



MapInfo MapBasic v. 8.0

User 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: Getting Started	13
Hardware & Software Requirements	14
Compatibility with Previous Versions	14
Installing the MapBasic Development Environment	14
Starting MapBasic	14
MapBasic File Names and File Types	15
MapBasic Documentation Set	16
MapBasic® Reference	16
Installing Online Documentation	16
Conventions Used in This Manual	16
Terms	16
Typographical Conventions	17
Register Today!	17
Working with Technical Support	17
Before You Call	17
The Support Tracking System	18
Expected Response Time	18
Exchanging Information	18
Software Defects	18
Other Resources	18
Chapter 2: New and Enhanced MapBasic Statements and Functions	20
Enhanced MapBasic Functions and Statements	37
Enabling Transparent Patterns on Same Layer	47
Export Windows to Additional Formats	47
Chapter 3: A Quick Look at MapBasic	49
Getting Started	50
How Do I Create and Run a MapBasic Application?	51
What Are the Key Features of MapBasic?	51
MapBasic Lets You Customize MapInfo	51
MapBasic Lets You Automate MapInfo	51
MapBasic Provides Powerful Database-Access Tools	52
MapBasic Lets You Connect MapInfo To Other Applications	52
How Do I Learn MapBasic?	52
The MapBasic Window in MapInfo	54
Training and On-Site Consulting	54

Chapter 4: Using the Development Environment	56
Introduction to MapBasic Development Environment	57
Editing Your Program	57
Keyboard Shortcuts	57
Limitations of the MapBasic Text Editor	59
Compiling Your Program	60
A Note on Compilation Errors	61
Running a Compiled Application	61
Using Another Editor to Write MapBasic Programs	61
Linking Multiple Modules Into a Single Project	62
What is a MapBasic Project File?	62
Creating a Project File	64
Compiling and Linking a Project	64
Calling Functions or Procedures From Other Modules	65
Menu Summary in MapBasic Development Environment	66
The Edit Menu	67
The Search Menu	68
The Project Menu	69
The Window Menu	70
The Help Menu	70
Chapter 5: MapBasic Fundamentals	71
General Notes on MapBasic Syntax	72
Comments	72
Case-Sensitivity	72
Continuing a Statement Across Multiple Lines	72
Codes Defined In mapbasic.def	72
Typing Statements Into the MapBasic Window	73
Variables	73
Fixed-length and variable-length String variables	75
Array Variables	75
Custom Data Types (Data Structures)	76
Global Variables	77
Scope of Variables	78
Expressions	78
What is a Constant?	78
What is an Operator?	79
What is a Function Call?	79
A Closer Look At Constants	80
Variable Type Conversion	83
A Closer Look At Operators	83
MapBasic Operator Precedence	86
Looping, Branching, and Other Flow-Control	87
If...Then Statement	87
Do Case Statement	88
GoTo Statement	89

For...Next Statement	90
Do...Loop	90
While...Wend Loop	91
Ending Your Program	91
Ending Your Program and MapInfo Professional	91
Procedures	92
Main Procedure	92
Calling a Procedure	92
Calling a Procedure That Has Parameters	93
Passing Parameters By Reference	93
Passing Parameters By Value	93
Calling Procedures Recursively	94
Procedures That Act As System Event Handlers	95
What Is a System Event?	95
What Is an Event Handler?	95
When Is a System Event Handler Called?	97
Tips for Handler Procedures	98
Keep Handler Procedures Short.	98
Selecting Without Calling SelChangedHandler	98
Preventing Infinite Loops	98
Custom Functions	99
Scope of Functions	99
Compiler Instructions	100
The Define Statement	100
The Include Statement	100
Program Organization	102
Chapter 6: Debugging and Trapping Runtime Errors	103
Runtime Error Behavior	104
Debugging a MapBasic Program	104
Summary of the Debugging Process	105
Limitations of the Stop Statement.	105
Other Debugging Tools	106
Error Trapping	106
Example of Error Trapping	107
Chapter 7: Creating the User Interface	108
Introduction to MapBasic User Interface Principles	109
Event-Driven Programming	109
What Is an Event?	109
What Happens When The User Generates A Menu Event?	109
How Does a Program Handle ButtonPad Events?	110
How Does a Program Handle Dialog Events?	111
Menus	111
Menu Fundamentals.	111
Adding New Items To A Menu	112
Removing Items From A Menu	112

Creating A New Menu	113
Altering A Menu Item	114
Re-Defining The Menu Bar	116
Specifying Language-Independent Menu References	116
Customizing MapInfo Professional's Shortcut Menus	117
Assigning One Handler Procedure To Multiple Menu Items	117
Simulating Menu Selections	118
Defining Shortcut Keys And Hot Keys	118
Controlling Menus Through the MapInfo Professional Menus File	119
Standard Dialog Boxes	121
Displaying a Message	121
Asking a Yes-or-No Question	121
Selecting a File	122
Indicating the Percent Complete	122
Displaying One Row From a Table	122
Custom Dialog Boxes	123
Sizes and Positions of Controls	124
Control Types	125
Specifying a Control's Initial Value	127
Reading a Control's Final Value	127
Responding to User Actions by Calling a Handler Procedure	128
Enabled / Disabled Controls	128
Letting the User Choose From a List	129
Managing MultiListBox Controls	129
Specifying Shortcut Keys for Controls	130
Terminating a Dialog Box	130
Windows	131
Specifying a Window's Size and Position	132
Map Windows	132
Using Animation Layers to Speed Up Map Redraws	133
Sample Program	133
Performance Tips for Animation Layers	133
Browser Windows	134
Graph Windows	135
Layout Windows	135
Redistrict Windows	136
Message Window	136
ButtonPads (Toolbars)	138
What Happens When The User Chooses A Button?	138
MapBasic Statements Related To ButtonPads	138
Create ButtonPad	139
Alter ButtonPad	139
Alter Button	139
CommandInfo()	139
ToolHandler	139
Creating A Custom PushButton	140

Adding A Button To The Main ButtonPad	140
Creating A Custom ToolButton	141
Choosing Icons for Custom Buttons	142
Selecting Objects by Clicking With a ToolButton	143
Including Standard Buttons in Custom ButtonPads	143
Assigning Help Messages to Buttons	144
Docking a ButtonPad to the Top of the Screen	145
Other Features of ButtonPads	145
Integrating Your Application Into MapInfo Professional	145
Loading Applications Through the Startup Workspace	146
Manipulating Workspaces through MapBasic	147
Performance Tips for the User Interface	147
Animation Layers	147
Avoiding Unnecessary Window Redraws	147
Purging the Message Window	148
Chapter 8: Working With Tables	149
Opening Tables Through MapBasic	150
Determining Table Names at Runtime	150
Opening Two Tables With The Same Name	150
Opening Non-Native Files As Tables	151
Reading Row-And-Column Values From a Table	152
Alias Data Types as Column References	153
Scope	154
Using the "RowID" Column Name To Refer To Row Numbers	155
Using the "Obj" Column Name To Refer To Graphic Objects	155
Finding Map Addresses In Tables	156
Geocoding	156
Performing SQL Select Queries	156
Error Checking for Table and Column References	156
Writing Row-And-Column Values to a Table	157
Creating New Tables	157
Modifying a Table's Structure	157
Creating Indexes and Making Tables Mappable	158
Reading A Table's Structural Information	159
Working With The Selection Table	159
Changing the Selection	160
Updating the Currently-Selected Rows	161
Using the Selection for User Input	161
Accessing the Cosmetic Layer	162
Accessing Layout Windows	162
Multi-User Editing	163
The Rules of Multi-User Editing	163
Preventing Conflicts When Writing Shared Data	165
Opening a Table for Writing	166
Files that Make Up a Table	166

Raster Image Tables	167
Working With Metadata	169
What is Metadata?	169
What Do Metadata Keys Look Like?	169
Examples of Working With Metadata	170
Working With Seamless Tables	171
What is a Seamless Table?	171
How Do Seamless Tables Work?	172
MapBasic Syntax for Seamless Tables	172
Limitations of Seamless Tables	173
Accessing DBMS Data	173
How Remote Data Commands Communicate with a Database	173
Connecting and Disconnecting	174
Accessing/Updating Remote Databases with Linked Tables	175
Live Access to Remote Databases	176
Performance Tips for Table Manipulation	176
Minimize Transaction-File Processing	176
Use Indices Where Appropriate	177
Using Sub-Selects	177
Optimized Select Statements	177
Using Update Statements	177
Chapter 9: File Input/Output	178
Overview of File Input/Output	179
Sequential File I/O	180
Random File I/O	182
Binary File I/O	182
Platform-Specific & International Character Sets	182
File Information Functions	183
Chapter 10: Graphical Objects	184
Using Object Variables	185
Using the “Obj” Column	185
Creating an Object Column	186
Limitations of the Object Column	186
Querying An Object’s Attributes	187
Object Styles (Pen, Brush, Symbol, Font)	188
Understanding Font Styles	189
Style Variables	190
Selecting Objects of a Particular Style	191
Creating New Objects	193
Object-Creation Statements	193
Object-Creation Functions	194
Creating Objects With Variable Numbers of Nodes	194
Storing Objects In a Table	195

Creating Objects Based On Existing Objects	196
Creating a Buffer	196
Using Union, Intersection, and Merge	196
Creating Offset Copies	197
Modifying Objects	197
General Procedure for Modifying an Object	197
Repositioning An Object	198
Moving Objects and Object Nodes	198
Modifying An Object's Pen, Brush, Font, or Symbol Style	198
Converting An Object To A Region or Polyline	198
Erasing Part Of An Object	199
Points Of Intersection	199
Working With Map Labels	199
Turning Labels On	199
Turning Labels Off	200
Editing Individual Labels	200
Querying Labels	200
Other Examples of the Set Map Statement	201
Differences Between Labels and Text Objects	201
Coordinates and Units of Measure	203
Units of Measure	204
Advanced Geographic Queries	205
Using Geographic Comparison Operators	205
Querying Objects in Tables	206
Using Geographic SQL Queries With Subselects	207
Using Geographic Joins	208
Proportional Data Aggregation	209
Chapter 11: Advanced Features of Microsoft Windows	210
Declaring and Calling Dynamic Link Libraries (DLLs)	211
Specifying the Library	211
Passing Parameters	212
Calling Standard Libraries	212
Calling a DLL Routine by an Alias	212
Array Arguments	213
User-Defined Types	213
Logical Arguments	213
Handles	214
Example: Calling a Routine in KERNEL	214
Troubleshooting Tips for DLLs	215
Creating Custom Button Icons and Draw Cursors	216
Reusing Standard Icons	216
Custom Icons	217
Custom Draw Cursors for Windows	218

Inter-Application Communication Using DDE	218
Overview of DDE Conversations	218
How MapBasic Acts as a DDE Client	218
How MapInfo Acts as a DDE Server	220
How MapInfo Handles DDE Execute Messages	222
Communicating With Visual Basic Using DDE	223
Examples of DDE Conversations	223
DDE Advise Links	223
Incorporating Windows Help Into Your Application	223
Chapter 12: Integrated Mapping	225
What Does Integrated Mapping Look Like?	226
Conceptual Overview of Integrated Mapping	227
Technical Overview of Integrated Mapping	228
System Requirements	228
Other Technical Notes	228
A Short Sample Program: “Hello, (Map of) World”	229
A Closer Look at Integrated Mapping	229
Sending Commands to MapInfo	230
Querying Data from MapInfo	230
Customizing MapInfo’s Shortcut Menus	235
Terminating Your Visual Basic Program	236
A Note About MapBasic Command Strings	236
A Note About Dialog Boxes	237
A Note About Accelerator Keys	237
Using Callbacks to Retrieve Info from MapInfo	237
Technical Requirements for Callbacks	238
General Procedure for Using OLE Callbacks	238
Processing the Data Sent to a Callback	239
C/C++ Syntax for Standard Notification Callbacks	240
Alternatives to Using OLE Callbacks	241
DDE Callbacks	241
MBX Callbacks	242
Displaying Standard MapInfo Help	242
Disabling Online Help	242
Displaying a Custom Help File	242
Related MapBasic Statements and Functions	243
MapInfo Command-Line Arguments	253
Getting Started with Integrated Mapping and Visual C++ with MFC	254
Add OLE Automation Client Support	255
Create the MapInfo Support class, and create an instance of it	255
Test your work	256
Redefine the Shortcut Menus	256
Reparenting MapInfo’s Dialogs	256
Adding a Map to your View	257
Adding a Map Menu Command	258

Adding Toolbar Buttons and Handlers	258
Using Exception Handling to Catch MapInfo Errors	260
Add OLE Automation Server Support.	260
Adding the WindowContentsChanged Callback.	261
Learning More	261
Appendix A: Sample Programs	262
Samples\Delphi Folder	263
Samples DLLEXAMP Folder	263
Samples\MFC Folder	268
Samples\PwrBldr Folder	268
Samples\VB4 Folder	268
Samples\VB6 Folder	269
Appendix B: Summary of Operators	270
Numeric Operators	271
Comparison Operators	272
Logical Operators	272
Geographic Operators	273
Precedence	274
Automatic Type Conversions	275
Appendix C: List of MapBasic Changes by Version	276
Features Introduced or Changed in MapBasic 7.8	277
Features Introduced in MapBasic 7.5	278
Features Introduced in MapBasic 7.0	278
Appendix D: Supported ODBC Table Types	280
Appendix E: Making a Remote Table Mappable	281
Prerequisites for Storing/Retrieving Spatial Data	282
Creating a MapInfo Map Catalog	282
Appendix F: Data Setting and Management	284
Upgrading Applications from Versions Prior to 6.5	285
A Glossary for Upgrading Applications	286
Application Data Files and Directories	287
Default Preferences Paths	289
Registry Changes	289
Installer Requirements and Group Policies	290
MapBasic 6.5	290
MapBasic 7.0	290
Appendix GL: MapBasic Glossary	291
Index	299

Getting Started

Welcome to the MapBasic Development Environment 8.0, the powerful, yet easy-to-use programming language that lets you customize and automate MapInfo Professional.

The following pages tell you what you need to know to install the MapBasic software. For information on the purpose and capabilities of MapBasic, see **Chapter 3: A Quick Look at MapBasic**.

Sections in this Chapter:

- ♦ **Hardware & Software Requirements. 14**
- ♦ **Installing the MapBasic Development Environment 14**
- ♦ **MapBasic File Names and File Types 15**
- ♦ **MapBasic Documentation Set 16**
- ♦ **Conventions Used in This Manual 16**

Hardware & Software Requirements

Before installing MapBasic for Windows, please make certain that your computer meets the following minimum requirements:

Requirement	Your choices are:
System Software	Microsoft Windows XP/2000/98 or Windows NT 4.0/2000
Display	Any display adapter supported by Windows
Mouse	Any mouse or pointing device supported by Windows
Disk space	10 MB

Compatibility with Previous Versions

MapInfo Professional can run applications created with current or earlier versions of MapBasic.

