Note: 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 Create Object.

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, **Create Object** 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	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

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.

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.

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, **Set Map**), 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 (such as Windows or System 7), 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.

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.

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. Note that the Retry/Cancel dialog is not trappable; the dialog 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 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.

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 **Str\$()**, 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 **Str\$()** 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.
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 **SystemInfo()**. 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 **Str\$()**, you force the **Str\$()** function to follow U.S. formatting (Month/Day/Year), 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.

Syntax 1 (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

The **Line** clause specifies a line style

The **Brush** clause specifies a fill style, and the **Pen** clause specifies the fill's border

The **Symbol** clause specifies a symbol style

The **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.

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.

Graph commands in workspaces or programs that were created prior to version 5.5 will still create a 5.0 graph window. When a 5.0 graph window is active in MapInfo Professional 5.5 and later, the 5.0 graph menu will be also be active, so the user can modify the graph using the 5.0 editing dialogs. The Create Graph wizard will always created a 5.5 graph window.

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 ("Helv",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 ("Helv",0,8,0)
  Pen (1,2,117440512)
  Title "Salesperson" Font ("Helv",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 ("Helv",0,8,0)
  Pen (1,2,117440512)
  Title "Order amounts ($)" Font ("Helv",0,8,0)
,
' the above title ("Order amounts...") appears
' along the grapher's y-axis
,
,
' The next set graph customizes graphical styles
' for series 2. This dictates what color bars will
' appear to represent the orders_rcvd column data.
' Also controls what description will appear in the
' legend
,
' Since this is a bar graph, the Brush is the style
' of prime importance; if this was a line graph,
' the Line and Symbol clauses would be important).
,
Set Graph Series 2
  Brush (8,255,16777215)
  Line (1,2,0,255) Symbol (32,255,12)
  Title "Orders Received ($)"
,
  '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 ("Helv",0,10,0) 'set the title font
  Subtitle "(by salesperson)"
  Font ("Helv",0,8,0) 'set subtitle font
  Range font ("Helv",2,8,0) 'set the font used for
                           'range descriptions

```

See Also

Graph statement, Set Window statement

Set Handler statement

Purpose

Enables or disables the automatic calling of system handler procedures, such as SelChangedHandler.

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.

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.

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 } ]
```

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

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 (i.e. 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, the **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
```

Note: 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.

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 | Fontsize } ]
  [ 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 **Shade** or **Set 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's 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 8.0. 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 a new way to specify legend label order of ascending or descending as well as new custom order. However, the original Ascending { On | Off } clause is still available for backwards compatibility. You can use either the new Order clause, or the old 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 Map statement

Purpose

Modifies an existing Map window.

Syntax

The main part of a **Set Map** statement has the following syntax:

```
Set Map
[ Window window_id ]
[ Center ( longitude, latitude ) [ Smart Redraw ] ]
[ Clipping { Object clipper | Off | On } | Using
  [Display {All | PolyObj} | Overlay] } ] ]
[ Zoom { zoom_distance [ Units dist_unit ] | Entire [ Layer layer_id ] } ]
[ Preserve { Scale | Zoom } ]
[ Display { Scale | Position | Zoom } ]
[ Order layer_id, layer_id [ , layer_id ... ] ]
[ Pan pan_distance [ Units dist_unit ]
  { North | South | East | West } [ Smart Redraw ] ]
[ CoordSys... ]
[ Area Units area_unit ]
[ Distance Units dist_unit ]
[ Distance Type { Spherical | Cartesian } ]
[ XY Units xy_unit ]
[Display Decimal {On | Off | [ Display Grid ]
[ Scale screen_dist [ Units dist_unit ] For map_dist [ Units dist_unit ] ]
[ Redraw { On | Off } ]
[Inflect num_inflections [ by percent ]
  Contrast contrast_value ]
  [ Brightness brightness_value ]
  [ {ALPHA <alpha_value> }|{TRANSLUCENCY <translucency_percent>}}
    [ TRANSPARENCY {OFF|ON }
      [ COLOR <transparent_color_value> ]
    [ GrayScale { On | Off } ]
[Round rounding_factor ]
[ Relief { On | Off } ]
[ Move Nodes { value | Default } ]
[
  LAYERCLAUSE
  LAYERCLAUSE . . .
```

window_id is the Integer window identifier of a Map window.

longitude, *latitude* is the new center point of the map.

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.

zoom_distance is a numeric expression dictating how wide an area to display.

layer_id 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.

pan_distance is a distance to pan the map.

The **CoordSys** clause specifies a coordinate system; for details, see separate discussion.

area_unit is a string representing the name of an area unit (for example, "sq mi" for square miles, "sq km" for square kilometers; see Set Area Units for a list of unit names).

distance is either be **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 **Coordsys** of the Map Window is NonEarth, then the calculations will be performed using **Cartesian** methods regardless of the option chosen, and if the **Coordsys** of the Map Window is Latitude/Longitude, then calculations will be performed using **Spherical** methods regardless of the option chosen.

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.

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** command

Move Node 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), Move Nodes Default can be specified.

screen_dist and *map_dist* specify a map scale (for example, *screen_dist* = 1 inch, *map_dist* = 1 mile).

num_inflections is a numeric expression, specifying the number of color:value inflection pairs.

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 raster or grid image. Values range between 0-100. 0 is completely opaque. 100 is completely transparent.

Either **ALPHA** or **TRANSLUCENCY** should be specified, not both since they are different ways of specifying the same thing. If multiple tokens are specified, the last value will be used.

The **ALPHA** and **TRANSLUCENCY** tokens are new for Set Map. They apply to raster and grid layers.

The **CONTRAST**, **BRIGHTNESS** and **GRAYSCALE** tokens are supported for raster layers. They apply to both raster and grid layers.

The **TRANSPARENCY** and **COLOR** tokens are new for Set Map and only apply to raster layers.

The **TRANSPARENCY** token determines whether and individual color is transparent for a raster layer.

The **COLOR** token specifies which color is transparent in a raster layer.

color:expr is a color expression of, part of a color:value inflection pair.

In the syntax above, **LAYERCLAUSE** is a shorthand notation, not a MapBasic keyword. Each **LAYERCLAUSE** has the syntax described below.

```
[ Layer layer_id
  [ Activate { [ Using launch_expr ] | [ On { [ Labels ] | [ Objects ] } ] |
    [ Relative Path { On | Off } ] } ]
  [ 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 } ]
LABELCLAUSE
  [ Display { Off | Graphic | Global } ]
  [ Global Line ... ]
  [ Global Pen ... ]
  [ Global Brush ... ]
  [ Global Symbol ... ]
  [ Global Font ... ]
]
```

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.

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.

launch_expr is an expression that will resolve to the name of the file to launch when the object is activated.

The **Using** clause sets the filename expression and the **On** clause sets the activation mode. At least one of these clauses is required. If the **Using** clause is included, then *filename_expr* is required.

If the **On** clause is included, then one or both of the Labels and Objects clauses are required. If just Labels is included, then activation occurs on labels only. If just **Objects** is included, then activation occurs on objects only. If both keywords are included, then activation occurs on both labels and objects. By default activation occurs on labels only.

Use **Relative Path On** when the files to be launched are stored in a location relative to the table in which the links are defined. Use **Relative Path Off** when the HotLinks are URLs or full path files descriptions; this is the default.

The **Line** clause specifies a line style used to draw lines and polylines; identical to a **Pen** clause, except that the keyword **Pen** is replaced by the keyword **Line**.

The **Pen** clause specifies a line style used to draw frames around filled objects.

The **Brush** clause specifies a fill style.

The **Symbol** clause specifies a symbol style.

The **Font** clause specifies a text style.

In the syntax above, **LABELCLAUSE** is a shorthand notation, not a MapBasic keyword.

Each **LABELCLAUSE** has the syntax described below:

```
[ Label [ Line { Simple | Arrow | None } ]
[ Position [ Center ] [ Above | Below ] [ Left | Right ] ]
[ Font ... ] [ Pen ... ]
[ With label_expr ]
[ Parallel { On | Off } ]
[ 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 ]
[ 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 ]
  [ Offset offset_amount ]
  [ Callout ( callout_x, callout_y ) ] }
]
```

label_expr is the expression to use for creating 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 representing the name of a distance unit (for example, "mi" for miles, "m" for meters; see Set Distance Units for a list of available unit names).

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 50 (representing a distance in points), causing the label to be offset from its anchor point.

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 **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.

Note: **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 MapBasic text edit window.

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.

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.

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
```

Changing the Behavior of the Entire Map

The following clauses affect how the Map window behaves.

Area Units

Specifies the unit of measure used to display area calculations. For a list of area unit names, see the **Set Area Units** statement.

```
Set Map Area Units "sq km"
```

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, you can enable or disable clipping by specifying **Clipping On** or **Clipping Off**.

```
Set Map Clipping Object obj_variable_name
```

Set Map Statement for Clip Region

Sets a clipping object for the Map window; corresponds to MapInfo Professional's Map, Set Clip Region command. Once a clipping region is set, you can enable or disable clipping by specifying Clipping On or Clipping Off.

There are three modes that can be used for Clipping. Using the Overlay mode will use the MapInfo Professional Overlay (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. (This method is used in ALL versions prior to MapInfo Professional 6.0.)

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. This mode approximates the Overlay functionality found in MapInfo Professional prior to version 6.0.

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
```

CoordSys... clause

Assigns the Map window a different coordinate system and projection. For details on the syntax of a **CoordSys** clause, see the separate **CoordSys** discussion.

The MapBasic CoordSys must be set explicitly with a **Set CoordSys** statement and can be retrieved with the **SessionInfo()** function

Note: 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.

In versions prior to 7.x, the following example would set both the map's Coordsys to this UTM system as well as set the underlying MapBasic CoordSys to this system:

```
Set Map XY Units "m" CoordSys Earth Projection 8, 33, "m", -55.5, 0, 0.9999, 304800, 0
```

In versions 7.x and later, this example would only alter the map's Coordsys and Units; the MapBasic Coordsys is unaffected.

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  
Distance Units
```

Specifies the unit of measure used to display distance calculations (for example, in the Ruler Tool window). For a list of area unit names, see the **Set Distance Units** statement.

```
Set Map Distance Units "km"
```

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 Change View dialog box (Map menu > Change View).

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).

Note: 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.

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"
```

Changing the Order of Layers

The **Order** clause resets the order in which map layers are drawn. Each *layer_num* is a number identifying a map layer, 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).

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
```

Changing the Behavior of Individual Layers

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.

Set Map Clause for HotLinks

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 new 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 it's label, will result in the default browser being started with the site *http://www.boston.com*. You can associated 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.

About Relative Path Settings

The **Relative Path** setting allows you to 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".

Note: 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.

Changing the Appearance of Individual Layers

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.

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
```

Display

This clause 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 **Global Line** clause 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.

The **Global Pen** clause specifies the style used to display the borders of filled objects.

The **Global Brush** clause specifies the style used to display filled objects.

The **Global Symbol** clause specifies the style used to display point objects.

The **Global Font** clause specifies the font used to display text objects.

The following statement displays layer 1 in its default style:

```
Set Map
Layer 1 Display Graphic
```

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)
```

Changing Labeling Options for Individual Layers

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
```

Font

Specifies the font used in labels.

Pen

Specifies 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. In version 7.0, this feature was expanded to all object types. For versions previous to 7.0, only linear objects were affected.

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
```

Object

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.

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.