See [Appendix C: List of MapBasic Changes by Version](#) for more information about backwards compatibility.

Installing the MapBasic Development Environment

Before You Begin

The MapBasic installation procedure is described below. If you haven't already done so:

- Install MapInfo Professional before you install MapBasic. Please see the MapInfo Professional *User Guide* for installation instructions.
- Write your MapBasic serial number in an easy-to-remember place, such as the title page of the manual.

Installation

1. From the MapBasic CD, choose Install MapBasic and follow the on-screen installation.
The default location for MapBasic is a directory inside the MapInfo directory (for example, C:\ProgramFiles\MAPINFO\MAPBASIC\MAPBASIC.EXE).
If the CD does not automatically start, from the CD drive, click **SETUP**.

Starting MapBasic

To start the MapBasic Development Environment,

1. Run the **WINDOWS PROGRAM MANAGER**.
2. To run MapBasic, choose **MAPBASIC** from the MapInfo Program Group.

Note: You can check for product updates to your version anytime by selecting **HELP > CHECK FOR UPDATE**.

MapBasic File Names and File Types

The MapBasic installation procedure places these files on your computer:

File Name	Description
errors.doc:	Text file listing MapBasic error codes
mapbasic.exe:	executable file which runs the MapBasic development environment
mapbasic.def:	Include file containing standard define codes
menu.def:	Include file containing menu-related define codes
icons.def:	Include file containing ButtonPad- and cursor-related define codes
mapbasic.hlp:	MapBasic on-line help file
mapbasic.h:	Header file for C/C++ programmers; contents similar to mapbasic.def, but using C/C++ syntax
mapbasic.bas:	Header file for Visual Basic programmers; contents similar to mapbasic.def, but using Visual Basic syntax
mapbasic65.isu	Uninstall log file -- needed to properly uninstall MapBasic.
mbres650.dll	Part of the software; contains resources such as strings and dialogs.
milib650.dll	Part of the software; contains XVT executable code
papersize.def	Include file for use by MB application developers; contains defines for use with printer control MapBasic statements
usrinfmb.log	Contains log of installation process.
samples folder	<i>contains filename.mb, filename.mbp: sample programs;</i>

As you use the MapBasic development environment, you produce files with the following extensions:

File Name	Description
<i>filename.mb</i>	Program files (source code)
<i>filename.mbx</i>	Compiled (executable) files
<i>filename.mbp</i>	Project files (which list all modules to include in a project)
<i>filename.mbo</i>	Object files (files created after compiling modules in a project)
<i>filename.err</i>	Error listings, generated if you compile a program that has compilation errors.

MapBasic Documentation Set

In addition to the *User Guide*, MapBasic's documentation set includes an online version of this guide, online MapBasic *Reference*, and online Help.

MapBasic® Reference

The MapBasic online *Reference* is a complete guide to all MapBasic commands. See Using the MapBasic Window, for a discussion of which MapBasic commands can be used.

Installing Online Documentation

Access the online MapBasic *Reference* or *User Guide* directly from the MapBasic CD, or install the Adobe® Acrobat Reader to access the files locally.

Choose to access either of the online manuals directly from the CD.

To install the documentation locally:

1. Install the Acrobat® Reader.
2. Copy the files from the [CD_ROM]:\PDF_DOCS folder to a local directory.
mb70ug.pdf is this Guide and requires ~8 MB of disk space.
mb_ref.pdf is the MapBasic Reference Guide and requires ~10 MB of disk space.
3. From Windows Explorer, double-click on either file to automatically launch the Acrobat® Reader and the online books.

Conventions Used in This Manual

This manual uses the following terms and typographical conventions.

Terms

This manual addresses the application developer as *you*, and refers to the person using an application as the *user*. For example:

You can use MapBasic's **Note** statement to give *the user* a message.

The terms *program* and *application* are used in the following manner:

A *program* is a text file typed in by you, the programmer. Typically, MapBasic program files have the extension .MB.

An *application* file is a binary file executable by MapInfo. The application file must be present when the user runs the application. MapBasic creates the application file when you compile your program. MapBasic application files typically have the extension .MBX (MapBasic eExecutable).

A *command* is an item that you choose from a menu. For example, to open a file, choose the Open command from the File menu.

A *statement* is an instruction you can issue from a MapBasic program. For example, a MapBasic program can issue a **Select** statement to select one or more rows from a table.

Typographical Conventions

The `Courier` font shows sample MapBasic program statements:

```
Note "hello, world!"
```

Bold Capitalization identifies MapBasic keywords:

The **Stop** statement is used for debugging purposes.

In the examples that appear in this manual, the first letter of each MapBasic language keyword is capitalized. However, you are not required to use the same capitalization when you type in your own programs. If you prefer, you can enter your programs using upper case, lower case or mixed case.

References to menu commands in the MapBasic development environment use the greater-than sign (>), as in the following example:

- Choose the File > New command to open a new edit window.

The expression "File > New" refers to the New command on the File menu.

Register Today!

If you haven't already done so, please fill in your product registration card. If you register, you can receive newsletters and information about future upgrades.

Working with Technical Support

Technical Support is here to help you, and your call is important. This section lists the information you need to provide when you call your local support center. It also explains some of the technical support procedures so that you will know what to expect about the handling and resolution of your particular issue.

Before You Call

Please have the following information ready when contacting us for assistance on MapInfo Professional.

1. Serial Number. You must have a registered serial number to receive Technical Support.
2. Your name and organization. The person calling must be the contact person listed on the support agreement.
3. Version of the product you are calling about.
4. The operating system name and version.
5. A brief explanation of the problem. Some details that can be helpful in this context are:
 - Error messages
 - Context in which the problem occurs
 - Consistency - is the problem reoccurring or occurring erratically?

The Support Tracking System

The Support Tracking System is used internally by the Technical Support department to manage and track customer issues. The system also provides the ability to track calls with accountability. This system helps Tech Support respond to all customer issues effectively, efficiently, and fairly.

Expected Response Time

Most issues can be resolved during the customer's initial call. If this is not possible, a response will be issued before the end of the business day. A Technical Support representative will provide a status each business day until the issue is resolved.

Support requests submitted by e-mail are handled using the same guidelines as telephone support requests; however, there is an unavoidable delay of up to several hours for message transmission and recognition.

Exchanging Information

Occasionally a Technical Support representative will ask you to provide sample data in order to duplicate your scenario. In the case of our developer tools (such as MapX and MapXtreme), a small subset of sample code may be requested to help duplicate the issue.

The preferred method of exchanging information is either via e-mail or our FTP site. Use following e-mail addresses:

- United States - techsupport@mapinfo.com
- Europe - support-europe@mapinfo.com
- Australia - ozsupport@mapinfo.com

Software Defects

If the issue is deemed to be a bug in the software, the representative will log the issue in MapInfo Corporation's bug base and provide you with an incident number that can be used to track the bug. Future upgrades and patches have fixes for many of the bugs logged against the current version.

Other Resources

MapInfo Test Drive Center

The Test Drive Center on MapInfo Corporation's Web site is a forum for technical users of our products to learn about MapInfo Corporation's latest software offerings. You can download trial versions of software, as well as obtain patches and fixes.

You'll need to complete a registration form to gain access to most areas of the Test Drive Center. This is a one-time process. As new products and services become available in the Test Drive Center, you will not need to re-register to access them. You can simply update your existing registration information to indicate an interest in the new product.

MapInfo-L Archive Database

MapInfo Corporation, in conjunction with Bill Thoen, provides a web-based, searchable archive database of MapInfo-L postings. The postings are currently organized by Discussion Threads and Postings by Date.

Disclaimer: While MapInfo Corporation provides this database as a service to its user community, administration of the MapInfo-L mailing list is still provided by Bill Thoen. More information on MapInfo-L can be obtained at the MapInfo Test Drive Center (<http://testdrive.mapinfo.com>).

MapInfo Automated Fax Support

MapInfo Technical Support's Automated Fax Support system puts the latest Technical Support solutions to common technical questions into your hands almost immediately. You can access hundreds of technical documents on MapInfo products using the system. These fax documents are updated constantly to ensure that you are receiving the latest technical information. This service is available 24 hours a day, 7 days a week, free of charge. No support agreement is required.

To use Automated Fax Support, all you need is a touch-tone phone and a fax machine. Here's how:

- Call 518-285-7283, and choose option 4.
- Follow the simple instructions.
- Select a fax using a document number or receive an index of available documents.
- Enter your fax number and the selected documents will be delivered immediately.

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 21**
- ♦ **Enhanced MapBasic Functions and Statements 37**

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 (for example, 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 (for example, 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 (for example, 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 (for example, 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 (for example, 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 (for example, 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 (for example, *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 (for example, 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 (for example, *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:

```
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 (for example, 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 node 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 is 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=TROJNY;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=TROJNY;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 (for example, 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 (for example, 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 (for example, 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 doesn't 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 case-sensitive name of the current workspace in which the table is operated.

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

Note: In order to use have this statement be effective, the table has to be version-enabled, i.e., *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
```

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 "tiff1zw"
```

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.

A Quick Look at MapBasic

MapBasic is a software package that lets you customize and automate the MapInfo desktop-mapping software.

Sections in this Chapter:

- ♦ **Getting Started 50**
- ♦ **What Are the Key Features of MapBasic? 51**
- ♦ **How Do I Learn MapBasic? 52**
- ♦ **The MapBasic Window in MapInfo 54**

Getting Started

The MapBasic software provides you with a *development environment*. Using this development environment, you can write programs in the MapBasic programming language.

The MapBasic development environment includes:

- A text editor you can use to type your programs. If you already have a text editor you would rather use, you can use that editor instead of the MapBasic text editor. For details, see [Using the Development Environment on page 56](#).
- The MapBasic compiler. After you have written a program, compile it to produce an “executable” application (i.e., an application that can be run by MapInfo).
- The MapBasic linker. If you are creating a large, complex application, you can divide your program into separate modules, then “link” those modules together into one application.
- MapBasic online help, providing reference information for each statement and function in the MapBasic language.
- From looking at the name, you might expect the MapBasic programming language to be reminiscent of traditional BASIC languages. In fact, MapBasic programs do not look much like traditional BASIC programs. MapBasic does, however, bear a resemblance to newer versions of BASIC which have been developed in recent years (for example, Microsoft’s Visual Basic language). Newer BASICs, such as Visual Basic and MapBasic, resemble Pascal more than traditional BASIC.

A Traditional BASIC Code Sample	A MapBasic Code Sample
20 GOSUB 3000	Call Check_Status(quit_time)
30 IF DONE = 1 THEN GOTO 90	Do While Not quit_time
40 FOR X = 1 TO 10	For x = 1 To 10
50 GOSUB 4000	Call Process_batch(x)
60 NEXT X	Next
80 GOTO 30	Loop

Every MapBasic program works in conjunction with MapInfo. First, you use the MapBasic development environment to create and compile your program; then you run MapInfo when you want to run your program. Thus, a MapBasic program is not a stand-alone program; it can only run when MapInfo is running. You could say that a MapBasic program runs on top of MapInfo.

However, MapBasic is not merely a macro language, MapBasic is a full-featured programming language, with over 300 statements and functions. Furthermore, since MapBasic programs run on top of MapInfo, MapBasic is able to take advantage of all of MapInfo’s geographic data-management capabilities.

How Do I Create and Run a MapBasic Application?

Chapter 4: Using the Development Environment provides detailed instructions on creating a MapBasic application.

If you're in a hurry to get started, you can create your first program by following these steps:

1. Run the MapBasic development environment.
2. Choose **FILE > NEW** to open an edit window.
3. Type a MapBasic program into the edit window. If you do not have a program in mind, you can enter the following one-line MapBasic program:
4. Note "Welcome to MapBasic!"
5. Choose **FILE > SAVE** to save the program to a file. Enter a file name such as `welcome.mb`.

Note: Do not close the Edit window.

6. Choose **PROJECT > COMPILE CURRENT FILE**. MapBasic compiles your program (`welcome.mb`), and then creates a corresponding executable application file (`welcome.mbx`).
7. Run MapInfo.
8. Choose **TOOLS > RUN MAPBASIC PROGRAM**. MapInfo prompts you to choose the program you want to run. If you select `welcome.mbx`, MapInfo runs your program, which displays the message, "Welcome to MapBasic!" in a dialog box.

Those are the main steps involved in creating, compiling, and running a MapBasic application. In practice, of course, the process is more complex. For example, the procedure outlined above does not describe what happens if you encounter a compilation error. For more details on creating and compiling MapBasic programs, see **Chapter 4: Using the Development Environment**.

What Are the Key Features of MapBasic?

MapBasic Lets You Customize MapInfo

Through MapBasic, you can customize the MapInfo user-interface. A MapBasic application can modify or replace the standard MapInfo menus, add entirely new menus to the MapInfo menu bar, and present the user with dialogs custom-tailored to the task at hand.

Thus, MapBasic lets you create *turn-key systems*, custom-tailored systems that help the user perform tasks quickly and easily, with minimal training.

MapBasic Lets You Automate MapInfo

MapBasic applications are often used to spare end-users the tedium of doing time-consuming manual work. For example, a MapInfo user may need to develop a *graticule* (a grid of horizontal and vertical longitude and latitude lines) in the course of producing a map. Drawing a graticule by hand is tedious, because every line in the graticule must be drawn at a precise latitude or longitude. However, a MapBasic application can make it very easy to produce a graticule with little or no manual effort.

MapBasic Provides Powerful Database-Access Tools

You can perform complex, sophisticated database queries with a single MapBasic statement. For example, by issuing a MapBasic **Select** statement (which is modeled after the Select statement in the SQL query language), you can query a database, apply a filter to screen out any unwanted records, sort and sub-total the query results. All of this can be accomplished *with a single MapBasic statement*.

Using powerful MapBasic statements like **Select** and **Update**, you can accomplish in a few lines of code what might take dozens or even hundreds of lines of code using another programming language.

MapBasic Lets You Connect MapInfo To Other Applications

You are not limited to the statements and functions that are built into the MapBasic programming language. Because MapBasic provides open architecture, your programs can call routines in external libraries. If you need functionality that isn't built into the standard MapBasic command set, MapBasic's open architecture lets you get the job done.

MapBasic programs can use Dynamic Data Exchange (DDE) to communicate with other software packages, including Visual Basic applications. MapBasic programs also can call routines in Windows Dynamic Link Library (DLL) files. You can obtain DLL files from commercial sources, or you can write your own DLL files using programming languages such as C or Pascal. MapBasic provides Integrated Mapping, that lets you integrate MapInfo functionality into applications written using other development environments, such as Visual Basic. For details see **Chapter 12: Integrated Mapping**.

How Do I Learn MapBasic?

If you have not already done so, you should learn how to use MapInfo before you begin working with MapBasic. This manual assumes that you are familiar with MapInfo concepts and terminology, such as *tables*, *Map windows*, and *workspaces*.

Once you are comfortable using MapInfo, you can use the following printed and online instructional materials to help you learn about MapBasic.

MapBasic User Guide

This book explains the concepts behind MapBasic programming. Read the *User Guide* when you are learning how to program in MapBasic.

Each chapter in the *User Guide* discusses a different area of programming. For example, **Chapter 7: Creating the User Interface** explains how to create a user interface (custom menus and dialog boxes), while **Chapter 9: File Input/Output** tells you how to perform file input/output. Every MapBasic programmer should read **Chapter 5: MapBasic Fundamentals**.

MapBasic Reference

This A-to-Z reference contains detailed information about every statement and function in the MapBasic language. Use the *Reference* when you need a complete description of a particular statement or function.

Sample Programs

Many programmers find that the best way to learn a programming language is to study sample programs. Accordingly, MapBasic comes with a library of sample programs. See the Samples folder installed on your MapBasic CD for sample programs included with MapBasic.

Note: The MapBasic *User Guide* frequently refers to the TextBox sample program (textbox.mb). You may want to become familiar with this program before you learn MapBasic. See Appendix B for a listing of the TextBox program.

MapInfo Workspace Files

MapInfo can save session information (for example, the list of what tables and windows are open) in a workspace file. If you use a text editor to examine a workspace file, you will see that the workspace contains MapBasic statements. You can copy MapBasic statements out of a workspace file, and paste the statements into your program. In a sense, any MapInfo workspace is a sample MapBasic program.

For example, suppose you want to write a MapBasic program that creates an elaborate page layout. You could create the page layout interactively, using MapInfo, and save the layout in a MapInfo workspace file. The workspace file would contain a set of MapBasic statements relating to page layouts. You then could copy the layout-related statements from the workspace file, and paste the statements into your MapBasic program.

Online Help

The MapBasic development environment provides extensive online Help. Much of the online Help is reference information, providing descriptions of every statement and function in the language. The Help file also provides instructions on using the MapBasic development environment.

Tip: as you are typing in your program, if you select a statement or function name and press **F1**, the Help window shows you help for that statement or function.

The Help system contains many brief sample programs which you can copy from the Help window and paste into your program. You can copy text out of the Help window by clicking and dragging within the Help window.

If you are viewing a Help screen and you click on a MapBasic menu or a MapBasic edit window, the Help window disappears. This is standard behavior for Windows Help. The Help window has not been closed, it is simply in the background. Note that you can return to the Help window by pressing **ALT-TAB**. You can also prevent the Help window from disappearing by checking the Help window's **HELP > ALWAYS** on Top menu item.

The MapBasic Window in MapInfo

The MapInfo software provides a feature known as the MapBasic window. This window can help you learn the syntax of statements in the MapBasic language.

To open the MapBasic window:

1. Run MapInfo
2. Choose Options > Show MapBasic Window.

The MapBasic window appears on the screen. Thereafter, as you use MapInfo's menus and dialogs, the MapBasic window displays corresponding MapBasic statements.

For example, if you perform a query by using MapInfo's Select dialog, the MapBasic window automatically shows you how you could perform the same operation through statements in the MapBasic language.

You can also enter statements directly into the MapBasic window, although not all MapBasic statements may be executed in this manner. To determine if a statement may be issued through the MapBasic window, consult the MapBasic *Reference*. Statements that are not supported through the MapBasic window are identified by a notice that appears under the **Restrictions** heading. As a general rule, you cannot enter flow-control statements (for example, **For...Next** loops) through the MapBasic window.

The MapBasic window is also a debugging tool. For details, see [Chapter 6: Debugging and Trapping Runtime Errors](#).

Training and On-Site Consulting

MapInfo Corporation offers MapBasic training classes. If you want to become proficient in MapBasic as quickly as possible, you may want to attend MapBasic training. To ensure an ideal training environment, class size is limited to eight to ten people. For information on scheduled classes, call MapInfo Professional Services.

If you require extensive assistance in developing your MapBasic application, you may be interested in MapInfo's Consulting Services. You can arrange to have MapBasic systems engineers work on-site with you. For additional information, call MapInfo Professional Services.

Using the Development Environment

The MapBasic software includes a text editor you can use to type your program. Conventional menu items (for example, Undo, Copy, Paste) make it easy to edit your program. Other menu items let you compile (and, optionally, link) your program(s) into executable form. Online help for the MapBasic language is available as well.

The MapBasic text editor, MapBasic compiler, and MapBasic online help are collectively known as the *development environment*.

Sections in this Chapter:

- ♦ Introduction to MapBasic Development Environment 57
- ♦ Editing Your Program 57
- ♦ Compiling Your Program 60
- ♦ Linking Multiple Modules Into a Single Project 62
- ♦ Menu Summary in MapBasic Development Environment . 66

Introduction to MapBasic Development Environment

The MapBasic development environment contains a built-in text editor that you can use to create and edit MapBasic programs. Pull-down menus — **File**, **Edit**, **Search**, **Project**, **Window**, and **Help** — provide you with everything you need to create and edit programs, compile them, and handle any syntax errors detected by the MapBasic compiler.

If you are familiar with other text editors, you will find MapBasic's text editor easy to use. Most of the MapBasic menus are predictable: the **File** menu contains **Open**, **Close**, **Print**, and **Save** commands, while the **Edit** menu contains **Undo**, **Cut**, **Copy**, and **Paste** commands. However, MapBasic also contains elements not found in conventional text editors (for example, a compiler and a linker).

Editing Your Program

If you have not already done so, run MapBasic. Then, from the **File** menu, either choose **Open** (to display an existing program) or **New** (to open a blank edit window).

Type your program into the edit window. If you don't yet have a program to type in, you can use the following one-line sample MapBasic program:

```
Note "Welcome to MapBasic!"
```

Once you have typed in your program, you can save your program to disk by choosing **Save** from the **File** menu. Give your program a name such as **welcome.mb**.

MapBasic automatically appends the file extension **.mb** to program files. Thus, if you name your program **welcome**, the actual file name is **welcome.mb**.

Since MapBasic saves your program in a conventional text file, you can use other text editing software to edit your program if you wish.

Keyboard Shortcuts

The following table lists the keyboard shortcuts you can use within the MapBasic edit window.

Keyboard Action	Effect of Action
Home / End	Insertion point moves to beginning/end of line
Ctrl-Home/ Ctrl-End	Insertion point moves to beginning/end of document
Ctrl-TAB/ Ctrl-Shift-TAB	Insertion point moves backward/forward one word
Ctrl-T	Displays the Go To Line dialog box
Ctrl-O	Displays the Open dialog box
Ctrl-N	Opens a new, empty edit window

Keyboard Action	Effect of Action
Ctrl-S	Saves the active edit window
Ctrl-P	Prints the active edit window
Ctrl-A	Selects all text in the edit window
Ctrl-C	Copies selected text to the clipboard
Ctrl-X	Cuts selected text and copies it to the clipboard
Ctrl-V	Pastes text from the clipboard into the edit window
Ctrl-Del	Deletes the word after the insertion point
Del	Deletes selected text; does not copy to clipboard
Ctrl-F	Displays the Find And Replace dialog box
Ctrl-G	Repeats the most recent Find command
Ctrl-R	Replaces the selected text (using the replacement text from the Find And Replace dialog box), and performs another Find
Ctrl-J	Displays Select Project File dialog
Ctrl-K	Compiles the program in the active window
Ctrl-E	Next Error command; scrolls the edit window to show the line that caused a compilation error
Ctrl-L	Links the active project
Ctrl-U	Sends message to MapInfo Professional to run the active program
F1	Displays Help.
F8	Displays Text Style dialog, allowing you to change the font
Ctrl-F4	Closes the active edit window
Alt-F4	Exits the MapBasic development environment
Shift-F4	Tile windows
Shift-F5	Cascade windows

Tip: If you select a function name before pressing **F1**, Help shows a topic describing that function.

Mouse Shortcuts

Mouse Action	Effect of Action
Double-click	Double-clicking on text within your program selects a word. Double-clicking in the list of error messages scrolls the window to show the line of your program that caused the error.
Triple-click	Highlights entire line of text (32-bit version only).
Drag & Drop	Dragging text to another window copies the text. Dragging text within the same window moves the text (unless you hold down the Ctrl key during the drag, in which case the text is copied).

Tip: The MapBasic online help contains code samples. You can drag & drop code samples from the help window to your edit window.

1. Display help.
2. Click and drag within the help window to highlight the text you want to copy.
3. Click on the text you highlighted. Without releasing the mouse button, drag the text out of the help window.
4. Move the mouse pointer over your edit window, and release the mouse button. The text is dropped into your program.

Limitations of the MapBasic Text Editor

Each MapBasic edit window can hold a limited amount of text. If the MapBasic text editor beeps when you try to insert text, the beeping indicates that the edit window is full.

There are three ways to work around this size limitation:

- If you have another text editor, you can use that editor to edit your program. To compile your program, switch to MapBasic and choose the Compile From File menu command.
- You can break your program file (.mb file) into two or more smaller files, and then use the MapBasic **Include** statement to incorporate the various files into a single application. For more information about the **Include** statement, see the MapBasic *Reference*.
- You can break your program file (.mb file) into two or more smaller files, and then create a MapBasic *project file* which links the various program files into a single application. In some ways, this is similar to using the **Include** statement to combine program modules. Project files, however, provide a more efficient solution. Each file included in a project can be compiled separately; this means that when you edit only one of your modules, you only need to recompile that module.

Compiling Your Program

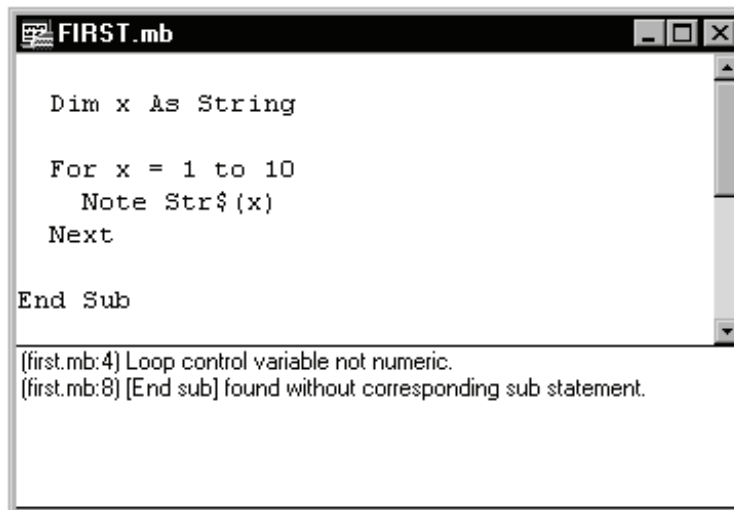
If you haven't already done so, display your program in a MapBasic edit window. Then, to compile your program, choose **Compile Current File** from the **Project** menu.

Note: You can have multiple edit windows open at one time. When you choose **Compile Current File**, MapBasic compiles the program that is in the front-most window. Thus, if you have multiple edit windows open, you must make the appropriate window active before you compile.

The MapBasic compiler checks the syntax of your program. If your program contains any syntax errors, MapBasic displays a dialog indicating that errors were found, and then displays descriptions of the errors in a list beneath the edit window.

Each error message begins with a line number, indicating which line in the program caused the error. You must correct your program's errors before MapBasic can successfully compile your program.

Figure: First.mb



If you double-click on an error message that appears beneath the edit window, MapBasic scrolls the window to show you the line of the program that caused the error.

After you correct any errors in your program, choose **Compile Current File** again to try to recompile. Once your program compiles successfully, MapBasic displays a dialog indicating that compilation was complete.

When compilation is successful, MapBasic creates an .mbx file (MapBasic eXecutable). This .mbx file must be present when the user actually runs the finished application. Thus, if you want to provide your users with a finished MapBasic application, but you do not want to give them all of your source code, give the users your .mbx file but not your .mb file.

A Note on Compilation Errors

There are some types of spelling errors which the MapBasic compiler cannot detect. For example, the MapBasic compiler will compile the following program, even though the program contains a typographical error on the second line (STATES is misspelled as TATES):

```
Open Table "states"  
Map From tates
```

The MapBasic compiler cannot identify the typographical error on the second line. This is not a defect of the compiler, rather, it is simply a result of the fact that some variable and table references are not evaluated until runtime (until the moment the user runs the program). When the user runs the preceding program, MapInfo Professional attempts to carry out the **Map From tates** statement. At that time, MapInfo Professional displays an error message (for example, "Table tates is not open") unless a table called **tates** is actually available.

Running a Compiled Application

To run the compiled application, choose Run MapBasic Program from MapInfo Professional's File menu. MapInfo Professional's Run MapBasic Program dialog prompts you to choose which MapBasic application file (.mbx file) to run.

The MapBasic development environment also provides a shortcut to running your program: After compiling your program, choose Run from MapBasic's Project menu (or press Ctrl-U). MapBasic sends a message to MapInfo Professional, telling MapInfo Professional to execute the application.

The MapBasic development environment also provides a shortcut to running your program: after compiling your program, you can choose Run from MapBasic's Project menu. MapBasic sends a message to MapInfo Professional, telling it to execute the application.

Note: MapInfo Professional must already be running.

Using Another Editor to Write MapBasic Programs

If you already have a favorite text editor, you can use that editor for editing your MapBasic program. Just save your MapBasic program as a standard text file.

You can also use word processing software to edit your programs. However, if you use a word processor to edit your programs, you may need to take special steps to make sure that the word processor saves your work in a plain text file format. Saving a document as plain text often involves choosing **Save As** instead of **Save**. For more details on saving a document in a plain text format, see the documentation for your word processing software.

Compiling Programs Written In Another Editor

Earlier, we discussed how MapBasic's Compile Current File menu item compiles whichever program is on the screen in the active edit window. MapBasic also provides an alternate method for compiling your program: the Compile From File command on MapBasic's File menu.

If you use a text editor other than MapBasic to edit your program, you probably will want to use Compile From File to compile your program. Compile From File compiles a program without displaying the program in a MapBasic edit window.

When you choose Compile From File, MapBasic prompts you to choose a file to compile. If the chosen file has any compilation errors, MapBasic writes the error messages to a text file with the .err extension. For example, if you choose Compile From File to compile the program dispatch.mb, MapBasic writes any error messages to the text file dispatch.err. To view the error file, choose File > Open.

Compiling and Linking Programs From the Command Line

If you use a text editor other than MapBasic to edit your programs, you may find it awkward switching to MapBasic whenever you want to compile or link your application. However, there is a way to automate the process of compiling and linking: if you can configure your text editor so that it issues a command string, then you can compile programs without leaving your editor.

You can start the MapBasic development environment by executing the command:

```
mapbasic
```

If the command line also includes the parameter **-D** followed by one or more program names, MapBasic automatically compiles the program files. For example, the following command line launches MapBasic and compiles two program files (main and sub1):

```
mapbasic -D main.mb sub1.mb
```

If the command line includes the parameter **-L** followed by one or more project file names, MapBasic links the projects. (Linking and Project files are discussed in the next section.) For example, the following command line links the TextBox application:

```
mapbasic -L tbproj.mbp
```

The command line can include both the **-D** and the **-L** parameters, as shown below:

```
mapbasic -D textbox.mb -L tbproj.mbp
```

If you launch MapBasic with a command line that includes the **-D** parameter or the **-L** parameter, MapBasic shuts down after compiling or linking the appropriate files.

To start MapBasic without displaying a splash screen use the **-Nosplash** parameter:

```
mapbasic -Nosplash
```

Linking Multiple Modules Into a Single Project

What is a MapBasic Project File?

A project file is a text file that allows MapBasic to link separate program files into one application. If you are developing a large, complex application, your program could eventually contain thousands of lines of code. You could type the entire program into a single program file. However, most programmers dislike managing program files that large; once a program file grows to over a thousand lines, it can be difficult to locate a particular part of the program. Therefore, many programmers break up large applications into two or more smaller files. The practice of breaking large programs down into smaller, more manageable pieces is known as *modular* programming.

If you do divide your program into two or more modules, you need to create a project file. The project file tells the MapBasic linker how to combine separate modules into a single, executable application.

Project files are an optional part of MapBasic programming. You can create, compile, and run applications without ever using a project file. However, if you plan to develop a large-scale MapBasic application, it is worth your while to take advantage of MapBasic's project-file capabilities.

What Are The Benefits of Using Project Files?

- Project files let you modularize your programming. Once you set up a project file, you can divide your program into numerous, small files. Modular programs are generally easier to maintain in the long run. Also, having modular programs makes it unlikely that your program will grow too large to be edited in a MapBasic edit window.
- Project files let you modularize your programming. Once you set up a project file, you can divide your program into numerous, small files. Modular programs are generally easier to maintain in the long run.
- Project files make it easy to have two or more programmers working on a project at the same time. Once you have set up a project file, each programmer can work on a separate module, and the modules can be joined (or, more specifically, "linked") by the project file.
- Project files can reduce the time it takes to recompile your application. If you change one module in a multiple-module project, you can recompile just that module, then relink the project. This is often much faster than recompiling all source code in the project—which is what you must do if you do not use project files.

Examples of Project Files

The TextBox application uses a project file (tbproj.mbp) that looks like this:

```
[Link]
Application=textbox.mbx
Module=textbox.mbo
Module=auto_lib.mbo
```

Similarly, the ScaleBar application uses a project file (sbproj.mbp) that looks like this:

```
[Link]
Application=scalebar.mbx
Module=scalebar.mbo
Module=auto_lib.mbo
```

In both examples, the final line of the project file tells MapBasic to build the auto_lib module into the project. The auto_lib module is one of the sample programs included with the MapBasic software.

If a MapBasic program includes the auto_lib module, the program can provide a special "Auto-Load..." button in its About dialog box. By choosing the Auto-Load button, the user can set up the application so that it loads automatically, every time the user runs MapInfo Professional. If the user does not turn on the Auto-Load feature, the MapBasic application stops running as soon as the user exits MapInfo Professional.

To build the Auto-Load feature into your MapBasic program, see the instructions listed in the file auto_lib.mb.

Creating a Project File

If you have already written a program file, and you want to create a project file for your program, follow these steps:

1. Choose File > New to open a new edit window.
2. Enter the following line in the edit window:
[Link]
3. Enter a line that contains the text **Application=***appfilename* (where *appfilename* specifies the file name of the executable file you want to create). For example:

```
Application=C:\MB\CODE\CUSTOM.MBX
Application=Local:MapBasic:custom.mbx
Application=/MapBasic/mb_code/custom.mbx
```

4. Enter a line that contains the text **Module=***modulename* (where *modulename* specifies the name of a MapBasic object file). For example:

```
Module=C:\MB\CODE\CUSTOM.MBO
Module=Local:MapBasic:custom.mbo
Module=/MapBasic/mb_code/custom.mbo
```

Note the extension on the filename; MapBasic *object files* have the file extension .mbo.

MapBasic creates an object file when you compile a single module that is part of a multiple-module project.

Whenever you choose Project > Compile Current File, MapBasic tries to compile the current file into an executable application file (ending with .mbx). However, if the program file contains calls to functions or procedures that are not in the file, MapBasic cannot create an .mbx file. In this case, MapBasic assumes that the program is part of a larger project. MapBasic then builds an object file (.mbo) instead of an executable file (.mbx). MapBasic also creates an object file whenever the module that you are compiling does not have a Main procedure.

5. Repeat step 4 for every file you wish to include in your application.
6. Choose File > Save As to save the project file.
In the Save As dialog, choose the file type "Project File" (from the list of file types in the lower left corner of the dialog), so that the file has the extension .mbp (MapBasic Project).
In the Save As dialog, choose the file type "Project File" (from the list of file types in the lower left corner of the dialog), so that the file has the extension .mbp (MapBasic Project).
7. Close the edit window (either choose File > Close or click on the window's close box).

If you add more modules to the project at a later date, remember to add appropriate "Module=" lines to the project file.

Compiling and Linking a Project

Once you have created a project file, you can compile and link your project by following these steps:

1. Compile each module that is used in the project. To compile a module, choose File > Open, then choose Project > Compile Current File. To compile a module without first displaying it, choose File > Compile From File.
2. Choose Project > Select Project File to tell MapBasic which project file you want to link. The Select Project File dialog displays. Choose the project (.mbp) file you want, and choose OK. The selected project file appears in an edit window. This file remains selected until you exit

MapBasic, close the project file's edit window, or choose the Project > Select Project File command again. Only one project file can be selected at any time.

Note: You cannot change which project file is selected by making an edit window the front-most window. You cannot change which project file is selected by choosing File > Open. To select the project file you want to link, choose Project > Select Project File.

3. Choose Project > Link Current Project to link your application. MapBasic reads the object (.mbo) files listed in the project file. If there are no link errors, MapBasic builds an executable (.mbx) file. If there are link errors, MapBasic displays an error message.

You also can link a project in a single step, without first displaying the project file in an edit window, by choosing File > Link From File.

The object files created by the MapBasic compiler cannot be linked using any other linker, such as a C-language linker. Only the MapBasic linker can link MapBasic object modules.

Opening Multiple Files

If you use project files, you may find that you sometimes need to open all of the program files in your project. To simplify this process, the Open dialog lets you open multiple files at the same time. To open multiple files at one time:

1. On the File menu, choose Open.
2. Click on a file name in the Open Program dialog box.
3. Hold down the Shift key or the Ctrl key as you click on another file name. Holding down the Shift key lets you select a list of adjacent files. Holding down the Ctrl key lets you add files to the selected set, one file at a time.

Calling Functions or Procedures From Other Modules

If a .MB file is part of a multiple-module project, it can call functions and sub procedures located in other modules. For example, textbox.mb calls the HandleInstallation procedure, which is located in the auto_lib library. Calling a function or sub procedure located in another module is known as an external reference.

If your MapBasic program calls an external procedure, your program file must contain a **Declare Sub** statement. Similarly, if your program calls an external function, your program file must contain a **Declare Function** statement. These **Declare** statements tell the MapBasic compiler what parameters are used by the procedure or function.

The sample program textbox.mb contains the statement **Include "auto_lib.def"**. The auto_lib.def definitions file contains a set of **Declare Sub** and **Declare Function** statements which correspond to the auto_lib module. If textbox.mb did not include the auto_lib.def definitions file, the MapBasic compiler would consider the call to the HandleInstallation procedure to be a syntax error ("Invalid sub procedure name").

Sharing Variables With Other Modules

To declare a global variable that can be used by two or more modules in a project:

1. Place **Global** statements in a definitions file (for example, "globals.def").
2. Use the **Include** statement to incorporate the definitions file into each module that needs to use the global variables.

For example, the `auto_lib.def` definitions file declares two global string variables, `gsAppFilename` and `gsAppDescription`. The `auto_lib.mb` program file and the `textbox.mb` program file both issue the statement:

```
Include "auto_lib.def"
```

Therefore, the two modules can share the global variables. When the `textbox.mb` program stores values in the global variables, the `auto_lib.mb` library is able to read the new values.

Global variables also allow you to share information with other applications that are running.

Declaring Variables That Cannot Be Shared With Other Modules

A program file can contain **Dim** statements that are located outside of any function or sub procedure definition. Such **Dim** statements are known as *module-level Dim* statements. If a variable is declared by a module-level **Dim** statement, all functions and procedures in that module (i.e., in that `.mb` file) can use that variable. However, a MapBasic file cannot reference another file's module-level Dims.

Use module-level **Dim** statements if you want to declare a variable that can be shared by all procedures in a file, but you want to be sure that you don't accidentally use a variable name that is already in use in another module.

Menu Summary in MapBasic Development Environment

The File Menu

The **File** menu provides commands that let you create, open, close, save, exit, and print MapBasic programs.

- **New** opens a new edit window where you can type in your program.
- **Open** displays an existing file in an edit window. The file can be a MapBasic program file (for example, `dispatch.mb`), a list of error messages (`dispatch.err`), or a MapInfo Professional workspace file. Each workspace is actually just a text file containing an assortment of MapBasic statements.

The Open dialog lets you open two or more files at the same time. To select multiple files, hold down the Shift key or the Ctrl key as you click on the file names.

Note: Some text files are too big to be displayed in a MapBasic edit window. For information on bypassing this limitation, see *Limitations of the MapBasic Text Editor* (above).

- **Close** closes the active edit window. If you have made changes in the current window, MapBasic prompts you to either save or discard the changes before closing the window. Close is available when at least one edit window is open.
- **Close All** closes all open edit windows. As with the Close command, MapBasic prompts you to either save or discard any unsaved changes. Close All is available when at least one edit window is open.
- **Save** saves the contents of the active edit window to disk. Save is available when you have changed the contents of an edit window.
- **Save As** saves the contents of the active edit window under a new file name. Save As is available when you have an open edit window.

- **Revert** discards any changes made to the edit window since it was last saved. Revert is available when you have changed the contents of an edit window.
- **Compile From File** compiles an existing .mb file directly from the contents of the disk file, without first displaying the contents of the file in an edit window. (As opposed to the Compile Current File command on the Project menu, which compiles whatever program is in the active edit window.) Use Compile From File to compile a program written in another text editor. If there are compilation errors, Compile From File writes error messages to a text file named *filename.err*. To view the errors file, choose **File > Open**.
- **Link From File** links an existing project without first displaying the contents of the project file in an edit window. (As opposed to the Link Current Project command on the Project menu, which links the current project.)
- **Page Setup** defines printer options (for example, print margins).
- **Page Setup** defines printer options (for example, paper size and orientation).
- **Printer Setup** defines printer options (for example, which print command to use).
- **Print** prints the active edit window.
Print is available when there is at least one Edit window open.
- **Exit** exits the MapBasic environment. MapBasic prompts you to either save or discard any changes that have not been saved.
- **Quit** exits the MapBasic environment. MapBasic prompts you to either save or discard any changes that have not been saved.

The Edit Menu

The **Edit** menu provides commands that you can use when drafting and editing your MapBasic program.

- **Undo** cancels the most recent change you made in the active edit window. When you select Undo, MapBasic discards the last change you performed, and then the menu item changes to read **Redo**. If you select Redo, MapBasic then re-applies the discarded change.
Undo is enabled when there is at least one open edit window, and you have made changes to the text in that window.
- **Cut** copies the selected (highlighted) text to the Clipboard, then removes the selected text from the edit window. The text remains on the Clipboard and you can later insert it elsewhere through the **Paste** command (see below).
Cut is available when text is selected in the active edit window.
Copy copies the selected text to the Clipboard, but does not delete it.
Copy is available when text is selected in the active edit window.
Paste copies the contents of the Clipboard to the active edit window at the current cursor location. If you select text in the edit window, and then perform Paste, the text from the clipboard replaces the selected text.
Paste is available when text is in the Clipboard and there is at least one open edit window.
- **Clear** deletes selected text without copying it to the Clipboard.
Clear is available when there is selected text in an open edit window.
Select All selects the entire contents of the active edit window. Select All is available when there is at least one open edit window.

The Search Menu

The **Search** menu helps you to locate and replace text in the edit window. Some of these commands simplify the process of locating statements that have syntax errors.

- **Find** searches the active edit window for a particular text string. Find is available when there is at least one open edit window.

To find the next occurrence of a text string: Type the text string you want to find into the Find box. If you want the search to be case-sensitive, check the Match Case check box.

When you click on the Find button, MapBasic searches forward from the current insertion point. If MapBasic finds an occurrence of the Find string, the window scrolls to show that occurrence. If the text is not found, MapBasic beeps.

To replace all occurrences of a text string:

- Type the replacement string in the Replace With box, and click the Replace All button. MapBasic replaces all occurrences of the Find string with the Replace With string.

Note: This replacement happens instantly, with no confirmation prompt.

To confirm each string replacement:

1. Choose Search > Find. The Find dialog appears.
2. Fill in the Find and Replace With text boxes.
3. Within the Find dialog, click the Find button.

MapBasic finds and highlights the next occurrence of the text string.

If you want to replace the currently-highlighted string, press Ctrl-R (the hot-key for the Replace And Find Again menu command).

If you do not want to replace the currently-highlighted occurrence of the Find string, press Ctrl-G (the hot-key for the Find Again menu command).

If you want to replace the currently-highlighted string, press Command-R (which is the hot-key for the Replace and Find Again command).

If you do not want to replace the currently-highlighted occurrence of the Find string, press Command-G, the hot-key for the Find Again menu item.

If you want to replace the currently-highlighted string, press Ctrl-R (which is the hot-key for the Replace and Find Again command).

If you do not want to replace the currently-highlighted occurrence of the Find string, press Ctrl-G, the hot-key for the Find Again menu item.

- **Find Again** finds the next occurrence of the string specified in the previous Find dialog. Find Again is available when there is at least one open edit window, and a Find operation has been performed.
- **Replace And Find Again** replaces the selected text with text specified in the Find dialog, then finds and highlights the next occurrence of the search string

Next Error is a feature of the compiler that helps you correct syntax errors. When a program does not compile correctly, MapBasic displays a list of the errors at the bottom of the edit window. **Next Error** scrolls forward through the edit window, to the line in your program which corresponds to the next error in the error list.

Next Error is available when there are error messages in the active edit window.

Previous Error is similar to Next Error. Previous Error scrolls backward through the edit window to the previous item in the error list. Previous Error is available when there are error messages relating to the active edit window.

Go To Line prompts you to type in a line number, then scrolls through the edit window to that line in your program.

A program may compile successfully, yet it may encounter an error at runtime. When this happens, a dialog appears, indicating that an error occurred at a certain line in your program. Typically, you then want to return to the MapBasic development environment and go to the appropriate line of your program. Go To Line is available when there is at least one edit window open.

The Project Menu

The **Project** menu lets you compile and run MapBasic programs, display program statistics, and show or hide the error window.

- **Select Project File** presents a dialog which lets you open an existing project file. A project file is a text file that lists all the modules that comprise your application. Once you select a project file, that project file becomes the active project file, and you can compile the file by choosing **Link Current Project**.

Compile Current File compiles the program in the active edit window. Compile is available if there is at least one open edit window.

If the compiler detects syntax errors in the program, MapBasic displays a list of errors at the bottom of the edit window. If there are no syntax errors, MapBasic builds an mbx file (if the module is a stand-alone program) or an object module (mbo) file.

- **Link Current Project** links the modules listed in the current project file, and produces an executable application file (unless there are errors, in which case an error message displays). Link Current Project is available whenever a project file is open.
- **Run** sends a message to the MapInfo Professional software, telling it to execute the application in the front-most edit window.
- **Get Info** displays statistics about the program in the active edit window. Get Info is available if there is at least one open edit window.
- **Show/Hide Error List** activates or deactivates the error list associated with the active edit window. If the error list is currently displayed, the menu item reads Hide Error List. If the error list is currently hidden, the menu item reads Show Error List. Show/Hide Error List is available when there is an open edit window with associated error messages.

The Window Menu

If you have more than one edit window open, MapBasic's **Window** menu lets you arrange your windows or switch which window is active.

Commands on this menu are available when there is at least one edit window open.

- **Tile Windows** arranges the edit windows in a side-by-side pattern.
- **Cascade Windows** arranges the edit windows in an overlapping pattern.
- **Arrange Icons** organizes the icons that correspond to your minimized edit windows. You can click an edit window's minimize button to temporarily shrink that window down to an icon.
- **Text Style** lets you choose the font in which the window is displayed. The font you choose is applied to the entire window.
- The bottom of the Window menu lists a menu item for each open edit window. To make one of the edit windows active (i.e., to bring that window to the front), select the appropriate item from the Window menu.

The Help Menu

Use the Help menu to access online help. The online help file contains descriptions of all statements and functions in the MapBasic language. Help also includes a comprehensive set of cross-reference screens to help you find the name of the statement you need.

- **Contents** opens the help window at the Contents screen. From there, you can navigate through help by clicking on hypertext jumps, or you can click on the Search button to display the Search dialog.
- **Search For Help On** jumps directly to the Search dialog.
- **How To Use Help** displays a help screen that explains how to use online help.
- **About MapBasic** displays the About dialog, which shows you copyright and version number information.

Note: Many of the help screens contain brief sample programs. You can copy those program fragments onto the clipboard, then paste them into your program. To copy text from a help screen, choose Edit > Copy from the help window's Edit menu or by dragging text directly out of the help window, and drop it into your program.

MapBasic Fundamentals

Every MapBasic programmer should read this chapter, which describes many fundamental aspects of the MapBasic programming syntax.

Sections in this Chapter:

- ♦ General Notes on MapBasic Syntax 72
- ♦ Expressions 78
- ♦ Looping, Branching, and Other Flow-Control 87
- ♦ Procedures 92
- ♦ Procedures That Act As System Event Handlers 95
- ♦ Tips for Handler Procedures 98
- ♦ Compiler Instructions 100
- ♦ Program Organization 102

General Notes on MapBasic Syntax

Before getting into discussions of specific MapBasic statements, it is appropriate to make some observations about MapBasic program syntax in general.

Comments

In MapBasic, as in some other BASIC languages, the apostrophe character (') signifies the beginning of a comment. When an apostrophe appears in a program, MapBasic treats the remainder of the line as a comment, unless the apostrophe appears within a quoted string constant.

Case-Sensitivity

The MapBasic compiler is case-insensitive. You can enter programs with UPPER-CASE, lower-case, or Mixed-Case capitalization.

For clarity, this manual capitalizes the first letter of each MapBasic language keyword. Program variables appear in lower-case. For example, in the following program sample, the words **If** and **Then** have proper capitalization because they are keywords in MapBasic, whereas the word **counter** appears in lower-case, because it is the name of a variable.

```
If counter > 5 Then
    Note "Count is too high"
End If
```

Continuing a Statement Across Multiple Lines

When you write a MapBasic program, you can continue longer statements across more than one line. For example, the following code sample continues the **If...Then** statement across several lines:

```
If counter = 55
    Or counter = 34 Then
    Note "Counter is invalid"
End If
```

Codes Defined In mapbasic.def

Many MapBasic statements and function calls will not work properly unless the following statement appears at or near the top of your program:

```
Include "mapbasic.def"
```

The file mapbasic.def is a text file containing definitions for many standard MapBasic codes. As a rule, the codes defined in mapbasic.def are all in upper-case (for example, TRUE, FALSE, BLACK, WHITE, CMD_INFO_X, OBJ_INFO_TYPE, etc.). As you read the program examples that appear in the MapBasic documentation, you will see many such codes. For example:

```
If CommandInfo( CMD_INFO_DLG_OK ) Then
```


If your program references standard codes (such as `CMD_INFO_DLG_OK` in the example above), your program must issue an **Include** statement to include `mapbasic.def`. If you omit the **Include** statement, your program will generate a runtime error (for example, “Variable or Field `CMD_INFO_DLG_OK` not defined”).

Typing Statements Into the MapBasic Window

The MapInfo Professional software has a feature known as the MapBasic Window. Typing statements directly into the MapBasic Window helps you to learn MapBasic statement syntax. However, some restrictions apply to the MapBasic window:

- Some MapBasic statements may not be entered through the MapBasic window, although you may use those statements within compiled MapBasic programs. The general rule is: flow-control statements (such as **If...Then**, **For...Next**, and **GoTo**) do not work in the MapBasic window.
- To determine whether you can type a particular statement into the MapBasic window, see the MapBasic *Reference* or online Help. If a statement does not work in the MapBasic window, that statement's entry in the *Reference* indicates the restriction.
- When you type statements directly into MapInfo Professional's MapBasic Window, you must take special steps if you want to continue the statement across multiple lines. At the end of the each partial line, type Ctrl-Enter instead of Enter. After you have typed the entire statement, highlight the lines that make up the statement, and press Enter.
- Codes that are defined in `mapbasic.def` (for example, `BLACK`, `WHITE`, etc.) may not be entered in the MapBasic window. However, each code has a specific value, which you can determine by reading `mapbasic.def`; for example, the code `BLACK` has a numerical value of zero. When you are entering commands into the MapBasic window, you must use the actual value of each code, instead of using the name of the code (for example, use zero instead of “BLACK”).
- Each statement that you type into the MapBasic window is limited to 256 characters.

Variables

MapBasic's syntax for declaring and assigning values to variables is much like the syntax of other modern BASIC languages. However, MapBasic supports some types of variables that are not available in other languages (such as the Object variable; for a complete list of MapBasic variable types, see the description of the **Dim** statement in the MapBasic *Reference*).

What Is a Variable?

Think of a variable as a very small piece of your computer's memory. As you write programs, you will find that you need to temporarily store various types of information in memory. To do this, you declare one or more variables. Each variable has a unique name (for example, `counter`, `x`, `y2`, `customer_name`). For each variable that you declare, MapBasic sets aside a small piece of memory. Thereafter, each variable can contain one small piece of information.

Declaring Variables and Assigning Values to Variables

The **Dim** statement defines variables. You must declare every variable that you use, and the variable declaration must appear before the variable is used.

Use the equal operator (=) to assign a value to a variable.

The following example declares an Integer variable and assigns a value of 23 to that variable:

```
Dim counter As Integer
counter = 23
```

A single **Dim** statement can declare multiple variables, provided that the variable names are separated by commas. The following **Dim** statement declares three floating-point numeric variables:

```
Dim total_distance, longitude, latitude As Float
longitude = -73.55
latitude = 42.917
```

A single **Dim** statement can declare variables of different types. The following statement declares two Date variables and two String variables:

```
Dim start_date, end_date As Date,
first_name, last_name As String
```

Variable Names

Variable names must conform to the following rules:

- Each variable name can be up to thirty-one characters long.
- Variable names may not contain spaces.
- Each variable name must begin with a letter, an underscore (_) or a tilde (~).
- Each variable name can consist of letters, numbers, pound signs (#), or underscore characters (_).
- A variable name may end in one of the following characters: \$, %, &, !, or @. In some BASIC languages, these characters dictate variable types. In MapBasic, however, these characters have no special significance.
- You may not use a MapBasic keyword as a variable name. Thus, you may not declare variables with names such as **If**, **Then**, **Select**, **Open**, **Close**, or **Count**. For a list of reserved keywords, see the discussion of the **Dim** statement in the *MapBasic Reference*.

Data Types

MapBasic supports the following types of variables:

Type	Description
SmallInt	Integer value between -32767 and 32767; stored in two bytes
Integer	Integer value between -2 billion and 2 billion; stored in four bytes
Float	Floating-point value; stored in eight-byte IEEE format
String	Variable-length character string, up to 32,767 characters long
String * n	Fixed-length character string, n characters long (up to 32,767 characters)

Type	Description
Logical	True or False
Date	Date
Object	Graphical object, such as a line or a circle; see Chapter 10: Graphical Objects for details
Alias	Column reference of a table; see Chapter 8: Working With Tables for details
Pen	Pen (line) style setting; see Chapter 10: Graphical Objects
Brush	Brush (fill) style setting; see Chapter 10: Graphical Objects

Fixed-length and variable-length String variables

MapBasic supports both fixed-length and variable-length String variables. A variable-length String variable can store any string value, up to 32,767 characters long. A fixed-length String variable, however, has a specific length limit, which you specify in the **Dim** statement.

To declare a variable-length String variable, use String as the variable type. To declare a fixed-length String variable, follow the String keyword with an asterisk (*), followed by the length of the string in bytes. In the following example, `full_name` is declared as a variable-length String variable, while `employee_id` is declared as a fixed-length String variable, nine characters long:

```
Dim full_name As String,
    employee_id As String * 9
```

Note: Like other BASIC languages, MapBasic automatically pads every fixed-length String variable with blanks, so that the variable always fills the allotted space. Thus, if you declare a fixed-length String variable with a size of five characters, and then you assign the string "ABC" to the variable, the variable will actually contain the string "ABC" ("ABC" followed by two spaces). This feature is helpful if you need to write an application that produces formatted output.

Array Variables

To declare an array variable, follow the variable name with the size of the array enclosed in parentheses. The array size must be a positive integer constant expression. The following **Dim** statement declares an array of ten Date variables:

```
Dim start_date(10) As Date
```

To refer to an individual element of an array, use the syntax:

```
array_name(element-number)
```

Thus, the following statement assigns a value to the first element of the `start_date` array:

```
start_date(1) = "6/11/93"
```

To resize an array, use the **ReDim** statement. Thus, in cases where you do not know in advance how much data your program will need to manage—perhaps because you do not know how much data the user will enter—your program can use the **ReDim** statement to enlarge the array as needed. Use the **UBound()** function to determine the current size of an array.

The following example declares an array of String variables called `name_list`. The latter part of the program increases the size of the array by ten elements.

```
Dim counter As Integer, name_list(5) As String
...
counter = UBound(names)      ' Determine current array size
ReDim names(counter + 10)    ' Increase array size by 10
```

MapBasic arrays are subject to the following rules:

- MapBasic supports only one-dimensional arrays.
- In MapBasic, the first element in an array always has an index of one. In other words, in the example above, the first element of the names array is `names(1)`.

If you need to store more data than will fit in an array, you may want to store your data in a table. For more information on using tables, see [Chapter 8: Working With Tables](#).

MapBasic initializes the contents of numeric arrays and variables to zero when they are defined. The contents of string arrays and variables are initially set to the null string.

Custom Data Types (Data Structures)

Use the **Type...End Type** statement to define a custom data type. A custom data type is a grouping of one or more variables types. Once you define a custom data type, you can declare variables of that type by using the **Dim** statement.

The following program defines a custom data type, `employee`, then declares variables of the `employee` type.

```
Type employee
    name As String
    title As String
    id As Integer
End Type
Dim manager, staff(10) As employee
```

Each component of a custom data type is referred to as an *element*. Thus, the `employee` data type in the preceding example has three elements: `name`, `title`, and `id`. To refer to an individual element of an array, use the generic syntax:

```
variable_name.element_name
```

The following statement assigns values to each element of the `manager` variable:

```
manager.name = "Joe"
manager.title = "Director of Publications"
manager.id = 111223333
```

You can declare an array of variables of a custom type. The following statement assigns values to some of the elements of the first item in the `employee` array:

```
staff(1).name = "Ed"
staff(1).title = "Programmer"
```

Type...End Type statements must appear outside of any sub procedure definition. Sub procedures are discussed later in this chapter. Typically, **Type...End Type** statements appear at or near the very top of your program. A **Type** definition may include elements of any other type, including previously-defined custom data types. You can also declare global variables and arrays of custom data types.

Global Variables

Variables declared with the **Dim** statement are *local* variables. A local variable may only be used within the procedure where it is defined. MapBasic also lets you declare global variables, which may be referenced within any procedure, anywhere in the program.

To declare a global variable, use the **Global** statement. The syntax for the **Global** statement is identical to the syntax for the **Dim** statement, except that the keyword **Global** appears instead of the keyword **Dim**. Thus, the following **Global** statement declares a pair of global Integer variables:

```
Global first_row, last_row As Integer
```

Global statements must appear outside of any sub procedure definition. Sub procedures are discussed later in this chapter. Typically, **Global** statements appear at or near the top of the program.

The following program declares several global variables, then references those global variables within a sub procedure.