Setting Move Duplicate Nodes

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
  Layer 1
    Editable On
  Layer 2
    Selectable Off
    Display Global
    Global Brush (2, 255, 65535)
```

See Also

Add Map statement, LayerInfo() function, Map statement, MapperInfo() function, Remove Map statement, Set Window statement

Set Map3D statement

Purpose

Change the settings of an existing 3DMap window.

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) ] ]
[ 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 ranges from 0-360 degrees and rotates the maps around the center point of the grid. The vertical angle in the dialog 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 can't 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

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, ViewPlane Normal and Clipping Range (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 unitless grids (i.e. 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 be specified when you create the 3Dmap. You cannot change them later with Set Map3D or the Properties dialog.

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).

Restrictions

This statement is only available under Microsoft Windows.

Syntax

```
Set Next Document
{ Parent HWND | Style style_flag | Parent HWND Style style_flag }
```

HWND is an Integer Windows 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 Chapter 13 of 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**, **Browse**, **Graph**, **Layout**, or **Create Legend**.

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.

<i>style_flag</i> code	Effect on the next document window:
WIN_STYLE_CHILD	Next window is created as a child window. (Code has a value of 1.)
WIN_STYLE_POPUP	Next window is created as a popup window with a half-height title bar caption. (Code has a value of 3.)
WIN_STYLE_POPUP_FULLCAPTION	Next window is created as a popup window, but with a full-height title bar caption. (Code has a value of 2.)
WIN_STYLE_STANDARD	This code resets the style flag to its default value. (Code has a value of 0.) 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.

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.

Note: 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.

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, **Set Window**) 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 following **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 PrismMap statement

Purpose

Change the settings of an existing Prism Map window.

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) ] ]
[ Light [ Position (x,y,z) | Color lightcolor ] ]
[ Scale grid_scale ]
[ Background backgroundcolor ]
[ Label With infotips_expr ]
[ Refresh ]
```

window_id is a window identifier 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 ranges from 0-360 degrees and rotates the maps around the center point of the grid. The vertical angle in the dialog 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, ViewPlane Normal and Clipping Range (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

Changes the settings of an already created Prism Map.

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 change 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 dialogs.

Syntax

```
Set ProgressBars { On | Off }
```

Description

Some MapBasic statements, such as the **Create Object As Buffer** statement, automatically display a progress-bar dialog (a “percent complete” dialog showing a horizontal bar and a Cancel button). To suppress progress-bar dialogs, use the **Set ProgressBars Off** statement. By suppressing these dialogs, you guarantee that the user will not interrupt the operation by clicking the Cancel button. To resume displaying progress-bar dialogs, 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 dialogs caused by the MBX file. Actions taken by the user can still cause progress bars to display. Also, **Run Menu Command** statements can still cause progress bars to display, because **Run Menu Command** simulates the user selecting a menu command.

To disable progress-bar dialogs that are caused by user actions or **Run Menu Command** statements, 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 statement **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

Set Redistricter statement

Purpose

Changes the characteristics of a districts table during a redistricting session.

Syntax 1

```
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 2

```
Set Redistricter districts_table
  Order { "Alpha" | "MRU" | "Unordered" }
```

districts_table is the name of the districts table (for example, Districts)

district_name is a String: the name of an existing district

new_district_name is a String: new district name, used when adding a district or renaming an existing district

Pen... is a Pen clause, for example, **Pen MakePen (width , pattern , color)**

Brush... is a Brush clause, for example, **Brush MakeBrush (pattern , forecolor , backcolor)**

Symbol... is a Symbol clause, 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.

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()** and **ConvertToPline()** functions. 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.

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.

Syntax

```
Set Style
{ Brush ... |
  Font ... |
  Pen ... |
  BorderPen |
  LinePen |
  Symbol ... }
```

Brush clause specifies a fill style

Font clause specifies a text style

Pen clause specifies a line style

Symbol clause specifies 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. For more information about Pen, Brush, Symbol, and Font parameters, see the discussions of the Pen, Brush, Font, and Symbol clauses.

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("Helv", 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 statement

Purpose

Configures various settings of an open table.

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 } ]
```

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** 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** statement or a **Rollback** statement. By issuing a **Commit** or **Rollback** statement, you reset the file so that other network users can access it.

Setting Read-Only Mode

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.

Setting Undo Mode

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

MapInfo Professional versions 4.0 and later support a table type known as 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.

Preventing the User from Accessing Tables

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.

Note: 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.
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 button (in the Layer Control dialog box) is disabled for this table.
UserDisplayMap Off	If this table appears in a Map window, the Display check box (in the Layer Control dialog box) is disabled for this table.

Example

The following statement prevents the World table from appearing in the Close Table dialog.

```
Set Table World UserClose Off
```

See Also

TableInfo() function

Set Target statement

Purpose

Sets or clears the map editing target object(s).

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

Change the size, position, title, or status of a window, and control the printer, paper size and margins used by 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 ... ]
[ 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 } ]
      [ Transparency
        [ Raster { Device | ROP } ]
        [ Vector { Device | ROP } ] ]
      [ Margins
        [ Left d1 ]
        [ Right d2 ]
        [ Top d3 ]
        [ Bottom d4 ]
        Units <units> ] } ]
[ Export { Default |
[ Border { On | Off } ]
[ TrueColor { On | Off } ]
[ Dither { Halftone | ErrorDiffusion } ]
[ Transparency
  [ Raster { Device | ROP } ]
  [ Vector { Device | ROP } ] ] ]
} ]
[ 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 top of MapInfo Professional's workspace to the top edge of the window

y states the desired distance from the left of MapInfo Professional's workspace to the left 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.

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

Printer will specify window-specific overrides for printing.

Export will specify 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 papersize 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 papersize. 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 papersize information stored in a workspace correctly. In that case, papersize 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 new keyword and 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. Previous to this release, MapInfo Professional always drew directly to the device. The new 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 RasterInternal Removed for version 7.0; however, if present, the token 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. For example, vector data that is under transparent pixels will not be rasterized. In metafiles, the ROP method will not draw any data in the areas of the raster pixels and allow the background

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.

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()** and **WindowId()** functions. 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.
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 statements. The following example disables the Close command of the active window:

```
Set Window FrontWindow( ) SysMenuClose Off
```

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*.

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 **WindowInfo()**.

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.

Note: 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*.

Note: 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 or Create Cartographic Legend MapBasic statements.

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.

Attribute codes, WIN_INFO_PRINTER_NAME, WIN_INFO_PRINTER_ORIENT or WIN_INFO_PRINTER_COPIES, are also returned with WindowInfo() function.

Example

```
Set Window frontwindow( )
Printer Name "\\Discovery\HP 2500CP"
Orientation Portrait
Copies 10
```

Note: 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.

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
```

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).

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.

Syntax 1 (shading by ranges of values)

```
Shade      [ Window window_id ]
  { layer_id | layer_name }
  With expr
  [ Ignore value_to_ignore ]
  Ranges
  [ Apply { Color | Size | All } ]
  [ Use { Color | Size | All } [ Pen... ] [ 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... ] ]
```

Syntax 2 (shading by individual values)

```
Shade      [ Window window_id ]
  { layer_id | layer_name }
  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... ] ]
```

Syntax 3 (dot density)

```
Shade      [ Window window_id ]
  { layer_id | layer_name }
  With expr
  Density dot_value {Circle | Square}
  Width dot_size
  [ Color color ]
```

Note: For backwards compatibility, the older MapBasic syntax (version 7.5 or earlier) is still supported.

Syntax 4 (graduated symbols)

```
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" } ]
```

Syntax 5 (pie charts)

```

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 . . . ] ]

```

Syntax 6 (bar charts)

```

Shade [ Window window_id ]
    { layer_id | layer_name | Selection }
    With expr [ , expr . . . ]
    Bar [ Normalized ] | Stacked Bar [ Fixed ] }f
    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 . . . ] ]

```

symbol_size is the point size to use for symbols having the appropriate *value*

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

const is a constant numeric expression or a constant string expression

The **Pen** clause specifies a line style (for example, Pen(*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, Brush(*pattern*, *forecolor*, *backcolor*))

The **Symbol** clause specifies a symbol style (for example, Symbol(*shape*, *color*, *size*))

minimum is the minimum numeric value for a range

maximum is the maximum numeric value for a range

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.

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 **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.

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 **Ranges** keyword results in a shaded map where each object falls into a range of values; the **Values** keyword creates a map where each unique value has its own display style; the **Density** keyword creates a dot density map; the **Graduated** keyword results in a graduated symbols map; and the **Pie** and **Bar** keywords specify thematically constructed charts.

Ranges of Values

For the specific syntax of a **Ranges** map, see [Syntax 1 \(shading by ranges of values\) on page 527](#).

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** and **Create Styles** statements. For an example of using arrays in **Shade** statements, see **Create Ranges**.

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()** clauses to control the region fill styles. If you are shading points, specify **Symbol()** clauses. If you are shading linear objects (lines, polylines, or arcs) specify **Line()** clauses, *not* **Pen()** clauses; 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 : 2000000 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 : 2000000 Symbol(35,BLUE,27)
```

Note: 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** clauses (9, 18, 27 point); MapInfo Professional ignores the other display attributes (i.e. 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** clauses. 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).

Individual Values

For the specific syntax of an Individual Values map, see [Syntax 2 \(shading by individual values\) on page 527](#).

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 for each 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.

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 Maps

For the specific syntax of a Dot Density map, see [Syntax 3 \(dot density\) on page 527](#).

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
```

Note: For backwards compatibility, the older MapBasic syntax (version 7.5 or earlier) is still supported.

Graduated Symbols Maps

For the specific syntax of a Graduated map, see [Syntax 4 \(graduated symbols\) on page 527](#).

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.

Note: 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 Chart Maps

For the specific syntax of a Pie map, see [Syntax 5 \(pie charts\) on page 528](#).

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 Chart Maps

For the specific syntax of a Bar map, see [Syntax 6 \(bar charts\) on page 528](#).

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.

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 minus 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.

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.

Syntax

```
SphericalArea( expr, unit_name )
```

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 the **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 can't 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 SQL **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.

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.

Returns

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.

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 the **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.

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 can't 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.

Syntax

```
SphericalObjectDistance(object1, object2, unit_name)
```

object1 and *object2* are object expressions.

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

Returns

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.

Syntax

```
SphericalObjectLen( 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 can't 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 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 = 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.

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, and

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 or Perimeter, 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 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

```
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 Y offset values using a spherical DistanceType.

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, and

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 or Perimeter, is the unit for the 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 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

```
SphericalOffsetXY(Rect, 92, -22, "mi")
```

See Also

SphericalOffset() function

SphericalPerimeter() function**Purpose**

Returns the perimeter of a graphical object.

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 **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 **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 the **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 can't 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 SQL **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.

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.

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 yes, 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 yes, the user will be able to select the editable layer from the status bar.

Stop statement

Purpose

Suspends a running MapBasic application, for debugging purposes.

Syntax

```
Stop
```

Restrictions

You cannot issue a **Stop** statement from within a user-defined function or within a dialog's handler procedure; therefore you cannot issue a **Stop** statement to debug a **Dialog** statement while the dialog 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.

Str\$() function

Purpose

Returns a string representing an expression (for example, a printout of a number).

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 the *expression* is a negative number, the first character in the returned string is the minus sign (-). If the *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 the *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 the *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.

Note: Passing an uninitialized Object variable to the **Str\$()** function generates an error.

If the *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 the *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.

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.

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.

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.

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.

Note: To avoid the entire issue of how the user's computer is set up, call **NumberToDate()** 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 **Set Format** 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

StyleAttr() function

Purpose

Returns one attribute of a Pen, Brush, Font, or Symbol style.

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.

<i>attribute setting</i>	StyleAttr() returns:
BRUSH_PATTERN	Integer, indicating the Brush style's pattern.
BRUSH_FORECOLOR	Integer, indicating the Brush style's foreground color, as an RGB value.
BRUSH_BACKCOLOR	Integer, indicating the Brush style's background color as an RGB value, or -1 if the brush has a transparent background.
FONT_NAME	String, indicating the Font name.
FONT_STYLE	Integer value, indicating the Font style (0 = Plain, 1 = Bold, etc.); see Font clause for details.
FONT_POINTSIZE	Integer indicating the Font size, in points. Note: 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	Integer value representing the RGB color of the Font foreground.
FONT_BACKCOLOR	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	Integer, indicating the Pen style's line width, in pixels or points.
PEN_PATTERN	Integer, indicating the Pen style's pattern.
PEN_COLOR	Integer, indicating the Pen style's RGB color value.
PEN_INTERLEAVED	Logical, TRUE if line style is interleaved.
PEN_INDEX	Integer, representing the pen index number from the pen pattern.
SYMBOL_KIND	Integer, indicating the type of symbol: 2 for TrueType symbols; 3 for bitmap file symbols.
SYMBOL_CODE	Integer, indicating the Symbol style's shape code. Applies to TrueType symbols.
SYMBOL_COLOR	Integer, indicating the Symbol style's color as an RGB value.

<i>attribute setting</i>	StyleAttr() returns:
SYMBOL_POINTSIZE	Integer from 1 to 48, indicating the Symbol's size, in points.
SYMBOL_FONT_NAME	String, indicating the name of the font used by a TrueType symbol.
SYMBOL_FONT_STYLE	Integer, indicating the style attributes of a TrueType symbol (0 = plain, 1 = Bold, etc.). See Symbol clause for a listing of possible values.
SYMBOL_ANGLE	Float number, indicating the rotation angle of a TrueType symbol.
SYMBOL_CUSTOM_NAME	String, indicating the file name used by a bitmap file symbol.
SYMBOL_CUSTOM_STYLE	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 error 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

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. Each parameter is defined with the following syntax:

```
[ ByVal ] parameter As var_type
```

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.

Syntax 1 (MapInfo 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. 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 the **RGB()** function.

size is an Integer point size, from 1 to 48.

Syntax 2 (TrueType 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 the **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.

Syntax 3 (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 the **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 4

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*).

Description

Note: 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).

MapInfo 3.0 Symbol Syntax

The following table lists the standard symbol shapes that are available when you use syntax 1.

31		41	☆	51	✱	61	🛡
32	■	42	△	52	✈	62	🛡
33	◆	43	▽	53	🚩	63	🚩
34	●	44	▣	54	🚩	64	✂
35	★	45	▲	55	🏠	65	🏠
36	▲	46	●	56	+	66	🚧
37	▽	47	➡	57	⋈	67	🚧
38	□	48	↙	58	⚓		
39	◇	49	+	59	⦿		
40	○	50	×	60	🏠		

TrueType Font Syntax

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:

<i>fontstyle</i> 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 Symbol (Bitmap File) Syntax

When you specify a custom symbol, the *customstyle* argument controls background and color settings, as described in the following table.

<i>customstyle</i> value	Symbol Style
0	Both the Show Background setting and the Apply Color setting are off; the symbol appears in its default state. 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.

To specify both Show Background and Apply Color, use a value of 3.

Example

The following example shows how a **Set Map** statement can incorporate a **Symbol** clause. The **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)
```

See Also

MakeCustomSymbol() function, **MakeFontSymbol() function**, **MakeSymbol() function**, **StyleAttr() function**

SystemInfo() function

Purpose

Returns information about the operating system or software version.

Syntax

```
SystemInfo( attribute )
```

attribute is an Integer code indicating which system attribute to query

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

<i>attribute</i> code	SystemInfo() Return Value
SYS_INFO_APPLICATIONWND	Integer, representing the Windows HWND specified by the Set Application Window statement (or zero if no such HWND has been set).
SYS_INFO_APPVERSION	Integer value: the version number with which the application was compiled, multiplied by 100.
SYS_INFO_CHARSET	String value: the name of the native character set.
SYS_INFO_COPYPROTECTED	Logical value: TRUE means the user is running a copy-protected version of MapInfo Professional.
SYS_INFO_DATE_FORMAT	String: "US" or "Local" depending on the date formatting in effect; for details, see Set Format .
SYS_INFO_DDESTATUS	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	Logical value: TRUE if a digitizer is installed, along with a compatible driver.
SYS_INFO_DIG_MODE	Logical value: TRUE if Digitizer Mode is on.
SYS_INFO_MAPINFOWND	Integer, representing a Windows HWND of the MapInfo Professional frame window, or zero on non-Windows platforms.
SYS_INFO_MDICLIENTWND	Integer, representing a Windows HWND of the MapInfo Professional MDICLIENT window, or 0 on non-Windows platforms.
SYS_INFO_MIPLATFORM	Integer value, indicating the type of MapInfo Professional software that is running.

<i>attribute code</i>	SystemInfo() Return Value
SYS_INFO_MIVERSION	Integer value, indicating the version of MapInfo Professional that is currently running, multiplied by 100.
SYS_INFO_NUMBER_FORMAT	String: "9,999.9" or "Local" depending on the number formatting in effect; for details, see Set Format .
SYS_INFO_PLATFORM	Integer value, indicating the hardware platform on which the application is running. The return value will be PLATFORM_WIN.
SYS_INFO_PRODUCTLEVEL	Integer value, indicating the product level of the version of MapInfo Professional that is running (for example, 200 for MapInfo Professional).
SYS_INFO_RUNTIME	Logical value: TRUE if invoked within a run-time version of MapInfo Professional, FALSE otherwise.
SYS_INFO_APPIDISPATCH (value=17)	Integer, representing the IDispatch OLE Automation pointer for the MapInfo Application.

Error Conditions

ERR_FCN_ARG_RANGE error 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.

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

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.

<i>attribute code</i>	TableInfo() returns
TAB_INFO_COORDSYS_CLAUSE	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_MINX, TAB_INFO_COORDSYS_MINY, TAB_INFO_COORDSYS_MAXX, TAB_INFO_COORDSYS_MAXY	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.
TAB_INFO_COORDSYS_NAME	String result, representing the name of the CoordSys 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 CoordSys is not found in MAPINFOW.PRJ.
TAB_INFO_EDITED	Logical result; TRUE if table has unsaved edits.
TAB_INFO_FASTEDIT	Logical result; TRUE if the table has FastEdit mode turned on, FALSE otherwise. (See Set Table for information on FastEdit mode.)
TAB_INFO_MAPPABLE	Logical result; TRUE if the table is mappable.
TAB_INFO_MAPPABLE_TABLE	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	Float results, indicating the minimum and maximum x- and y-coordinates of all objects in the table.
TAB_INFO_NAME	String result, indicating the name of the table.
TAB_INFO_NCOLS	SmallInt, indicating the number of columns.
TAB_INFO_NREFS	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	SmallInt, indicating the number of rows.

<i>attribute code</i>	TableInfo() returns
TAB_INFO_NUM	SmallInt result, indicating the number of the table.
TAB_INFO_READONLY	Logical result; TRUE if the table is read-only.
TAB_INFO_SEAMLESS	Logical result; TRUE if seamless behavior is on for this table.
TAB_INFO_TABFILE	String result, representing the table's full directory path. Returns an empty string if the table is a query table.
TAB_INFO_TEMP	Logical result; TRUE if the table is temporary (for example, QUERY1).
TAB_INFO_TYPE	SmallInt result, indicating the type of table. The returned value will match one of these values: TAB_TYPE_BASE (if a normal or seamless table) TAB_TYPE_RESULT (if results of a query) TAB_TYPE_IMAGE (if table is a raster image) TAB_TYPE_VIEW (if table is actually a view; for example, StreetInfo tables are actually views) TAB_TYPE_LINKED (if this table is linked). TAB_TYPE_WMS (if table is from a Web Map Service) TAB_TYPE_WFS (if table is from a Web Feature Service)
TAB_INFO_UNDO	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	Logical result: FALSE if a Set Table statement has set the UserBrowse option to Off.
TAB_INFO_USERCLOSE	Logical result: FALSE if a Set Table statement has set the UserClose option to Off.
TAB_INFO_USERDISPLAYMAP	Logical result: FALSE if a Set Table statement has set the UserDisplayMap option to Off.
TAB_INFO_USEREDITABLE	Logical result: FALSE if a Set Table statement has set the UserEdit option to Off.
TAB_INFO_USERMAP	Logical result: FALSE if a Set Table statement has set the UserMap option to Off.
TAB_INFO_USERREMOVEMAP	Logical result: FALSE if a Set Table statement has set the UserRemoveMap option to Off.
TAB_INFO_SUPPORT_MZ	Logical result: TRUE if table supports M and Z-values.
TAB_INFO_Z_UNIT_SET	Logical result: TRUE is unit is set for Z-values.
TAB_INFO_Z_UNIT	String result: indicates distance units used for Z-values. Return empty string if units are not specified.

Error Conditions

ERR_TABLE_NOT_FOUND error generated if the specified table was not available

ERR_FCN_ARG_RANGE error 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)
```

See Also

Open Table statement

Tan() function**Purpose**

Returns the tangent of a number.

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. 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 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.

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.

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.

Note: **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.

Syntax

```
TextSize( window_id , text_obj )
```

window_id is the Integer window identifier of a Map or Layout window. Call **FrontWindow()** or **WindowID()** to obtain window identifiers.

text_obj is a text object.

Note: 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.

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.

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** and **Create ButtonPad** 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, **CommandInfo()** returns the row and column where the user clicked. If the user clicked in a Map, **CommandInfo()** returns the map coordinates of the location where the user clicked; these coordinates are in MapBasic's current coordinate system (see the **Set CoordSys** statement).

If the user clicked in a Layout window, **CommandInfo()** 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 the **Set Paper Units** statement).

By calling **CommandInfo()**, 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.

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** 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 error generated if the TriggerControl() function is called when no dialog 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.

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** and **Global** statements.

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 (i.e. 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.

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** 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.

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.

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.

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")
```

For a listing of MapInfo Professional's standard distance unit names (for example, "km"), see the **Set Distance Units** statement. For a listing of area unit names (for example, "sq km"), see the **Set Area Units** statement. For a listing of paper unit names (for example, "in" for inches on a page layout), see the **Set Paper Units** statement.

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.

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" )
```

For a listing of MapInfo Professional’s standard distance unit names (for example, “km”), see the **Set Distance Units** statement. For a listing of area unit names (for example, “sq km”), see the **Set Area Units** statement. For a listing of paper unit names (for example, “in” for inches on a page layout), see the **Set Paper Units** statement.

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

Use the **Unlink** statement to unlink a table which was downloaded and linked from a remote database with the **Server Link Table** statement.

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.

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.

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

Val () function

Purpose

Returns the numeric value represented by a string.

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.)

Note: 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.

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

Syntax

```
WFS Refresh Table alias
```

alias is the an alias for an open registered WFS table.

Example

The following example refreshes the local table named watershed.