```
Declare Sub Main
Declare Sub initialize_globals
Global gx, gy As Float      ' Declare global Float variables
Global start_date As Date  ' Declare global Date variable
Sub Main
    Dim x, y, z As Float    ' Declare Main proc's local vars
    Call initialize_globals
    ...
End Sub
Sub initialize_globals
    gx = -1                 ' Assign global var: GX
    gy = -1                 ' Assign global var: GY
    start_date = CurDate()  ' Assign global var: START_DATE
End Sub
```

Whenever possible, you should try to use local variables instead of global variables, because each global variable occupies memory for the entire time that your program is running. A local variable, however, only occupies memory while MapBasic is executing the sub procedure where the local variable is defined.

MapBasic global variables can be used to exchange data with other software packages. When an application runs on Windows, other applications can use Dynamic Data Exchange to read and modify the values of MapBasic global variables.

Scope of Variables

A sub procedure may declare a local variable which has the same name as a global variable. Thus, even if a program has a global variable called `counter`, a sub procedure in that program may also have a local variable called `counter`:

```
Declare Sub Main
Declare Sub setup
Global counter As Integer
...
Sub setup
  Dim counter As Integer
  counter = 0
...
End Sub
```

If a local variable has the same name as a global variable, then the sub procedure will not be able to read or modify the global variable. Within the sub procedure, any references to the variable will affect only the local variable. Thus, in the example above, the statement: `counter = 0` has no effect on the global `counter` variable.

Upon encountering a reference to a variable name, MapBasic attempts to interpret the reference as the name of a local variable. If there is no local variable by that name, MapBasic attempts to interpret the reference as the name of a global variable. If there is no global variable by that name, MapBasic tries to interpret the reference as a reference to an open table. Finally, if, at runtime, the reference cannot be interpreted as a table reference, MapBasic generates an error message.

Expressions

In this section, we take a closer look at expressions. An expression is a grouping of one or more variables, constant values, function calls, table references, and operators.

What is a Constant?

An expression can be very simple. For example, the following statement:

```
counter = 23
```

assigns a simple integer expression namely, the value 23 to the variable, `counter`. We refer to the expression 23 as a *numeric constant*. You might think of a constant as a specific value you can assign to a variable.

The following program declares a String variable, then assigns a *string constant* (the name “Brian Nichols”) to the variable:

```
Dim name As String
name = "Brian Nichols"
```

The syntax for numeric expressions is different than the syntax for string expressions: string constants must be enclosed in double-quotation marks (for example, “Brian Nichols”) whereas numeric constants (for example, 23) are not. You cannot assign a String expression, such as “Brian Nichols,” to a numeric variable. For more information on constant expressions, see *A Closer Look at Constants*.

What is an Operator?

An operator is a special character (for example, +, *, >) or a word (for example, And, Or, Not) which acts upon one or more constants, variables, or other values. An expression can consist of two or more values that are combined through an operator. In the following example, the plus operator (+) is used within the expression `y + z`, to perform addition. The result of the addition (the sum) is then assigned to the variable, `x`:

```
Dim x, y, z As Float
y = 1.5
z = 2.7
x = y + z
```

In this example, the plus sign (+) acts as an *operator* - specifically, a *numeric operator*. Other numeric operators include the minus operator (-), which performs subtraction; the asterisk (*), which performs multiplication; and the caret (^), which performs exponentiation. A complete list of numeric operators appears later in this chapter.

The plus operator can also be used within a String expression to concatenate separate strings into one string. The following program builds a three-part string expression and stores the string in the variable, `full_name`:

```
Dim first_name, last_name, middle_init, full_name As String
first_name = "Brian "
middle_init = "R. "
last_name = "Nichols"
full_name = first_name + middle_init + last_name

' At this point, the variable full_name contains:
'   Brian R. Nichols
```

What is a Function Call?

The MapBasic language supports many different function calls. Each function has a different purpose. For example, the **Sqr()** function calculates square root values, while the **UCase\$()** function converts a text string to uppercase. When you enter a function name into your program, your program calls the named function, and the function returns a value.

A function call can comprise all or part of an expression. For example, the following statement assigns a value to the variable, `x`, based on the value returned by the **Minimum()** function:

```
x = Minimum( y, z )
```

The MapBasic function call syntax is similar to that of other modern BASIC languages. The function name (for example, "Minimum", in the example above) is followed by a pair of parentheses. If the function takes any parameters, the parameters appear inside the parentheses. If the function takes more than one parameter, the parameters are separated by commas (the **Minimum()** function takes two parameters).

A function call is different than a generic statement, in that the function call returns a value. A function call cannot act as a stand-alone statement; instead, the value returned by the function must be incorporated into some larger statement. Thus, the following program consists of two statements: a

Dim statement declares a variable, *x*; and then an assignment statement assigns a value to the variable. The assignment statement incorporates a function call (calling the **Sqr()** function to calculate the square root of a number):

```
Dim x As Float
x = Sqr(2)
```

Similarly, the following program uses the **CurDate()** function, which returns a Date value representing the current date:

```
Dim today, yesterday As Date
today = CurDate( )
yesterday = today - 1
```

The **CurDate()** function takes no parameters. When you call a function in MapBasic, you must follow the function name with a pair of parentheses, as in the example above, even if the function takes no parameters.

MapBasic supports many standard BASIC functions, such as **Chr\$()** and **Sqr()**, as well as a variety of special geographic functions such as **Area()** and **Perimeter()**.

A Closer Look At Constants

A constant is a specific value that does not change during program execution. Programmers sometimes refer to constants as “hard-coded” expressions, or as “literals.”

Numeric Constants: Different types of numeric variables require different types of constants. For instance, the constant value 36 is a generic numeric constant. You can assign the value 36 to any numeric variable, regardless of whether the variable is Integer, SmallInt, or Float. The value 86.4 is a floating-point numeric constant.

Hexadecimal Numeric Constants: MapBasic 4.0 and later supports hexadecimal numeric constants using the Visual Basic syntax: **&Hnumber** (where *number* is a hexadecimal number). The following example assigns the hexadecimal value 1A (which equals decimal 26) to a variable:

```
Dim i_num As Integer
i_num = &H1A
```

Numeric constants may not include commas (thousand separators). Thus, the following statement will not compile correctly:

```
counter = 1,250,000 ' This won't work!
```

If a numeric constant includes a decimal point (decimal separator), the separator character must be a period, even if the user's computer is set up to use some other character as the decimal separator.

String Constants: A String constant is enclosed in double quotation marks. For example:

```
last_name = "Nichols"
```

Each string constant can be up to 256 characters long.

The double quotation marks are not actually part of the string constant, they merely indicate the starting and ending points of the string constant. If you need to incorporate a double-quotation mark character within a string constant, insert two consecutive double-quotation marks into the string. The following program illustrates how to embed quotation marks within a string:

Note "The table ""World"" is already open."

Logical Constants: Logical constants can be either one (1) for TRUE or zero (0) for FALSE. Many MapBasic programs refer to the values TRUE and FALSE; note that TRUE and FALSE are actually defined within the standard MapBasic definitions file, mapbasic.def. To refer to standard definitions like TRUE and FALSE, a program must issue an Include statement, to include mapbasic.def. For example:

```
Include "mapbasic.def"
Dim edits_pending As Logical
edits_pending = FALSE
```

Date Constants: To specify a date constant, enter an eight-digit Integer with the format YYYYMMDD. This example specifies the date December 31, 1995:

```
Dim d_enddate As Date
d_enddate = 19951231
```

Alternately, you can specify a string expression that acts as a date constant:

```
d_enddate = "12/31/1995"
```

When you specify a string as a date constant, the year component can be four digits or two digits:

```
d_enddate = "12/31/95"
```

You can omit the year, in which case the current year is used:

```
d_enddate = "12/31"
```

Caution: Using a string as a date constant is sometimes unreliable, because the results you get depend on how the user's computer is configured. If the user's computer is configured to use Month/Day/Year formatting, then "06/11/95" represents June 11, but if the computer is set up to use Day/Month/Year formatting, then "06/11/95" represents the 6th of November.

If the user's computer is set up to use "-" as the separator, MapInfo Professional cannot convert string expressions such as "12/31" into dates.

To guarantee predictable results, use the **NumberToDate()** function, which accepts the eight-digit numeric date syntax. (Numeric date constants, such as 19951231, are not affected by how the user's computer is configured.) If you need to use strings as date values - perhaps because you are reading date values from a text file - use the **Set Format** statement to control how the strings are interpreted. For **Set Format** statement details, see the MapBasic *Reference* or online Help.

To configure date formatting options under Microsoft Windows, use the Regional Settings control panel. **Alias Constants:** Alias variables are discussed in detail in **Chapter 8: Working With Tables**. You can assign a string expression to a variable of type Alias. For example:

```
Dim column_name As Alias
column_name = "City"
```

The following table contains examples of various types of constants.

Types	Sample assignments	Notes
Integer	i = 1234567	
SmallInt	m = 90	
Float	f = 4 size = 3.31 debt = 3.4e9	
String	s_mesg = "Brian Nichols"	Enclose string in double quotes. To embed quotes in a string, type two quotation marks. To include special characters use the Chr\$() function.
Logical	edits_pending = 1 edits_pending = TRUE	1= true, 0 = false The MapBasic definition file defines TRUE and FALSE.
Date	d_starting = 19940105 date_done = "3/23/88" paidddate = "12-24-1993" yesterday = CurDate() - 1	
Alias	col_name = "Pop_1990" col_name = "COL1"	Aliases can be assigned like strings. See Chapter 8: Working With Tables for more information about Alias variables.
Pen	hwypen = MakePen(1, 3, BLACK)	There is no constant syntax for Pen expressions.
Brush	zbrush = MakeBrush(5, BLUE, WHITE)	There is no Brush constant syntax.
Font	lbl_font = MakeFont("Helv", 1, 20, BLACK, WHITE)	There is no Font constant syntax.
Symbol	loc_sym = MakeSymbol(44, RED, 16)	There is no Symbol constant syntax.
Object	path = CreateLine(73.2, 40, 73.6, 40.4)	There is no Object constant syntax.

Variable Type Conversion

MapBasic provides functions for converting data of one type to another type. For instance, given a number, you can produce a string representing the number calling the function **Str\$()**:

```
Dim q1, q2, q3, q4, total As Float, s_message As String
...
total = q1 + q2 + q3 + q4
s_message = "Grand total: " + Str$(total)
```

A Closer Look At Operators

Operators act on one or more values to produce a result. Operators can be classified by the data types they use and the types of results they produce.

Numeric Operators: Each of the operators in the following table is a numeric operator. Two numeric values can be combined using a numeric operator to produce a numeric result.

Operator	Performs	Example
+	addition	$x = a + b$
-	subtraction	$x = a - b$
*	multiplication	$x = a * b$
/	division	$x = a / b$
\	integer division	$x = a \backslash b$
Mod	integer remainder	$x = a \text{ Mod } b$
^	exponentiation	$x = a ^ b$

The \ and Mod operators perform integer division. For example:

10 / 8	returns	1.25
10 \ 8	returns	1 (the integer portion of 1.25)
10 Mod 8	returns	2 (the remainder after dividing 10 by 8)

The minus sign (-) operator can be used to negate a numeric value

```
x = -23
```

String Operators: The plus operator (+) lets you concatenate two or more string expressions into one long string expression.

```
Note "Employee name: " + first_name + " " + last_name
```

You can use the ampersand operator (&) instead of the plus operator when concatenating strings. The & operator forces both operands to be strings, and then concatenates the strings. This is different than the + operator, which can work with numbers or dates without forcing conversion to strings.

Note: The & character is also used to specify hexadecimal numbers (&H*number*). When you use & for string concatenation, make sure you put a space before and after the & so that the MapBasic compiler does not mistake the & for a hex number prefix.

The **Like** operator performs string comparisons involving wild-card matching. The following example tests whether the contents of a String variable begins with the string "North":

```
If s_state_name Like "North%" Then ...
```

The **Like** operator is similar to the **Like ()** function. For a description of the **Like ()** function, see the *MapBasic Reference* or online Help.

Date Operators: The plus and minus operators may both be used in date expressions, as summarized below.

Expression	Returns
date + integer	a Date value, representing a later date
date - integer	a Date value, representing an earlier date
date - date	an Integer value, representing the number of elapsed days

The following example uses the **CurDate()** function to determine the current date, and then calculates other date expressions representing tomorrow's date and the date one week ago:

```
Dim today, one_week_ago, tomorrow As Date,
    days_elapsed As Integer
today = CurDate( )
tomorrow = today + 1
one_week_ago = today - 7
' calculate days elapsed since January 1:
days_elapsed = today - StringToDate("1/1")
```

Comparison Operators: A comparison operator compares two items of the same general type to produce a logical value of TRUE or FALSE. Comparison operators are often used in conditional expressions (for example, in an If...Then statement).

Operator	Returns TRUE if	Example
=	equal to	If a = b Then ...
<>	not equal to	If a <> b Then ...
<	less than	If a < b Then ...
>	greater than	If a > b Then ...
<=	less than or equal to	If a <= b Then ...
>=	greater than or equal to	If a >= b Then ...
Between...And...value is within range	If x Between f_low And f_high Then...	

Each of these comparison operators may be used to compare string expressions, numeric expressions, or date expressions. Note, however, that comparison operators may not be used to compare Object, Pen, Brush, Symbol, or Font expressions.

The Between...And... comparison operator lets you test whether a data value is within a range. The following **If...Then** statement uses a Between...And... comparison:

```
If x Between 0 And 100 Then
    Note "Data within range."
Else
    Note "Data out of range."
End If
```

The same program could be written another way:

```
If x >= 0 And x <= 100 Then
    Note "Data within range."
Else
    Note "Data out of range."
End If
```

When you use the = operator to compare two strings, MapBasic examines the entire length of both strings, and returns TRUE if the strings are identical. String comparisons are not case sensitive; so this **If...Then** statement considers the two names ("Albany" and "ALBANY") to be identical:

```
Dim city_name As String
city_name = "ALBANY"
If city_name = "Albany" Then
    Note "City names match."
End If
```

If you wish to perform case-sensitive string comparison, use the **StringCompare()** function, which is described in the MapBasic *Reference*.

Note: Be careful when comparing fixed-length and variable-length strings. MapBasic automatically pads every fixed-length string with spaces, if necessary, to ensure that the string fills the allotted space. Variable-length strings, however, are not padded in this manner. Depending on your data and variables, this difference might mean that two seemingly-identical strings are not actually equal.

You can use the **RTrim\$()** function to obtain a non-padded version of a fixed-length string. You then can compare the value returned by **RTrim\$()** with a variable-length string, without worrying about interference from padded spaces.

Logical Operators: 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	If a And b Then...
Or	either operand is TRUE	If a Or b Then...
Not	operand is FALSE.	If Not a Then...

For example, the following **If...Then** statement performs two tests, testing whether the variable *x* is less than zero, and testing whether *x* is greater than ten. The program then displays an error message if either test failed.