```
WFS Refresh Table watershed
```

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 (i.e. for the sake of those programmers who prefer the **While...Wend** syntax over the **Do...Loop** 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 **CommandInfo()** 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 **End Program**.

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.

Note: 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.

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, **Set Map**) take window identifiers as parameters.

The following table lists the various ways that you can specify the *window_num* parameter:

Value of <i>window_num</i>	Result
Positive Smallint value (1, 2, ... <i>n</i>)	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 <i>n</i> is the number of open document windows; call NumWindows() to determine <i>n</i> .
Negative Smallint value (-1,-2, ...- <i>m</i>)	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 <i>m</i> is the total number of windows owned by MapInfo Professional; call NumAllWindows() to determine <i>m</i> . Using this syntax, you could call WindowID() within a loop to build a list of the ID numbers of all open windows.
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 error generated if the *window_num* parameter is invalid

See Also

FrontWindow() function, **NumWindows() function**

WindowInfo() function**Purpose**

Returns information about a window.

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 <i>window_spec</i>	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, ... <i>n</i>)	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 <i>n</i> is the number of open document windows; call NumWindows() to determine <i>n</i> .
Negative Smallint value (-1,-2, ...- <i>m</i>)	The function queries a window, which may be a document window or a floating window such as the Info window. Note that <i>m</i> is the total number of windows owned by MapInfo Professional; call NumAllWindows() to determine <i>m</i> . 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:

<i>attribute</i> code	WindowInfo(<i>attribute</i>) 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, see Set Window.
WIN_INFO_CLONEWINDOW (15)	String value: a string of MapBasic statements that can be used in a Run Command statement to duplicate a window. See Run Command .
WIN_INFO_HEIGHT (5)	Float value: window height (in paper units).

<i>attribute code</i>	WindowInfo(attribute) returns:
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_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 if minimized, WIN_STATE_MAXIMIZED 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.
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. 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 WindowID() . This is useful if you pass zero as the <i>window_spec</i> .
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 in that the results include Open Table statements, etc.

<i>attribute code</i>	WindowInfo(<i>attribute</i>) returns:
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, <code>\\DISCOVERYHP4_DEVEL</code>)
WIN_INFO_PRINTER_ORIENT (22)	Returns WIN_PRINTER_PORTRAIT or WIN_PRINTER_LANDSCAPE
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.
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 folder) 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)	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 is disabled or if the printer color depth is 256 colors or less.

<i>attribute code</i>	WindowInfo(<i>attribute</i>) returns:
WIN_INFO_PRINTER_METHOD (32)	String value: possible return values are DEVICE and EMF.
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.
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.

If you specify WIN_INFO_TYPE as the *attribute*, **WindowInfo()** returns one of these values:

Window type	Window description
WIN_MAPPER	Map window
WIN_BROWSER	Browse window
WIN_LAYOUT	Layout window
WIN_GRAPH	Graph window
WIN_HELP	The Help window
WIN_MAPBASIC	The MapBasic window
WIN_MESSAGE	The Message window (used with the MapBasic Print statement)
WIN_RULER	The Ruler window (displays the distances measured by the Ruler tool)
WIN_INFO	The Info window (displays data when the user clicks with the Info tool)
WIN_LEGEND	The Theme Legend window
WIN_STATISTICS	The Statistics window
WIN_MAPINFO	The MapInfo application window
WIN_BUTTONPAD	A ButtonPad window
WIN_TOOLBAR	The Toolbar window
WIN_CART_LEGEND	The Cartographic Legend window
WIN_3DMAP	The 3D Map 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. Similarly, you can obtain the name of a Layout window’s temporary table (for example, “Layout1”) by calling **WindowInfo()** with the WIN_INFO_TABLE attribute.

Error Conditions

ERR_BAD_WINDOW error generated if the *window_id* parameter is invalid

ERR_FCN_ARG_RANGE error 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.

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 SelChangedHandler.

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.

Syntax

```
Year( date_expr )
```

date_expr is a date expression

Return Value

SmallInt

Description

If **Set Date Window** is off then the year also depends on your system clock. If your system clock says that today is 2/2/1998, then the year function returns, 1993, if your system clock says that today is 1/4/2004, then the year function returns 2093. MapInfo Professional uses the current century.

Examples

The following example shows how you can use the **Year()** function to extract only the year component of a particular date value.

```
If Year( CurDate( ) ) = 1994 Then  
    ' ...then it is still 1994...  
End If
```

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**

Character Code Table

A

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	í	193	Á	225	á
34	"	66	B	98	b	130	■	162	ç	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

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.

Sections in this Appendix:

♦ Numeric Operators	589
♦ Comparison Operators	590
♦ Logical Operators	590
♦ Geographical Operators	591
♦ Automatic Type Conversions	592

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
>	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 token *Integer* represents an integer value, which can be an Integer variable, a SmallInt variable, or an Integer constant. The token *Number* represents a numeric expression which is not necessarily an integer.

Operator	Combination of Operands	Result
+	<i>Date + Number</i>	Date
	<i>Number + Date</i>	Date
	<i>Integer + Integer</i>	Integer
	<i>Number + Number</i>	Float
	<i>Other + Other</i>	String
-	<i>Date - Number</i>	Date
	<i>Date - Date</i>	Integer
	<i>Integer - Integer</i>	Integer
	<i>Number - Number</i>	Float
*	<i>Integer * Integer</i>	Integer
	<i>Number * Number</i>	Float
/	<i>Number / Number</i>	Float
\	<i>Number \ Number</i>	Integer
MOD	<i>Number MOD Number</i>	Integer
^	<i>Number ^ Number</i>	Float

C

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:

```
'=====
' MapInfo version 8.0 - 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
'     ButtonPadInfo() defines
'     ColumnInfo() and column type defines
'     CommandInfo() and task switch defines
'     DateWindow() defines
'     FileAttr() and file access mode defines
'     GetFolderPath$() defines
'     IntersectNodes() parameters
'     LabelInfo() defines
'     LayerInfo(), display mode, label property, layer type, hotlink
defines
'     LegendInfo() and legend orientation defines
'     LegendFrameInfo() and frame type defines
'     LegendStyleInfo() defines
'     LocateFile$() 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

'-----
' 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

```

```

'=====
' 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

'=====
' 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

```

```

=====
' 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

'=====
' 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

```

```

'=====
' 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

'-----
' 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

'-----
' 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

'-----
' Hotlink activation modes, from LayerInfo() for LAYER_INFO_HOTLINK_MODE
'-----
Define HOTLINK_MODE_LABEL              1
Define HOTLINK_MODE_OBJ                2
Define HOTLINK_MODE_BOTH               3

'=====
' 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

'=====
' 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

```

```

'=====
' 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

```

```

'-----
' 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

'-----
' 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_ROUNDRAIDUS 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

'=====
' 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

```

```

'-----
' 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_USERREMOVEDMAP  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

```

```

'-----
' 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

'=====
' 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_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

```

```

'-----
' 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_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

```

```

'=====
' 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

'=====
' end of MAPBASIC.DEF

```


Index

Symbols

- ! (exclamation point) in menus** 160
- (backslash)**
 - in menus 159
 - integer division 589
- & (ampersand)**
 - dialog hotkeys 210
 - finding street intersections 247–249
 - hexadecimal numbers 572
 - menu hotkeys 161
 - string concatenation 589
- ((open parenthesis) in menus** 160
- * (asterisk)**
 - fixed length strings 217
 - multiplication 589
- + (plus)** 589
- / (slash)**
 - division 589
 - in menus 159, 161
- < (less than) character**
 - in menus 159
- ^ (caret)**
 - exponentiation 589
 - show/hide menu text 160

Numerics

- 3D Maps**
 - changing window settings 506–507
 - creating 157–158
 - prism maps 173–174
 - reading window settings 309–311

A

- Absolute value**
 - Abs() function 47
- Accelerator keys**
 - in dialogs 210
 - in menus 161
- Access databases**
 - connection string attributes 441
- Acos() function** 47
- Add Cartographic Frame statement** 48–50
- Add Column statement** 50–54
- Add Map statement** 54–56

Adding

- animation layers 55–56
- buttons 57–61
- columns to a table 50–54, 79–80
- map layers 54–56
- menu items 67–69
- nodes 76, 354, 368
- Addresses, finding** 247–249
- Aggregate functions** 429–430
- Alias variables** 217
- All-caps text** 252–253
- Alter Button statement** 56
- Alter ButtonPad statement** 57–61
- Alter Cartographic Frame statement** 61
- Alter Control statement** 62–64
- Alter MapInfoDialog statement** 64–66
- Alter Menu Bar statement** 70–71
- Alter Menu Item statement** 71–73
- Alter Menu statement** 67–69
- Alter Object statement** 73–78
- Alter Table statement** 79–80
- Animation layers**
 - adding 55–56
 - removing 401
- ApplicationDirectory\$() function** 80
- Arc objects**
 - creating 137
 - determining length of 338
 - modifying 73–76
 - querying the pen style 334–337
 - storing in a new row 278–279
 - storing in an existing row 570
- Area**
 - spherical calculation 537
- Area units of measure** 470
- Area() function** 81
- AreaOverlap() function** 82
- Arithmetic functions. See Math functions**
- Array variables**
 - declaring 218
 - determining size of array 567
 - resizing 390–391
- Asc() function** 82
- ASCII files**
 - exporting 235–237
 - using as tables 392–396
 - See also* File input/output
- Asin() function** 83

Ask() function 83

Assigning local storage
 Server Bind Column 436–437

Atn() function 84

AutoCAD files
 importing 273–277

AutoLabel statement 85

Automatic type conversions 592

Automation
 handling button event 139
 handling menu event 161

Autoscroll feature
 list of affected draw modes 60
 reading current setting 578
 turning on or off 524
 WinChangedHandler 575

Avg() aggregate function 429–430

B

Background colors
 Brush clause 88–89
 Font clause 252–253
 MakeBrush() function 303
 MakeFont() function 305

Bar charts
 in graph windows 268–269
 in thematic maps 535

Beep statement 86

Beginning a transaction
 Server Begin Transaction 435

Binary file i/o
 closing files 107
 opening files 363
 reading data 263–264
 writing data 387

BMP files, creating 417–419

Bold text 252–253

Boundaries. *See* Region objects

Bounding rectangle 316

Branching
 Do Case...End Case statement 221–223
 If...Then statement 270–272

Breakpoints (debugging) 544

Browse statement 86–87

Browser windows
 closing 109
 determining the name of the table 579
 modifying 472, 519–525
 opening 86–87
 restricting which columns appear 403

Brush styles
 Brush clause defined 87–89
 creating 303
 modifying an object's style 74
 querying an object's style 334–337
 querying parts of 549–551
 reading current style 190
 setting current style 515–516

Brush variables 217

BrushPicker controls 127

Buffer regions
 Buffer() function 89
 CartesianBuffer() function 94
 Create Object 168
 Create Object statement 166–169

Button controls (in dialogs) 120

ButtonPadInfo() function 90

ButtonPads
 adding/removing a button 57–61, 563–564
 creating a new pad 138–140
 docked vs. floating 59, 139
 drawing modes 60–61
 enabling/disabling a button 56
 querying current settings 90
 resetting to defaults 140
 responding to user action 115
 selecting/deselecting a button 56
 setting which button is active 412
 showing/hiding a pad 57–61

Byte order in file i/o 364

C

Call statement 91–93

Calling clause 139, 161

Callout lines
 map labels 502
 text objects 188

CancelButton controls 120

Capitalization
 lower case 293
 mixed case 386
 upper case 567

CartesianArea() function 93

CartesianBuffer() function 94

CartesianDistance() function 95

CartesianObjectLen() function 97

CartesianOffset() function 98

CartesianOffsetXY() function 99

CartesianPerimeter() function 100

Cartographic legends
 adding a frame 48–50
 changing a frame 61
 controlling settings 472
 creating 141–144
 removing a frame 400

Case statement. *See* Do Case...End Case statement

Case, converting
 LCase\$() function 293
 Proper\$() function 386
 UCase\$() function 567

Centroid
 setting a region's 77

Centroid() function 101

Centroids, displaying 501

CentroidX() function 101

CentroidY() function 102

Character codes

- character sets **103–105**
- converting codes to strings **106**
- converting strings to codes **82**
- listing **586**

CharSet clause 103–105**Checkable menu items, creating 160****CheckBox controls 121****Checking**

- dialog check boxes (custom) **62–64**
- dialog check boxes (standard) **64–66**
- menu items **71–73**

ChooseProjection\$() function 105**Chr\$() function 106****Circle objects**

- creating **144–145, 148**
- determining area of **81**
- determining perimeter of **375**
- modifying **73–76**
- querying the pen or brush style **334–337**
- storing in a new row **278–279**
- storing in an existing row **570**

Cleaning objects 343**Clicking and dragging. *See* ButtonPads****Clipping a map 498****Cloning a map 412****Close All statement 107****Close File statement 107****Close Table statement 108****Close Window statement 109****Closing processing**

- Server Close **437**

Collection objects

- combining **111**
- creating **146**
- resetting objects within collection **77–78**

Color values

- RGB **404**
- See also* Brush, Font, Pen, Symbol

ColumnInfo() function 110–111**Columns in a table**

- adding **50–54, 79–80**
- deleting **79–80**
- determining column information **110–111, 330**
- dynamic columns **54**
- indexing **153**

Combining objects

- Combine() function **111**
- Create Object statement **166–169**
- Objects Clean statement **343**
- Objects Combine statement **344–346**

CommandInfo() function 112–116**Commit Table statement 116–119****Comparing strings 297, 547–548****Comparison operators 590****Compiler directives**

- Define statement **205**
- Include statement **277**

Concatenating strings

- & operator **589**
- + operator **589**

Conditional execution

- Do Case...End Case statement **221–223**
- If...Then statement **270–272**

Conflict Resolution dialog 118**Connect option**

- DLG=1 **440**

Connect_string

- defined **440**

Connecting to a data source

- Server_Connect **440–447**

Connection number

- returning **440–447**

Continue statement 120**Control DocumentWindow clause 122****Control key**

- detecting control-click **115**
- entering line feeds in EditText boxes **124**
- selecting multiple list items **125–127**

Control panels

- date formatting **480–481**
- number formatting **480–481**

Controls in dialogs

- BrushPicker **127**
- Button **120**
- CancelButton **120**
- CheckBox **121**
- EditText **124**
- FontPicker **127**
- GroupBox **125**
- ListBox **125–127**
- MultiListBox **125–127**
- OKButton **120**
- PenPicker **127**
- RadioGroup **129–130**
- StaticText **130**
- SymbolPicker **127**

Converting

- character codes to strings **106**
- numbers to dates **329**
- numbers to strings **545**
- objects to polylines **131**
- objects to regions **131**
- strings to character codes **82**
- strings to dates **548–549**
- strings to numbers **572**
- text to lower case **293**
- text to mixed case **386**
- text to upper case **567**
- two-digit input into four-digit years **475**

ConvertToPline() function 131**ConvertToRegion() function 131****Convex hull objects**

- Create Object **168**

ConvexHull() function 132

- CoordSys clause**
 - changing a table's CoordSys **116–119**
 - changing a window's projection **499**
 - querying a table's CoordSys **558**
 - querying a window's CoordSys **312**
 - setting the current MapBasic CoordSys **474**
 - specifying a coordinate system **133–136**
 - Copying**
 - an object offset from source **353–354**
 - object offset **359–360**
 - Copying a projection**
 - from a table **133–136**
 - from a window **133–136**
 - Copying files **415****
 - copying object**
 - offset by distance **540**
 - offset by XY values **541**
 - copying objects**
 - offset by specified distance **98**
 - offset by XY values **99**
 - Copying tables **116–119****
 - Copyright notice**
 - creating **418**
 - Cos() function **136****
 - Cosmetic layer, accessing as a table **579****
 - Count() aggregate function **429–430****
 - Create Arc statement **137****
 - Create ButtonPad statement **138–140****
 - Create ButtonPads As Default statement **140****
 - Create Cartographic Legend statement **141–144****
 - Create Collection statement **146****
 - Create Cutter statement **147****
 - Create Ellipse statement **148****
 - Create Frame statement **149–150****
 - Create Grid statement **150–152****
 - Create Index statement **153****
 - Create Legend statement **153****
 - Create Line statement **155****
 - Create Map statement **156****
 - Create Map3D statement **157–158****
 - Create Menu Bar statement **163–164****
 - Create Menu statement **158–163****
 - Create Multipoint statement **165–166****
 - Create Object statement **166–169****
 - Create Pline statement **170****
 - Create Point statement **172****
 - Create PrismMap statement **173–174****
 - Create Ranges statement **175–177****
 - Create Rect statement **177****
 - Create Redistricter statement **178****
 - Create Region statement **179–181****
 - Create Report From Table statement **181****
 - Create RoundRect statement **182****
 - Create Styles statement **183–184****
 - Create Table statement **184****
 - Create Text statement **188****
 - CreateCircle() function **144–145****
 - CreateLine() function **154****
 - CreatePoint() function **171–172****
 - CreateText() function **186–187****
 - Cross-reference. *See* cross-reference in online Help**
 - Crystal Reports**
 - creating **181**
 - loading **364**
 - CurDate() function **189****
 - CurrentBorder Pen() **189****
 - CurrentBrush() function **190****
 - CurrentFont() function **190****
 - CurrentLinePen() function **191****
 - CurrentPen() function **191****
 - CurrentSymbol() function **192****
 - Cursor coordinates, displaying **499****
 - Cursor shapes **59****
 - Cursor, position in table**
 - end-of-table condition **229**
 - positioning the row cursor **241–242**
 - Custom symbols**
 - Reload Symbols statement **397**
 - syntax **553–555**
 - Cutter objects**
 - creating **147**
- ## D
- Data aggregation**
 - combining objects **343–346**
 - filling a column with data aggregated from another table **52–54**
 - grouping rows **429–430**
 - Data disaggregation**
 - erasing part of an object **349–352**
 - splitting objects **357–358**
 - Data structures **566****
 - Databases, using as tables **392–396****
 - Date functions**
 - converting numbers to dates **329**
 - converting strings to dates **480–481, 548–549**
 - current date **189**
 - date window setting **193**
 - extracting day-of-month **193**
 - extracting day-of-week **573**
 - extracting the month **324**
 - extracting the year **584**
 - formatting based on locale **480–481**
 - Date variables **217****
 - DateWindow() function **193****
 - Day() function **193****
 - DBF files, exporting **235–237****
 - DDE, acting as client**
 - closing a conversation **201**
 - executing a command **194**
 - initiating a conversation **195–198**
 - reading data from the server **199–200**
 - sending data to the server **198**
 - DDE, acting as server**
 - handling execute event **398**
 - handling peek request **399**
 - retrieving execute string **114**

- DDEExecute statement** 194
 - DDEInitiate function** 195–198
 - DDEPoke statement** 198
 - DDERequest\$() function** 199–200
 - DDETerminate statement** 201
 - DDETerminateAll statement** 201
 - Debugging**
 - Continue statement 120
 - Stop statement 544
 - Decimal separators** 206, 259, 480–481
 - Decision-making**
 - Do Case...End Case statement 221–223
 - If...Then statement 270–272
 - Declare Function statement** 202–203
 - Declare Sub statement** 204–205
 - Define statement** 205
 - Definitions file** 594
 - DeformatNumber\$() function** 206
 - Delaying when user drags mouse** 479
 - Delete statement** 207
 - Deleting**
 - all objects from a table 225
 - columns from a table 79–80
 - files 282
 - nodes from an object 73
 - rows or objects 207
 - tables 226
 - Dialog Preserve statement** 214
 - Dialog Remove statement** 215
 - Dialog statement** 208–214
 - Dialogs, custom**
 - accelerator keys 210
 - creating 208–214
 - determining ID of a control 565
 - determining if user clicked OK 113
 - determining if user double-clicked 113
 - modal vs. modeless 209
 - modifying 62–64
 - preserving after user clicks OK 214
 - reading user's input 210, 388–390
 - sizes of dialogs and controls 210
 - tab order 211
 - terminating 210, 215
 - Dialogs, standard**
 - altering MapInfo dialogs 64–66
 - asking OK/Cancel question 83
 - opening a file 244–245
 - percent complete 383–385
 - saving a file 246
 - simple messages 328
 - suppressing progress bars 511
 - Digitizer setup** 476–477