```
If x < 0 Or x > 10 Then
  Note "Number is out of range."
End If
```

Geographic Operators: These operators act on Object expressions to produce a logical result of TRUE or FALSE.

Operator	Returns TRUE if	Example
Contains	first object contains centroid of second object	If a Contains b Then...
Contains Part	first object contains part of second object	If a Contains Part b Then...
Contains Entire	first object contains all of second object	If a Contains Entire b Then...
Within	first object's centroid is within second object	If a Within b Then...
Partly Within	part of first object is within second object	If a Partly Within b Then...
Entirely Within	all of first object is within second object	If a Entirely Within b Then...
Intersects	the two objects intersect at some point	If a Intersects b Then...

For a more complete discussion of graphic objects, see [Chapter 10: Graphical Objects](#).

MapBasic Operator Precedence

Some operators have higher *precedence* than others. This means that in a complex expression containing multiple operators, MapBasic follows certain rules when determining which operations to carry out first. To understand how MapBasic processes complex expressions, you must be familiar with the relative precedence of MapBasic's operators.

Consider the following mathematical assignment:

```
x = 2 + 3 * 4
```

This assignment involves two mathematical operations addition and multiplication. Note that the end result depends on which operation is performed first. If you perform the addition first (adding 2 + 3, to obtain 5), followed by the multiplication (multiplying 5 * 4), the end result is 20. In practice, however, multiplication has a higher precedence than addition. This means that MapBasic performs the multiplication first (multiplying 3 * 4, to obtain 12), followed by the addition (adding 2 + 12, to obtain 14).

You can use parentheses to override MapBasic's default order of precedence. The following assignment uses parentheses to ensure that addition is performed before multiplication:

```
x = (2 + 3) * 4
```

The following table identifies the precedence of each MapBasic operator.

Highest priority:	parentheses
	exponentiation
	negation
	multiplication, division, Mod, integer division
	addition, subtraction, string concatenation (&)
	geographic operators, comparison operators, Like
	Not
	And
Lowest Priority:	Or

Operators appearing on the same row have equal precedence. Operators of higher priority are processed first. Operators of the same precedence are evaluated left to right in the expression, except exponentiation, which evaluates from right to left.

Looping, Branching, and Other Flow-Control

Flow-control statements affect the order in which other statements are executed. MapBasic has three main types of flow-control statements:

- Branching statements cause MapBasic to skip over certain statements in your program (for example, **If...Then**, **GoTo**).
- Looping statements cause MapBasic to repeatedly execute one or more designated statements in your program (for example, **For...Next**, **Do...While**).
- Other statements provide special flow-control (for example, **End Program**).

If...Then Statement

MapBasic's **If...Then** statement is very similar to comparable If...Then statements in other languages. The **If...Then** statement tests a condition; if the condition is TRUE, MapBasic executes the statements which follow the **Then** keyword. In the following example, MapBasic displays an error message and calls a sub-procedure if a counter variable is too low:

```
If counter < 0 Then
  Note "Error: The counter is too low."
  Call reset_counter
End If
```

An **If...Then** statement can have an optional **Else** clause. In the event that the original test condition was FALSE, MapBasic executes the statements following the **Else** keyword *instead of* executing the statements following the **Then** keyword. The following example demonstrates the optional **Else** clause.

```
If counter < 0 Then
    Note "Error: The counter is too low."
    Call reset_counter
Else
    Note "The counter is OK."
End If
```

An **If...Then** statement can also have one or more optional **Elseif** clauses. The **Elseif** clause tests an additional condition. If the statement includes an **Elseif** clause, and if the original condition turned out to be FALSE, MapBasic will test the **Elseif** clause, as in the following example:

```
If counter < 0 Then
    Note "Error: The counter is too low."
    Call reset_counter
Elseif counter > 100 Then
    counter = 100
    Note "Error: The counter is too high; resetting to 100."
Else
    Note "The counter is OK."
End If
```

Note: **Elseif** is a single keyword. A single **If...Then** statement can include a succession of two or more **Elseif** clauses, subsequently testing for condition after condition. However, if you want to test for more than two or three different conditions, you may want to use the **Do...Case** statement (described below) instead of constructing an **If...Then** statement with a large number of **Elseif** clauses.

Do Case Statement

The **Do Case** statement performs a series of conditional tests, testing whether a certain expression is equal to one of the values in a list of potential values. Depending on which value the expression matches (if any), MapBasic carries out a different set of instructions.

The following example tests whether the current month is part of the first, second, third, or fourth quarter of the fiscal year. If the current month is part of the first quarter (January-February-March), the program assigns a text string an appropriate title ("First Quarter Results"). Alternately, if the current month is part of the second quarter, the program assigns a different title ("Second Quarter Results"), etc.

```
Dim current_month, quarter As SmallInt,
    report_title As String
current_month = Month( CurDate() )
' At this point, current_month is 1 if current date
' is in January, 2 if current date is in February, etc.
Do Case current_month
    Case 1, 2, 3
        ' If current month is 1 (Jan), 2 (Feb) or 3 (Mar),
        ' we're in the First fiscal quarter.
        ' Assign an appropriate title.
        report_title = "First Quarter Results"
        quarter = 1
    Case 4, 5, 6
        report_title = "Second Quarter Results"
        quarter = 2
    Case 7, 8, 9
        report_title = "Third Quarter Results"
        quarter = 3
    Case Else
        '
        ' If current month wasn't between 1 and 9, then
        ' current date must be in the Fourth Quarter.
        '
        report_title = "Fourth Quarter Results"
        quarter = 4
End Case
```

Note: The **Case Else** clause in the final part of the **Do Case** construction. **Case Else** is an optional clause. If a **Do Case** statement includes a **Case Else** clause, and if none of the previous **Case** clauses matched the expression being tested, MapBasic carries out the statements following the **Case Else** clause. The **Case Else** clause must be the final clause in the **Do Case** construction.

GoTo Statement

The **GoTo** statement tells MapBasic to go to a different part of the program and resume program execution from that point. The **GoTo** statement specifies a label. For the **GoTo** statement to work, there must be a label elsewhere within the same procedure. A label is a name which begins a line. Each label must end with a colon (although the colon is not included in the **GoTo** statement).

```
If counter < 0 Then
    GoTo get_out
End If
...
get_out:
End Program
```

Many programming professionals discourage the use of **GoTo** statements. Careful use of other flow-control statements, such as **If...Then**, usually eliminates the need to use **GoTo** statements. Thus, you may want to avoid using **GoTo** statements.

For...Next Statement

The **For...Next** statement sets up a loop that executes a specific number of times. With each iteration of the loop, MapBasic executes all statements that appear between the **For** and **Next** clauses. When creating a **For...Next** loop, you must specify the name of a numeric variable as a counter. You must also specify that counter variable's starting and ending values. With each iteration of the loop, MapBasic increments the counter variable by some step value. By default, this step value is one. To use a different increment, include the optional **Step** clause.

The following example uses a **For...Next** loop to add the values from an array of numbers:

```
Dim monthly_sales(12), grand_total As Float,
    next_one As SmallInt
...
For next_one = 1 To 12
    grand_total = grand_total + monthly_sales(next_one)
Next
```

At the start of the **For...Next** statement, MapBasic assigns the start value to the counter variable. In the example above, MapBasic assigns a value of one to the variable: `next_one`. MapBasic then executes the statements that appear up to the **Next** keyword. After each iteration of the loop, MapBasic increments the counter variable. If the counter variable is less than or equal to the end value (for example, if `next_one` is less than or equal to twelve), MapBasic performs another iteration of the loop.

A **For...Next** loop halts immediately if it encounters an **Exit For** statement. This allows you to conditionally halt the loop prematurely.

Note: If you construct a **For...Next** loop which uses precise floating-point values (for example, **For i = 0.1 to 1.0 Step 0.1**), the loop may behave differently on MapInfo for Macintosh than it behaves on MapInfo for Windows (i.e., there may be one more loop iteration with Macintosh than with Windows). This is a result of the way that floating-point math is handled internally on the Macintosh.

See the MapBasic *Reference* for more information on the **For...Next** loop.

Do...Loop

The **Do...Loop** statement continually executes a group of statements for as long as a test condition remains TRUE or, optionally, for as long as the condition remains FALSE.

There are different forms of the **Do...Loop** statement, depending on whether you want to test the looping condition before or after the body of the statements that are executed. The following program tests the loop condition at the end of the loop:

```
Dim sales_total, new_accounts(10) As Float,
    next_one As SmallInt
next_one = 1
Do
    sales_total = sales_total + new_accounts(next_one)
    next_one = next_one + 1
Loop While next_one <= UBound(new_accounts)
```

Note that the preceding loop always executes for at least one iteration, because the looping condition is not tested until the end of the loop.

The following loop tests the loop condition at the start of the loop. Because the condition is tested at the start of the loop, the statements within the body of the loop may never be executed. If the test condition is FALSE from the beginning, the statements within the following **Do...Loop** will never be executed.

```
Dim sales_total, new_accounts(10) As Float,  
    next_one As SmallInt  
next_one = 1  
Do While next_one <= UBound(new_accounts)  
    sales_total = sales_total + new_accounts(next_one)  
    next_one = next_one + 1  
Loop
```

In the examples above, both **Do...Loop** statements included the keyword **While**; thus, both loops continue while the test condition remains TRUE. Alternately, a **Do...Loop** can use the **Until** keyword *instead of* the keyword **While**. If a **Do...Loop** statement specifies **Until**, the loop will continue only for as long as the test condition remains FALSE.

A **Do...Loop** statement halts immediately if it encounters an **Exit Do** statement. This statement allows you to conditionally terminate a loop prematurely.

While...Wend Loop

MapBasic supports the conventional BASIC **While...Wend** loop syntax. A **While...Wend** statement is very similar to a **Do While...Loop** statement.

If you are an experienced BASIC programmer, and you therefore are in the habit of using **While...Wend** statements, you can continue to use **While...Wend** statements as you use MapBasic. Note, however, that the **Do...Loop** statement syntax is in some ways more powerful than the **While...Wend** syntax. You can exit a **Do...Loop** statement prematurely, through the **Exit Do** statement, but there is no corresponding statement for exiting a **While...Wend** loop.

See the MapBasic *Reference* for more information on the **While...Wend** loop.

Ending Your Program

The **End Program** statement halts the MapBasic application, removes any custom menu items created by the application, and removes the application from memory. **End Program** also closes any files opened by the application (through the **Open File** statement), but it does not close any open tables.

The **End Program** statement is not required. In fact, there are situations where you should be careful *not* to issue an **End Program** statement. For example, if your application adds menu items to a MapInfo Professional menu, you probably want your application to remain running for the duration of the MapInfo Professional session, because you want your custom menu items to remain available for the entire session. In such cases, you should be careful not to issue the **End Program** statement, because **End Program** would halt your application and remove your application's custom menu items. For a complete discussion of custom menus, see **Chapter 7: Creating the User Interface**.

Ending Your Program and MapInfo Professional

The **End MapInfo** statement halts the MapBasic application (much as the **End Program** statement does), and then exits the MapInfo Professional software as well.

Procedures

Procedures (often referred to as sub-procedures) are an integral part of the MapBasic program architecture. A typical MapBasic program is comprised of numerous sub-procedures; each sub-procedure contains a group of statements that perform a specific task. By breaking your program into several sub-procedures, you modularize your program, making program development and maintenance easier in the long run.

Main Procedure

Every MapBasic program has at least one procedure, known as the **Main** procedure. When you run a MapBasic application, MapBasic automatically calls that application's Main procedure.

The following program demonstrates the syntax for explicitly declaring the Main procedure. In this example, the Main procedure simply issues a **Note** statement:

```
Declare Sub Main
Sub Main
    Note "Hello from MapBasic!"
End Sub
```

The **Declare Sub** statement tells MapBasic that a sub-procedure definition will occur further down. You must have one **Declare Sub** statement for each sub-procedure in your program. The **Declare Sub** statement must appear before the actual sub-procedure definition. Typically, **Declare Sub** statements appear at or near the top of your program.

You may recall from [Chapter 4: Using the Development Environment](#) that a MapBasic program can be as simple as a single line. For example, the following statement:

```
Note "Hello from MapBasic!"
```

is a complete MapBasic program which you can compile and run. Note that even a simple, one-line program has a Main procedure. However, in this case, we say that the Main procedure is **implied** rather than being **explicit**.

Calling a Procedure

When you run a compiled application, MapInfo Professional automatically calls the Main procedure (regardless of whether the Main procedure is implied or explicitly defined). The Main procedure can then call other sub-procedures through the **Call** statement.

The following program contains two procedures: a Main procedure, and a procedure called `announce_date`.

```
Declare Sub Main
Declare Sub announce_date

Sub Main
    Call announce_date( )
End Sub

Sub announce_date
    Note "Today's date is " + Str$( CurDate() )
End Sub
```

Calling a Procedure That Has Parameters

Like other modern BASIC languages, MapBasic lets you create sub-procedures which take parameters. If a sub-procedure takes parameters, they are declared within parentheses which follow the procedure name in the **Sub...End Sub** statement.

The following example shows a sub-procedure called `check_date`, which takes one parameter (a Date value). The sub-procedure checks to see whether the value of the Date parameter is too old (more than 180 days old). If the Date parameter value is too old, the procedure sets the Date parameter to the current date.

```
Declare Sub Main
Declare Sub check_date(last_date As Date)
Sub Main
    Dim report_date As Date
    report_date = "01/01/94"
    Call check_date( report_date )
    ' At this point, the variable: report_date
    ' may contain the current date (depending on
    ' what happened in the check_date procedure).
End Sub

Sub check_date(last_date As Date)
    Dim elapsed_days As SmallInt
    elapsed_days = CurDate() - last_date
    If elapsed_days > 180 Then
        last_date = CurDate()
    End If
End Sub
```

Passing Parameters By Reference

By default, each MapBasic procedure parameter is passed *by reference*. When a parameter is passed by reference, the following rules apply:

- The **Call** statement *must* specify the name of a variable for each by-reference parameter.
- If the called sub-procedure assigns a new value to a by-reference parameter, the new value is automatically stored in the caller's variable. In other words, the sub-procedure can use a by-reference parameter to return a value to the caller.

Thus, in the example above, the **Call** statement specifies the name of a Date variable `report_date`:

```
Call check_date( report_date )
```

Then, within the `check_date` procedure, the parameter is known by the name `last_date`. When the `check_date` procedure performs the assignment **`last_date = CurDate()`**, MapBasic automatically updates the Main procedure's `report_date` variable.

Passing Parameters By Value

Sometimes, it is awkward to pass parameters by reference. For each by-reference parameter, you must specify the name of a variable in your **Call** statement. At times, you may find this awkward (for example, because you may not have a variable of the appropriate type).

Like other modern BASIC languages, MapBasic lets you specify that a procedure parameter will be passed *by value* rather than by reference. To specify that a parameter be passed by value, include the keyword **ByVal** before the parameter's name in the **Sub...End Sub** statement.

When a parameter is passed by value, the following rules apply:

- The **Call** statement does not need to specify the name of a variable as the parameter. The **Call** statement may specify a variable name, a constant value or some other expression.
- If the called sub-procedure assigns a new value to a by-value parameter, the calling procedure is not affected. In other words, the sub-procedure **cannot** use a by-value parameter to return a value to the caller.

The following example shows a procedure (`display_date_range`) which takes two by-value Date parameters.