 - Digitizer status** 556
 - Dim statement** 216–220
 - Directory names, extracting from a file name** 369
 - user's home directory 270
 - user's Windows directory 270
 - where application is installed 80
 - where MapInfo is installed 383
 - Disabling**
 - ButtonPad buttons 56
 - dialog controls (custom) 62–64
 - dialog controls (standard) 64–66
 - handler procedures 487
 - menu items 71–73
 - progress bar dialogs 511
 - shortcut menus 163
 - system menu's Close command 523
 - Discarding changes**
 - to a local table 406
 - to a remote server 463
 - Distance**
 - spherical calculation 538
 - Distance units of measure** 477
 - Distance() function** 220
 - DLG=1 connect option** 440
 - DLLs**
 - declaring as functions 203
 - declaring as procedures 204–205
 - Do Case...End Case statement** 221–223
 - Do...Loop statement** 223–224
 - Dockable ButtonPads**
 - docking after creation 59
 - docking at creation 139
 - querying current status 90
 - Document conventions** 34, 46
 - DOS commands, executing** 414
 - Dot density thematic maps** 532–533
 - Double byte character sets (DBCS)**
 - extracting part of a DBCS string 323
 - Double-clicking in dialogs** 113, 126
 - Dragging with the mouse**
 - time threshold 479
 - turning off autoscroll 524
 - Drawing modes** 60
 - Drawing objects.** *See* Objects, creating
 - Drawing tools, custom** 57–61
 - Drop Index statement** 224
 - Drop Map statement** 225
 - Drop Table statement** 226
 - Duplicating a map** 412
 - DXF files**
 - exporting 235–237
 - importing 273–277
 - Dynamic columns** 54
 - Dynamic Link Libraries.** *See* DLLs
- ## E
- Editable map layers** 313, 500
 - Editing an object.** *See* specific object type
 - Arc, Ellipse, Frame, Line, Point, Polyline, Rectangle, Region, Rounded Rectangle, Text
 - Edits**
 - determining if there are unsaved edits 557–560
 - discarding 406
 - saving 116–119
 - EditText controls** 124

Elapsed time 562

Ellipse objects

- Cartesian area of 93
- Cartesian perimeter of 100
- creating 144–145, 148
- determining area of 81
- determining perimeter of 375
- modifying 73–76
- querying pen or brush style 334–337
- storing in a new row 278–279
- storing in an existing row 570

Enabling

- ButtonPad buttons 56
- dialog controls 62–64
- menu items 71–73

End MapInfo statement 226

End Program statement 227

EndHandler procedure 228

Enlarging arrays 390–391

EOF() function 228

EOT() function 229, 242

Erase() function 229

Erasing

- entire objects 207
- files 282
- part of an object 229, 349–352
- tables 226

Err() function 230

Error handling

- determining error code 230
- determining error message 232
- enabling an error handler 360–362
- generating an error 231
- returning from an error handler 403

Error statement 231

Error\$() function 232

Escape key

- cancelling draw operations 58–61
- dismissing a dialog 113
- interrupting selection 425

Events, handling

- application terminated 228
- Automation method used 398
- execute string received 114, 398
- map window changed 114, 575
- MapInfo got or lost focus 114, 255
- peek request received 399
- selection changed 114
- user clicked with custom tool 115, 563–564
- user double-clicked in a dialog 113
- window closed 114, 576
- window focus changed 583
- See also* Error handling

Excel files, opening 392–396

Executing

- interpreted strings 410–412
- menu commands 412
- Run Application statement 410
- Run Program statement 414

Executing an SQL string, Server_Execute() 455

Execution speed

- animation layers 55–56
- screen updates 479
- table editing 516–518

Exit Do statement 232

Exit For statement 233

Exit Function statement 233

Exit Sub statement 234

Exiting from MapInfo 226

Exp() function 234

Expanded text 252–253

Exponentiation 234

Export statement 235–237

Extents of entire table 558

External functions 203

Extracting part of a string

- Left\$() function 293
- Mid\$() function 323
- MidByte\$() function 323
- Right\$() function 405

ExtractNodes() function 238

F

Fetch statement 241–242

Fields. *See* Columns

File input/output

- closing a file 107
- determining if file exists 243
- end-of-file condition 228
- file attributes, reading 243
- length of file 300
- opening a file 362–364
- reading current position 423
- reading data in binary mode 263–264
- reading data in random mode 263–264
- reading data in sequential mode 277, 298
- setting current position 424
- writing data in binary mode 387
- writing data in random mode 387
- writing data in sequential mode 378, 584

File names

- determining full file spec 565
- determining temporary name 560
- extracting directory from 369
- extracting from full file spec 370

File sharing conflicts 480

FileAttr() function 243

FileExists() function 243

FileOpenDlg() function 244–245

Files

- copying 415
- deleting 282
- determining if file exists 243
- importing 273–277
- length 300
- locating 299
- renaming 402

FileSaveAsDlg() function 246

Fill styles. *See* Brush styles

Filtering data 427–429

Find statement 247–249

Find Using statement 250–251

Finding

- a substring within a string 279
- an address in a map 247–249
- an intersection of two streets 247–249
- objects from map coordinates 419–423

Fix() function 251

Fixed length strings 217

Floating point variables 217

Flow control

- exiting a Do loop 232
- exiting a For loop 233
- exiting a function 233
- exiting a procedure 234
- exiting an application 227
- exiting MapInfo 226
- halting another application 561
- unconditional jump 267

Focus within a dialog 63

Focus, active window changes 583

Focus, getting or losing 114, 255

Folder names. *See* Directory names

Font styles

- creating 305
- Font clause defined 252–253
- modifying an object's style 74
- querying an object's style 334–337
- querying parts of 549–551
- reading current style 190
- setting current style 515–516

Font variables 217

FontPicker controls 127

For...Next statement 254–255

ForegroundTaskSwitchHandler procedure 255

Foreign character sets 103–105

Format() function 256–258

FormatDate\$ function 258

FormatNumber\$() function 259

Frame objects

- creating 149–150
- inserting into a layout 278–279
- modifying 73–76, 570
- querying the pen or brush style 334–337

Frames, cartographic legend

- adding a frame 48–50
- controlling settings 472
- creating 141–144
- modifying 61
- removing 400

FrontWindow() function 260

Function...End Function statement 260–262

Functions, creating

- Declare Function statement 202–203
- Exit Function statement 233
- Function...End Function statement 260–262

G

Gaps

- checking in regions 342–343
- cleaning 343
- snapping nodes 355–357

Geocoding

 247–249

Geographic calculations

- area of object 81
- area of overlap 82
- distance 220
- length of object 338
- perimeter of object 375

Geographic calculations.

See Objects, querying

Geographic objects.

See Objects

Geographic operators

 433, 591

Get statement

 263–264

GetFolderPath\$() function 264

GetMetadata\$() function 265

GetSeamlessTable() function 265

Global statement

 266

GML files

- importing 273–277

Goto statement

 267

GPS applications

 55–56

Graduated symbol thematic maps

 533

Graph statement

 268–269

Graph windows

- closing 109
- determining the name of the table 579
- modifying 482–486, 519–525
- opening 264, 268–269

Great circle distance

 220

Grid surfaces

- in thematic maps 150–152
- modifying 493–505

Grid tables

- adding relief shade information 397

Group By clause

 429–432

GroupBox controls

 125

H

Halo text

 252–253

Halting another application

 561

Handlers

- assigning to menu items 159

Hardware platform, determining

 556–557

Help messages

- button tooltips 58, 139
- status bar messages 58, 139

Help window

- closing 109
- modifying 519–525
- opening 366, 519–525

Hexadecimal numbers

 572

Hiding

- ButtonPads **57–61**
- dialog controls (custom) **62–64**
- dialog controls (standard) **64–66**
- menu bar **316**
- progress bar dialogs **511**
- screen activity **479**

Hierarchical menus

- Alter Menu statement **69**
- Create Menu statement **160**

HomeDirectory\$() function **270****HotLink tool**

- querying object attributes **116**
- Set Map clause **501**

HotLinks

- querying **288**

HWND values, querying

- SystemInfo() function **556**
- WindowInfo() function **579**

I**Iconizing MapInfo**

- Set Window statement **519–525**
- suppressing progress bars **511**

Icons for ButtonPads **59****Identifiers, defining** **205****If...Then statement** **270–272****Import statement** **273–277****Include statement** **277****Indexed columns**

- creating an index **153**
- deleting an index **224**

Individual value thematic maps **531****Infinite loops, avoiding** **487****Info tool**

- closing Info window **109**
- modifying Info window **519–525**
- opening Info window **366**
- setting to read-only **524**
- setting which data displays **524**

Informix databases

- connection string attributes **446–447**

Initializing variables **220****Input # statement** **277****Input/output. See File input/output****Insert statement** **278–279****Inserting**

- columns in a table **50–54, 79–80**
- nodes in an object **76**
- rows in a table **278–279**

InStr() function **279****Int() function** **280****Integer division** **589****Integer variables** **217****Integrated mapping**

- managing legends **153**
- reparenting dialogs **470**
- reparenting document windows **507–508**

International character sets **103–105****International formatting** **480–481****Interpreting strings as commands** **410–412****Interrupting the selection** **425****Intersection of objects**

- Create Object statement **166–169**
- Intersects operator **433**
- Objects Intersect statement **351–352**
- Overlap() function **367**

Intersection of two streets, finding **247–249****IntersectNodes() function** **281****IsPenWidthPixels() function** **282****Italic text** **252–253****J****Joining tables** **427–429****JPEG files, creating** **417–419****K****Keys**

- metadata **319–321**

Keywords **219, 429****Kill statement** **282****L****LabelFindByID() function** **283–284****LabelFindFirst() function** **284****LabelFindNext() function** **285****Labelinfo() function** **285–287****Labels**

- in dialogs **130**
- in programs **267**
- on maps **85, 283–287, 502–504**
- reading label expressions **289**

Launching other applications

- Run Application statement **410**
- Run Program statement **414**

LayerInfo() function **288–291****Layers**

- adding **54–56**
- Cosmetic **579**
- modifying settings **493–505**
- reading settings **288–291**
- removing **401**
- thematic maps **514, 527–535**

Layout statement **292****Layout windows**

- accessing as tables **579**
- closing **109**
- creating frames **149–150**
- modifying **487–489, 519–525**
- opening **292**
- specifying layout coordinates **133–136, 474**

LCase\$() function **293****Left\$() function** **293****Legend frames**

- querying attributes **294–295**
- querying styles **296**

- Legend window**
 - closing **109**
 - modifying **490–492, 519–525**
 - opening **153, 366**
 - querying **295**
- Legend, cartographic**
 - adding a frame **48–50**
 - controlling settings **472**
 - creating **141–144**
 - modifying a frame **61**
 - removing a frame **400**
- LegendFrameInfo() function** **294–295**
- LegendInfo() function** **295**
- LegendStyleInfo() function** **296**
- Len() function** **296**
- Length**
 - spherical calculation **539**
- Length of file** **300**
- Length of object** **338**
- Like() function** **297**
- Line feed character**
 - Chr\$(10) function **106**
 - used in EditText controls **124**
 - used in text objects **188**
- Line Input statement** **298**
- Line objects**
 - Cartesian length of **97**
 - creating **154–155**
 - determining length of **338**
 - modifying **73–76**
 - querying the pen style **334–337**
 - storing in a new row **278–279**
 - storing in an existing row **570**
- Line styles. See Pen styles**
- Linked tables**
 - creating **459–461**
 - determining if table is linked **559**
 - refreshing **462**
 - saving **118**
 - unlinking **570**
- ListBox controls** **125–127**
- Locale settings** **206, 259, 480–481**
- LocateFile\$()** **299**
- LOF() function** **300**
- Log() function** **300**
- Logical operators** **590**
- Logical variables** **217**
- Looping**
 - Do...Loop statement **223–224**
 - For...Next statement **254–255**
 - While...Wend statement **574**
- Lotus files, opening** **392–396**
- Lower case, converting to** **293**
- LTrim\$() function** **301**
- M**
- Main procedure** **302–303**
- MakeBrush() function** **303**
- MakeCustomSymbol() function** **304**
- MakeFont() function** **305**
- MakeFontSymbol() function** **305**
- MakePen() function** **306**
- MakeSymbol() function** **307**
- Map layers. See Layers**
- Map objects. See Objects**
- Map projections**
 - changing a table's projection **116–119**
 - changing a window's projection **499**
 - copying from a table or window **133–136**
 - querying a table's CoordSys **558**
 - querying a window's CoordSys **312**
 - setting the current MapBasic CoordSys **474**
- Map scale**
 - determining in Map windows **313**
 - displaying **498**
- Map statement** **308–309**
- Map windows**
 - adding map layers **54–56**
 - clipping **498**
 - closing **109**
 - controlling redrawing **54, 479, 500**
 - creating thematic layers **527–535**
 - duplicating **412**
 - handling window-changed event **114, 575**
 - labeling **502**
 - modifying **493–505, 519–525**
 - modifying thematic layers **514**
 - opening **308–309**
 - reading layer settings **288–291**
 - reading window settings **311–315**
 - removing map layers **401**
- Map3dInfo() function** **309–311**
- MapBasic Definitions file** **594**
 - language overview **34–46**
- MapInfo 3.0 symbols** **553–555**
- MAPINFOW.ABB file** **249**
- MapperInfo() function** **311–315**
- Maps windows, prism** **173–174**
- Math functions**
 - absolute value **47**
 - arc-cosine **47**
 - arc-sine **83**
 - arc-tangent **84**
 - area of object **81**
 - area of overlap **82**
 - converting strings to numbers **572**
 - cosine **136**
 - distance **220**
 - exponentiation **234**
 - logarithms **300**
 - maximum value **315**
 - minimum value **324**
 - rounding off a number **251, 280, 408**
 - sign **526**
 - sine **535**
 - square root **543**
 - tangent **560**

Max() aggregate function 429–430

Maximum() function 315

MBR() function 316

Memo fields 79, 116, 118

Menu Bar statement 316

Menu commands, executing 412

MenuItemInfoByHandler() function 317–318

MenuItemInfoByID() function 318

Menus, customizing

adding hierarchical menus 69

adding menu items 67–69

altering menu items 71–73

creating checkable menu items 160

creating new menus 158–163

disabling shortcut menus 163

querying menu item status 317–318

redefining the menu bar 70–71, 163–164

removing menu items 67–69

showing/hiding the menu bar 316

Menus, list of standard names and IDs 68–69

Merging objects. *See* Combining objects

Messages

displaying in a Note dialog 328

displaying on the status bar 544

opening the Message window 366

printing to the Message window 377

Metadata

code example 321

keys 319–321

managing in tables 319–321

reading keys 265

Metadata statement 319–321

Metric units

area 470

distance 477

Microsoft Access databases

connection string attributes 441

Mid\$() function 323

MidByte\$() function 323

MIF files

exporting 235–237

importing 273–277

Military grid reference format 313, 500

Min() aggregate function 429–430

Minimizing MapInfo

Set Window statement 519–525

suppressing progress bars 511

Minimum bounding rectangle

of an object 316

of entire table 558

Minimum() function 324

Mixed case, converting to 386

Mod operator 589

Modal dialog boxes 209

Modifying an object. *See* specific object type

Arc, Ellipse, Frame, Line, Point, Polyline, Rectangle, Region, Rounded Rectangle, Text

Module-level variables 217

Month() function 324

Most-recently-used list (File menu) 160

Mouse actions 479

Mouse cursor

customizing shape of 59

displaying coordinates of 499

Moving

an object 352–353

MRU list (File menu) 160

MultiListBox controls 125–127

Multipoint objects

combining 111

creating 165–166

inserting nodes 77–78

N

Natural Break thematic ranges 175

Network file sharing 480

Nodes

adding 76, 354, 368

displaying 502

extracting a range of nodes from an object 238

maximum number per object 170, 180

querying number of nodes 335

querying x/y coordinates 339–340

removing 76

Noselect keyword 429

Note statement 328

Null handling 457

NumAllWindows() function 329

Number of characters in a string 296

NumberToDate() function 329

NumCols() function 330

Numeric operators 589

NumTables() function 330

NumWindows() function 331

O

Object model. *See* User's Guide or online Help

Object variables 217

ObjectInfo() function 334–337

ObjectLen() function 338

ObjectNodeX() function 339–340

ObjectNodeY() function 340

Objects

copying offset by distance 359–360

Objects Check statement 342–343

Objects Clean statement 343

Objects Combine statement 344–346

Objects Disaggregate statement 346–348

Objects Enclose statement 348

Objects Erase statement 349–350

Objects Intersect statement 351–352

Objects Move statement 352–353

Objects Offset statement 353–354

Objects Overlay statement 354

Objects Snap statement 355–357

Objects Split statement 357–358

Objects, copying

to offset location **353–354**

Objects, creating

arcs **137**

buffer regions **168**

by buffering **89, 166–169**

by combining objects **111**

by intersecting objects **367**

circles **144–145, 148**

convex hull **168**

ellipses **144–145, 148**

frames **149–150**

lines **154–155**

map labels **85**

multipoint **165–166**

points **171–172**

polylines **170**

rectangles **177**

regions **179–181**

rounded rectangles **182**

text **186–188**

Voronoi polygons **169**

Objects, modifying

adding nodes **76, 354**

combining **111, 344–346**

converting to polylines **131**

converting to regions **131**

erasing entire object **207**

erasing part of an object **229, 349–352**

moving nodes **73–78**

removing nodes **76**

resolution of converted objects **513**

rotating **407**

rotating around specified point **408**

setting the target object **518**

snap setting **355–357**

splitting **357–358**

Objects, moving

within input table **352–353**

Objects, querying

area **81**

boundary gaps **342–343**

boundary overlap **342–343**

centroid **101–102**

content of a text object **335**

coordinates **332–334, 339–340**

HotLink support **116**

length **338**

minimum bounding rectangle **316**

number of nodes **335**

number of polygons in a region **335**

number of sections in a polyline **335**

overlap, area of **82**

overlap, proportion of **386**

perimeter **375**

points of intersection **281**

styles **334–337**

type of object **335–337**

ODBC connection **118****ODBC tables**

changing object styles in mappable tables **464**

Offset() function **359–360****OKButton controls **120******OLE Automation**

handling button event **139**

handling menu event **161**

OnError statement **360–362****Open File statement **362–364******Open Report statement **364******Open Table statement **364–366******Open Window statement **366******Opening windows**

Browse statement **86–87**

Create Redistricter statement **178**

Graph statement **264, 268–269**

Layout statement **292**

Map statement **308–309**

Open Window statement **366**

Operating environment, determining **556–557****Operators**

automatic type conversions **592**

summary of **588–593**

Optimizing performance

animation layers **55–56**

screen updates **479**

table editing **516–518**

Oracle databases

connection string attributes **441–443**

Oracle8i databases

connection string attributes **443**

Order By clause

sorting rows **432**

Overlap() function **367****Overlaps**

checking in regions **342–343**

cleaning **343**

snapping nodes **355–357**

OverlayNodes() function **368****P****Pack Table statement **368******Page layout, opening **292******Paper units of measure **509******Papersize attribute **580******Parallel labels **503******Parent windows**

reparenting dialogs **470**

reparenting document windows **507–508**

Partialsegments option **504****PathToDirectory\$() function **369******PathToFileName\$() function **370******PathToTableName\$() function **371******Pattern matching **297******Peek requests **399****

Pen styles

- creating **306**
- modifying an object's style **74**
- Pen clause defined **372–373**
- querying an object's style **334–337**
- querying parts of **549–551**
- reading current border style **189**
- reading current line style **191**
- reading current style **191**
- setting current style **515–516**

Pen variables **217****PenPicker controls** **127****PenWidthToPoints() function** **373****Percent complete dialog** **383–385****Performance, improving**

- animation layers **55–56**
- screen updates **479**
- table editing **516–518**

Perimeter

- spherical calculation **542**

Perimeter() function **375****Per-object styles** **450****PICT files**

- creating **417–419**
- importing **273–277**

Pie chart thematic maps **534****Pie charts**

- in graph windows **264, 268–269**
- in thematic maps **534**

Platform, determining **556–557****Pline. *See* Polyline****PNG files, creating** **417–419****Point objects**

- creating **171–172**
- modifying **73–76**
- querying the symbol style **334–337**
- storing in a new row **278–279**
- storing in an existing row **570**

Point styles. *See* Symbol styles**PointsToPenWidth() function** **374****Polygon draw mode** **60****Polygons. *See* Region objects****Polyline draw mode** **60****Polyline objects**

- adding/removing nodes **76**
- Cartesian length of **97**
- converting objects to polylines **131**
- creating **170**
- creating cutter objects **147**
- determining length of **338**
- extracting a range of nodes from **238**
- modifying the pen style **74**
- querying the pen style **334–337**
- storing in a new row **278–279**
- storing in an existing row **570**

PopupMenu controls **128–129****Positioning the row cursor** **241–242****Precedence of operators** **592****Preferences dialog(s)** **413****Preventing user from closing windows** **523****Print # statement** **378****Print statement** **377****Printer settings** **519–525**

- overriding default printer **525**

Printing attributes **580****PrintWin statement** **379****Prism maps**

- creating **173–174**
- properties **380–382**
- setting **510–511**

PrismMapInfo() function **380–382****Procedures, creating**

- Call statement **91–93**
- Declare Sub statement **204–205**
- Exit Sub statement **234**
- Sub...End Sub statement **551–553**

Procedures, special

- EndHandler **228**
- ForegroundTaskSwitchHandler **255**
- Main **302–303**
- RemoteMapGenHandler **398**
- RemoteMsgHandler **398**
- SelChangedHandler **424**
- ToolHandler **563–564**
- WinChangedHandler **575**
- WinClosedHandler **576**
- WinFocusChangedHandler **583**

ProgramDirectory\$() function **383****Progress bars, hiding** **511****ProgressBar statement** **383–385****Projections**

- changing a table's projection **116–119**
- changing a window's projection **499**
- copying from a table or window **133–136**
- querying a table's CoordSys **558**
- querying a window's CoordSys **312**
- setting the current MapBasic CoordSys **474**
- setting within an application **105**

Proper\$() function **386****Proportionate aggregates**

- Proportion Avg() **52–54**
- Proportion Sum() **52–54**
- Proportion WtAvg() **52–54**

ProportionOverlap() function **386****PSD files, creating** **417–419****Put statement** **387****Q****Quantiled ranges** **176****Querying. *See* Tables, querying****R****RadioGroup controls** **129–130****Random file i/o, closing files** **107**

- opening files **363**
- reading data **263–264**
- writing data **387**

Random numbersRandomize statement **388**Rnd() function **406****Ranged thematic maps 175–177, 529–531****ReadControlValue() function 388–390****Realtime applications 55–56****Records. *See* Rows****Rectangle objects**Cartesian area of **93**Cartesian perimeter of **100**creating **177, 182**determining area of **81**determining perimeter of **375**modifying **73–76**querying the pen or brush style **334–337**storing in a new row **278–279**storing in an existing row **570****ReDim statement 390–391****Redistricting windows**closing **109**modifying **512–513, 519–525**opening **178****Region**setting a centroid **77****Region objects**adding/removing nodes **76**Cartesian area of **93**Cartesian perimeter of **100**checking for data errors **342–343**converting objects to regions **131**creating **179–181**creating convex hull objects **132**determining area of **81**determining perimeter of **375**extracting a range of nodes from **238**modifying the pen or brush style **74**querying the pen or brush style **334–337**returning a buffer region **94**storing in a new row **278–279**storing in an existing row **570****Regional settings 480–481****Register Table statement 392–396****Relational joins 427–429****Relief Shade statement 397****Reload Symbols statement 397****Remote databases**creating new tables **450–452**refreshing linked tables **462**retrieving active database connection information **447**shutting down server connection **453****RemoteMapGenHandler procedure 398****RemoteMsgHandler procedure 398****RemoteQueryHandler function 399****Remove Cartographic Frame statement 400****Remove Map statement 401****Removing**buttons **57–61**menu items **67–69**nodes **76****Rename File statement 402****Rename Table statement 402****Reports**creating **181**loading **364****Reserved words 219****Resizing arrays 390–391****Responding to system events. *See* Events, handling****Resume statement 403****Retrieving column information**Server_ColumnInfo() **438–439****Retrieving data source information**Server_DriverInfo() **454****Retrieving number of columns in a results set**Server_NumCols() **461****Retrieving number of toolkits**Server_NumDrivers() **462****Retrieving records from an open table**Fetch statement **241–242****Retrieving rows from a results set, Server Fetch 456–457****Retrying on file access 480****Returning a connection number**Server_Connect **440–447****Returning a coordinate system**ChooseProjection\$() function **105****Returning a date**FormatDate\$ function **258****Returning a pen width for a point size**PointsToPenWidth() function **374****Returning a point size for a pen width**PenWidthToPoints() function **373****Returning ODBC connection handle**Server_GetodbcHConn() function **458****Returning ODBC statement handle**Server_GetodbcHStmt() function **458****Returning pen width units**IsPenWidthPixels() function **282****RGB() function 404****Right\$() function 405****Right-click menus 68–69****Rnd() function 406****Rollback statement 406****Rotate() function 407****RotateAtPoint() function 408****Rotated map labels 503****Rotated symbols 305, 553****Rounded Rectangle objects, Cartesian area of 93****Rounded rectangle objects**Cartesian perimeter of **100**creating **182**modifying **73–76**querying the pen or brush style **334–337**storing in a new row **278–279**storing in an existing row **570****Rounding off a number**Fix() function **251**Format\$() function **256–258**Int() function **280**Round() function **408**

RowID

- after Find operations **115**
- with SelChangedHandler **114**

RowID column. See online Help**Rows in a Browser, positioning **472******Rows in a table**

- deleting rows **207**
- end-of-table condition **229**
- inserting new rows **278–279**
- packing (purging deleted rows) **368**
- positioning the row cursor **241–242**
- selecting rows that satisfy criteria **427–429**
- updating existing rows **570**

RPC (Remote Procedure Calls) **204–205****RTrim\$() function **409******Ruler tool**

- closing Ruler window **109**
- modifying Ruler window **519–525**
- opening Ruler window **366**

Run Application statement **410****Run Command statement **410–412******Run Menu Command statement **412******Run Program statement **414******Runtime errors, trapping. See Error handling****S****Save Window statement **417–419******Save Workspace statement **419******Saving changes to a table **116–119******Saving linked tables **118******Saving work to the database**

- Server Commit **439**

Scale of a map

- determining **313**
- displaying **498**

Scope of variables

- global **266**
- local **216–220**
- module-level **217**

Scroll bars, showing/hiding **524****Scrolling automatically **524******Seamless tables**

- determine if table is seamless **559**
- prompt user to choose a sheet **265**
- turn seamless behavior on/off **517**

SearchInfo() function **419–421****Searching for map objects**

- at a point **422**
- processing search results **419–421**
- within a rectangle **423**

SearchPoint() function **422****SearchRect() function **423******Seconds, elapsed **562******Seek statement **424******Seek() function **423******SelChangedHandler procedure **424******Select Case. See Do Case...End Case statement****Select statement **425–434******Selectable map layers **501******Selection**

- handling selection-changed event **114, 424**
- interrupted by Esc key **425**
- querying current selection **434**
- Select statement **425–434**

SelectionInfo() function **434****Self-intersections**

- checking in regions **342–343**

Sequential file i/o

- closing files **107**
- opening files **363**
- reading data **277, 298**
- writing data **378, 584**

Server Begin Transaction statement **435****Server Bind Column statement **436–437******Server Close statement **437******Server Commit statement **439******Server Create Map statement **448–449******Server Create Style statement **450******Server Create Table statement **450–452******Server Disconnect statement **453******Server DriverInfo() function **454******Server Fetch statement **456–457******Server Link Table statement **459–461******Server Refresh statement **462******Server Rollback statement **463******Server Set Map statement **464******Server_ColumnInfo() function **438–439******Server_Connect() function **440–447******Server_ConnectInfo() function **447******Server_EOT() function **455******Server_Execute function **455******Server_GetodbcHConn() function **458******Server_GetodbcHStmt() function **458******Server_NumCols() function **461******Server_NumDrivers() function **462******server_string**

- defined **455**

SessionInfo() function **469****Set Application Window statement **470******Set Area Units statement **470******Set Browse statement **472******Set Cartographic Legend statement **472******Set Command Info statement **473******Set CoordSys statement **474******Set Date Window statement **475******Set Digitizer statement **476–477******Set Distance Units statement **477******Set Drag Threshold statement **479******Set Event Processing statement **479******Set File Timeout statement **480******Set Format statement **480–481******Set Graph statement **482–486******Set Handler statement **487******Set Layout statement **487–489******Set Legend statement **490–492******Set Map statement **493–505******Set Map3D statement **506–507****

Set Next Document statement **507–508**
 Set Paper Units statement **509**
 Set PrismMap statement **510–511**
 Set ProgressBars statement **511**
 Set Resolution statement **513**
 Set Shade statement **514**
 Set Style statement **515–516**
 Set Table statement **516–518**
 Set Target statement **518**
 Set Window statement **519–525**
 Sgn() function **526**
 Shadow text **252–253**
 Shapefiles **395**
 Shift key
 detecting shift-click **115**
 effect on drawing tools **60**
 selecting multiple list items **125–127**
 Shortcut menus
 defined **68–69**
 disabling **163**
 example **69**
 list of menu names and IDs **68–69**
 Show/Hide menu commands **160**
 Showing
 ButtonPads **57–61**
 dialog controls **62–64**
 menu bar **316**
 Shutting down the connection
 Server Disconnect **453**
 Simulating a menu selection **412**
 Sin() function **535**
 Small integer variables **217**
 Smart redraw **497**
 Snap tolerance
 controlling **525**
 Snapping nodes **355–357**
 Sorting rows in a table **432**
 Sounds, beeping **86**
 Space\$() function **536**
 Spaces, trimming from a string **301, 409**
 Speed, improving
 animation layers **55–56**
 screen updates **479**
 table editing **516–518**
 SphericalArea() function **537**
 SphericalDistance() function **538**
 SphericalObjectLen() function **539**
 SphericalOffset() function **540–541**
 SphericalPerimeter() function **542**
 Splitting objects **357–358**
 Spreadsheets, using as tables **392–396**
 SQL Select **425–434**
 SQL Server databases
 connection string attributes **443–446**
 Sqr() function **543**
 Starting other applications
 Run Application statement **410**
 Run Program statement **414**
 StaticText controls **130**

Statistical calculations
 average **52, 429–430**
 count **52, 429–430**
 min/max **52, 429–430**
 quantile **176**
 standard deviation **175**
 sum **52, 429–430**
 weighted average **52, 431**
 Statistics window, closing **109**
 modifying **519–525**
 opening **366**
 Status bar help **58, 139**
 StatusBar, statement **544**
 Stop statement **544**
 Str\$() function **545**
 Street addresses, finding **247–249**
 String concatenation
 & operator **589**
 + operator **589**
 String functions
 capitalization **293, 386, 567**
 comparison **547–548**
 converting codes to strings **106**
 converting strings to codes **82**
 converting strings to dates **480–481, 548–549**
 converting strings to numbers **480–481, 572**
 converting values to strings **545**
 extracting part of a string **293, 323, 405**
 finding a substring within a string **279**
 formatting a number **206, 256–259, 480–481**
 formatting based on locale **480–481**
 length of string **296**
 locale settings **480–481**
 pattern matching **297**
 repeated strings **546**
 spaces **536**
 trimming spaces from end **409**
 trimming spaces from start **301**
 String variables **217**
 String\$() function **546**
 StringCompare() function **547**
 StringCompareIntl() function **548**
 StringToDate() function **548–549**
 Structures **566**
 StyleAttr() function **549–551**
 Styles. *See* specific type, Pen, Brush, Font, Symbol
 Sub procedures. *See* Procedures
 Sub...End Sub statement **551–553**
 Subtotals, calculating **429–430**
 Sum() aggregate function **429–430**
 Symbol styles
 creating **304–305, 307**
 modifying an object's style **74**
 querying an object's style **334–337**
 querying parts of **549–551**
 reading current style **192**
 reloading symbol sets **397**
 setting current style **515–516**
 Symbol clause defined **553–555**

Symbol variables **217**

SYMBOL.MBX utility

custom symbols **397**

SymbolPicker controls **127**

SystemInfo() function **556–557**

T

TAB files

storing metadata in **319–321**

Tab order **211**

Table names

determining from file **371**

determining name of table in Browse or Graph window **579**

determining table name from number **557–560**

special names for Cosmetic layers **579**

special names for Layout windows **579**

Table structure

3DMap **157–158**

adding/removing columns **79–80**

determining how many columns **330, 557–560**

making a table mappable **156**

making an ODBC table mappable **448–449**

TableInfo() function **557–560**

Tables, closing

Close All statement **107**

Close Table statement **108**

Tables, copying **116–119**

Tables, creating

creating a new table **184**

importing a file **273–277**

on remote databases **450–452**

using a spreadsheet or database **392–396**

Tables, deleting **226**

Tables, importing **273–277**

Tables, modifying

adding columns **50–54, 79–80**

adding metadata **319–321**

adding rows **278–279**

creating an index **153**

deleting a table's objects **225**

deleting an index **224**

deleting columns **79–80**

deleting rows or objects **207**

discarding changes **406**

optimizing edit operations **516–518**

packing **368**

renaming **402**

saving changes **116–119**

setting a map's default view **504**

setting a map's projection **116–119**

setting to read-only **516–518**

sorting rows **432**

updating existing rows **570**

Tables, opening **364–366**

Tables, querying

column information **110–111**

directory path **559**

end-of-table condition **229**

Tables, querying (continued)

finding a map address **247–249**

joining **427–429**

metadata **265, 319–321**

number of open tables **330**

objects at a point **422**

objects in a rectangle **423**

positioning the row cursor **229, 241–242**

SQL Select **425–434**

table information **557–560**

Tan() function **560**

TempFileName\$() function **560**

Temporary columns **50**

Terminate Application statement **561**

Text files

using as tables **392–396**

See also File input/output, Files

Text objects

creating **186–188**

modifying **73–76**

querying the font style or string **335**

storing in a new row **278–279**

storing in an existing row **570**

Text styles. *See* Font styles

TextSize() function **562**

Thematic maps

bar chart maps **535**

counting number of themes in a 3D Map window **309–311**

counting number of themes in a Map window **311–315**

creating arrays of ranges **175–177**

creating arrays of styles **183–184**

dot density maps **532–533**

graduated symbol maps **533**

grid surface maps **150–152**

individual value maps **531**

modifying **514**

pie chart maps **534**

quantiled ranges **176**

ranged maps **529–531**

Thinning objects **355–357**

Thousands separators **206, 259, 480–481**

TIFF files, creating **417–419**

Time delay when user drags mouse **479**

Timer() function **562**

Toolbars. *See* ButtonPads

ToolHandler procedure **563–564**

Tooltip help **58, 139**

Totals, calculating **429–430**

Transparent fill patterns **88**

Trapping errors. *See* Error handling

TriggerControl() function **565**

Trigonometric functions

arc-cosine **47**

arc-sine **83**

arc-tangent **84**

cosine **136**

sine **535**

tangent **560**

Trimming spacesfrom end of string **409**from start of string **301****TrueFileName\$() function** **565****TrueType fonts, using as symbols** **305****TrueType symbols** **553–555****Type statement** **566****U****UBound() function** **567****UCase\$() function** **567****Unchecking**dialog check boxes (custom) **62–64**dialog check boxes (standard) **64–66**menu items **71–73****Underlined text** **252–253****UnDim statement** **568****Undo system, disabling** **516–518****UnitAbbr\$() function** **569****UnitName\$() function** **569****Units of measure**abbreviated names **569**area **470**distance **477**full names **569**paper **509****Unlink statement** **570****Unselecting** **108****Update statement** **570****Update Window statement** **571****Upper case, converting to** **567****User interface. See ButtonPads, Dialogs, Menus, Windows****V****Val() function** **572****Variable length strings** **217****Variables**arrays **218, 390–391, 567**custom types **566**global variables **266**initializing **220**list of types **216–217**local variables **216–220**module-level variables **217**reading another application's variables **266**restrictions on names **219**strings variables **218**undefining **568****Version number**.MBX version **556**MapInfo version **557****Vertices. See Nodes****Voronoi polygons**Create Object **169****W****Weekday() function** **573****Weighted averages** **52, 431****While...Wend statement** **574****Wildcards, matching** **297****WinChangedHandler procedure** **575****WinClosedHandler procedure** **576****WindowID() function** **576****WindowInfo() function** **577–582****Windows Latin 1 character set** **586****Windows operating system**16- v. 32-bit **556****Windows, closing**Close Window statement **109**preventing user from closing windows **523****Windows, modifying**adding map layers **54–56**browser windows **472**forcing windows to redraw **571**general window settings **519–525**graph windows **482–486**layout windows **487–489**legend window **490–492**map windows **493–505**redistrict windows **512–513**removing map layers **401****Windows, opening**Browse statement **86–87**Create Redistricter statement **178**Graph statement **268–269**Layout statement **292**Map statement **308–309**Open Window statement **366****Windows, printing**to a file **417–419**to an output device **379****Windows, querying**3D Map window settings **309–311**general window settings **577–582**ID of a window **576**ID of front window **260**map window settings **288–291, 311–315**number of document windows **331**total number of windows **329****WinFocusChangedHandler procedure** **583****WKS files, opening** **392–396****WMF files, creating** **417–419****Workspaces**loading **410**saving **419****Write # statement** **584****WtAvg() aggregate function** **431****X****XCMDs** **204–205****XFCNs** **203****XLS files, opening** **392–396****Y****Year() function** **584**