```
Declare Sub Main
Declare Sub display_date_range(ByVal start_date As Date,
                              ByVal end_date As Date )

Sub Main
  Call display_date_range( "1/1", CurDate() )
End Sub

Sub display_date_range(ByVal start_date As Date,
                      ByVal end_date As Date )
  Note "The report date range will be: " + Str$(start_date)
    + " through " + Str$(end_date) + "."
End Sub
```

In this example, both of the parameters to the `display_date_range` procedure are by-value date parameters. Thus, when the Main procedure calls `display_date_range`:

```
Call display_date_range( "1/1", CurDate() )
```

neither of the parameters needs to be a Date variable. The first parameter ("1/1") is a constant Date expression, and the second parameter is a date expression derived by calling the **CurDate()** function.

Calling Procedures Recursively

The MapBasic language supports recursive function and procedure calls. In other words, a MapBasic procedure can call itself.

Programs that issue recursive procedure or function calls may encounter memory limitations. Each time a program makes a recursive call, MapInfo Professional must store data on the stack; if too many nested recursive calls are made, the program may generate an out-of-memory error. The amount of memory used up by a recursive call depends on the number of parameters and local variables associated with the procedure or function.

Procedures That Act As System Event Handlers

Some procedure names have special meaning in MapBasic. For example, as we have seen, the sub-procedure named **Main** is special, since MapBasic automatically calls the **Main** procedure when you run an application.

In addition to **Main**, MapBasic has several other special procedure names: **EndHandler**, **ForegroundTaskSwitchHandler**, **RemoteMapGenHandler**, **RemoteMsgHandler**, **RemoteQueryHandler()**, **SelChangedHandler**, **ToolHandler**, **WinChangedHandler**, **WinClosedHandler**, and **WinFocusChangedHandler**. Each of these reserved procedure names plays a special role in MapBasic programming. To fully understand how they work, you need to understand MapBasic's approach to system events and event-handling.

What Is a System Event?

In a Graphical User Interface environment, the user controls what happens by typing and by using the mouse. Technically, we say that mouse-clicks and other actions taken by the user generate *system events*. There are many different kinds of events; for example, when the user chooses a menu item, we say that the user has generated a menu-choose event, and when the user closes a window, we say the user has generated a window-close event.

What Is an Event Handler?

An event-handler is part of a MapBasic program which responds to a system event. Once the user has generated an event, the application must respond accordingly. For instance, when the user generates a menu-choose event, the software may need to display a dialog. Alternately, when the user generates a window-close event, the software may need to gray out a menu item or hide an entire menu.

In MapBasic, sub-procedures can act as event-handlers. In other words, you can construct your program in such a way that MapBasic automatically calls one of your sub-procedures when and if a certain system event occurs.

To build event-handlers that respond to menu or button-pad choices, see [Chapter 7: Creating the User Interface](#). To build any other types of system event-handlers, you must define a sub-procedure with a special name. For example, if you want your program to respond automatically whenever the user closes a window, your application must contain a procedure named **WinClosedHandler**.

The following table lists all of MapBasic's special handler names. These special handlers are discussed in more detail in the MapBasic *Reference* and online Help.

Special Handler Name	Nature of Handler Procedure or Function (see <i>Reference</i> for details)
EndHandler	Called when the application terminates or when the user exits MapInfo Professional. EndHandler can be used to do clean-up work (for example, deleting temporary work files).
ForegroundTaskSwitchHandler	Called when MapInfo Professional gets the focus (becomes the active application) or loses the focus.
RemoteMapGenHandler	Called when an OLE Automation client calls the MapGenHandler method; used primarily in MapInfo ProServer applications.
RemoteMsgHandler	Called when the application is acting as the server in an interprocess conversation, and the remote client sends an execute request.
RemoteQueryHandler()	Called when the application is acting as the server in an interprocess conversation, and the remote client sends a peek request.
SelChangedHandler	Called whenever the Selection table changes. Since the Selection table changes frequently, the SelChangedHandler procedure should be as brief as possible to avoid slowing system performance.
ToolHandler	Called when the user clicks in a Mapper, Browser, or Layout window using the MapBasic tool.
WinChangedHandler	Called when the user pans, scrolls, or otherwise resets the area displayed in a Mapper. Since Mapper windows can change frequently, the WinChangedHandler procedure should be as brief as possible to avoid slowing system performance.
WinClosedHandler	Called when the user closes a Mapper, Browser, Grapher, or Layout.
WinFocusChangedHandler	Called when the window focus changes (i.e., when the user changes which window is the active window).

Typically, you do not use the **Call** statement to call the special procedures listed above. If your program contains one of these specially named procedures, MapBasic calls that procedure *automatically*, when and if a certain type of system event occurs. For example, if your program contains a procedure called WinClosedHandler, MapBasic automatically calls the WinClosedHandler procedure every time the user closes a window.

All of the special handler procedures are optional. Thus, you should only include a **WinClosedHandler** procedure in your application if you want your application to be notified every time a window is closed. You should only include a **SelChangedHandler** procedure in your application if you want your application to be notified each time Selection changes, etc.

The following program defines a special event-handler procedure named **ToolHandler**. Note that this program does not contain any **Call** statements. Once this program is running, MapBasic calls the ToolHandler procedure automatically, when and if the user selects the MapBasic tool and clicks on a Mapper, Browser, or Layout window.

```

Include "mapbasic.def"
Declare Sub Main
Declare Sub ToolHandler
Sub Main
    Note "The ToolHandler demonstration is now in place. "
    + "Select the MapBasic tool (+) and click on a Map "
    + "to see a printout of map coordinates."
End Sub
Sub ToolHandler
    If WindowInfo( FrontWindow(),
                  WIN_INFO_TYPE ) = WIN_MAPPER Then
        Print "X: " + Str$( CommandInfo(CMD_INFO_X) )
        Print "Y: " + Str$( CommandInfo(CMD_INFO_Y) )
        Print " "
    End If
End Sub

```

Within a system event handler procedure, you can call the **CommandInfo()** function to learn more about the event that made MapBasic call the handler. In the example above, the **ToolHandler** procedure calls **CommandInfo()** to determine the map coordinates where the user clicked.

The following sample SelChangedHandler procedure appears in the sample program, TextBox (textbox.mb). This procedure automatically disables (grays out) a menu item whenever the user de-selects all rows, and automatically re-enables the menu item whenever the user selects more rows.

See textbox.mb for more details.

```

Sub SelChangedHandler
    If SelectionInfo(SEL_INFO_NROWS) < 1 Then
        Alter Menu Item create_sub Disable
    Else
        Alter Menu Item create_sub Enable
    End If
End Sub

```

When Is a System Event Handler Called?

By default, a MapBasic application terminates after executing all statements in the **Main** procedure. However, if an application contains one or more of the special handler procedures listed above (for example, if an application contains a **ToolHandler** procedure), the application remains in memory after the **Main** procedure is finished. An application in this state is said to be *sleeping*. A sleeping application remains dormant in memory until an appropriate event occurs (for example, until the user clicks with the MapBasic tool). When the event occurs, MapBasic automatically calls the sleeping application's handler procedure.

Note: If any procedure in an application issues the **End Program** statement, the entire application is removed from memory, regardless of whether the application contains special handler procedures. You must avoid using the **End Program** statement for as long as you want your program to remain available.

Custom MapBasic menus work in a similar manner. If a MapBasic application adds its own items to the MapInfo Professional menu structure, the application goes to sleep and waits for the user to choose one of the custom menu items. For a complete discussion of how to customize MapInfo Professional's menus, see **Chapter 7: Creating the User Interface**.

Tips for Handler Procedures

Keep Handler Procedures Short

Bear in mind that some system event-handler procedures are called frequently. For example, if you create a `SelChangedHandler` procedure, MapInfo Professional calls the procedure *every time* the Selection table changes. In a typical MapInfo Professional session, the Selection table changes frequently, therefore, you should make event-handler procedures, such as `SelChangedHandler`, as short as possible.

Selecting Without Calling SelChangedHandler

If you are using a **Select** statement, but you do not want the statement to trigger the `SelChangedHandler` procedure, include the **NoSelect** keyword. For example:

```
Select *      From World      Into EarthQuery NoSelect
```

Preventing Infinite Loops

Performing actions within a system handler procedure can sometimes cause an infinite loop. For example, if you declare a `SelChangedHandler` procedure, MapInfo Professional calls that procedure whenever the selection changes. If you issue a **Select** statement inside of your `SelChangedHandler` procedure, the **Select** statement will cause MapInfo Professional to call the procedure *again* in a recursive call. The end result can be an infinite loop, which continues until your program runs out of memory.

The **Set Handler** statement can help prevent infinite loops. At the start of your handler procedure, issue a **Set Handler ... Off** statement to prevent recursive calling of the handler. At the end of the procedure, issue a **Set Handler ... On** statement to restore the handler.

```
Sub SelChangedHandler
  Set Handler SelChangedHandler Off

  ' Issuing a Select statement here
  ' will not cause an infinite loop.

  Set Handler SelChangedHandler On
End Sub
```

Custom Functions

The MapBasic language supports many different functions. Some are standard BASIC functions (for example, **Asc()**, **Format\$()**, **Val()**, etc.) and some are unique to MapInfo Professional and MapBasic (for example, **Distance()** and **ObjectGeography()**). MapBasic also lets you define custom functions. Once you have defined a custom function, you can call that function just as you can call any of MapBasic's standard functions.

The body of a custom function is defined within a **Function...End Function** construction, which is syntactically very similar to a **Sub...End Sub** construction. The general syntax of a **Function...End Function** construct is as follows:

```
Function  function_name( parameters, if any) As data_type
    statement list
End Function
```

The function itself has a data type. This dictates which type of value (for example, Integer, Date, String) the function returns when called.

Within the body of the **Function...End Function** construction, the function name acts like a by-reference parameter. A statement within the **Function...End Function** construction can assign a value to the function name; this is the value that MapBasic later returns to the function's caller.

The example below defines a custom function called `money_format()`. The `money_format()` function takes one numeric parameter (presumably representing a sum of money), and returns a string (obtained by calling the **Format\$()** function) representing the dollar amount, formatted with commas.

```
Declare Sub Main
Declare Function money_format(ByVal num As Float) As String
Sub Main
    Dim dollar_amount As String
    dollar_amount = money_format( 1234567.89 )
    ' dollar_amount now contains the string: "$1,234,567.89"
End Sub

Function money_format(ByVal num As Float) As String
    money_format = Format$(num, "$,###.##;($,###.##)")
End Function
```

Scope of Functions

A program can define a custom function that has the same name as a standard MapBasic function. When the program calls the function, the custom function is executed instead of the standard function.

Compiler Instructions

MapBasic provides two special statements which make it easier to manage large-scale application development:

- The **Define** statement lets you define a shorthand identifier which has a definition; the definition is substituted for the identifier at compile time.
- The **Include** statement lets you combine two or more separate program files into one compilable program.

The Define Statement

Through the **Define** statement, you can define an identifier which acts as a shorthand equivalent for some specific value.

Use a **Define** statement whenever you find yourself frequently typing an expression that is difficult to remember or to type.

For example, if your program deals extensively with objects and object colors, you might find that you frequently need to type in the value **16711680**, a numeric code representing the color red. Typing such a long number quickly becomes tedious. To spare yourself the tedium of typing in 16711680, you could place the following **Define** statement in your program:

```
Define MY_COLOR 16711680
```

This **Define** statement creates an easy-to-remember shorthand keyword (MY_COLOR) representing the number 16711680. After you enter this **Define** statement, you can simply type MY_COLOR in every place where you would have typed 16711680. When you compile your program, MapBasic will assign each occurrence of MY_COLOR a value of 16711680.

There are long-term benefits to using defined keywords. Suppose that you develop a large application which includes many references to the identifier MY_COLOR. Lets presume that you then decide that red is not a good color choice, and you want to use green (65280) instead. You could easily make the switch from red to green simply by changing your **Define** statement to read:

```
Define MY_COLOR 65280
```

The standard MapBasic definitions file, mapbasic.def, contains many **Define** statements, including **Define** statements for several commonly-used colors (BLACK, WHITE, RED, GREEN, BLUE, CYAN, MAGENTA, and YELLOW). Use the **Include** statement to incorporate mapbasic.def into your program.

The Include Statement

Through the **Include** statement, you can incorporate two or more separate program files into one MapBasic application. The **Include** statement has the following syntax:

```
Include "filename"
```

where *filename* is the name of a text file containing MapBasic statements. When you compile a program that contains an **Include** statement, the compiler acts as if the included text is part of the program being compiled.

Many MapBasic applications use the **Include** statement to include the standard MapBasic definitions file, `mapbasic.def`:

```
Include "mapbasic.def"
```

`mapbasic.def` provides **Define** statements for many standard MapBasic identifiers (TRUE, FALSE, RED, GREEN, BLUE, TAB_INFO_NAME, etc.).

The filename that you specify can include a directory path. If the filename that you specify does not include a directory path, the MapBasic compiler looks for the file in the current working directory. If the file is not found in that directory, the compiler looks in the directory where the MapBasic software is installed.

As you develop more and more MapBasic programs, you may find that you use certain sections of code repeatedly. Perhaps you have written a library of one or more custom functions, and you wish to use those custom functions in every MapBasic program that you write. You could put your custom functions into a separate text file, perhaps calling the text file `functs.mb`. You could then incorporate the function library into another program by issuing the statement:

```
Include "functs.mb"
```

Using **Include** statements also lets you work around the memory limitations of the MapBasic text editor. As discussed in **Chapter 4: Using the Development Environment**, each MapBasic edit window is subject to memory limits; once a program file grows too large, you can no longer add statements to the file using a MapBasic edit window. If this happens, you may want to break your program into two or more separate program files, then combine the files using the **Include** statement. Alternately, you could combine the separate modules using a project file; see **Using the Development Environment in Chapter 4 on page 56** for details.

Program Organization

A MapBasic application can include any or all of the different types of statements described in this chapter. However, the different pieces of a MapBasic program must be arranged in a particular manner. For example, **Global** statements may not be placed inside of a **Sub...End Sub** definition.

The following illustration shows a typical arrangement of the various program components.

Global level statements appear at the top of the program . . .

```
Include "mapbasic.def"
other Include statements
Type...End Type statements
Declare Sub statements
Declare Function statements
Define statements
Global statements
```

. . . followed by the Main procedure definition . . .

```
Sub Main
  Dim statements
  ...
End Sub
```

. . . followed by additional sub-procedure definitions . . .

```
Sub ...
  Dim statements
  ...
End Sub
```

. . . and custom Function definitions . . .

```
Function ...
  Dim statements
  ...
End Function
```

Debugging and Trapping Runtime Errors

Even if your program compiles successfully, it may still contain runtime errors (errors that occur when you run your program). For example, if your program creates large database files, the program may generate an error condition if you run it when there is no free disk space.

This chapter shows you how to deal with runtime errors. This is a two-step process: first, you debug your program to find out where the error occurs; then, you modify your program to prevent the error from happening again.

Sections in this Chapter:

- ♦ **Runtime Error Behavior** 104
- ♦ **Debugging a MapBasic Program** 104
- ♦ **Error Trapping** 106

Runtime Error Behavior

There are two main types of programming errors: compilation errors and runtime errors. Compilation errors, discussed in **Chapter 4: Using the Development Environment**, are syntax errors or other typographical mistakes that prevent a program from compiling successfully.

runtime errors are errors that occur when the user actually runs an application. runtime errors occur for various reasons; often, the reason has to do with precise conditions that exist at runtime. For example, the following statement compiles successfully:

```
Map From stats
```

However, if there is no table named “stats,” this program will generate a runtime error. When a runtime error occurs, MapInfo halts the MapBasic application, and then displays a dialog describing the error.



The error message identifies the name of the program file and the line number at which the error occurred. In the example above, the name of the program is **map_it**, and the line number containing the error is 22. This line number identifies which part of your program caused the runtime error. Once you know the line number, you can return to the MapBasic development environment and use the Go To Line command (on the Search menu) to locate the statement that caused the problem.

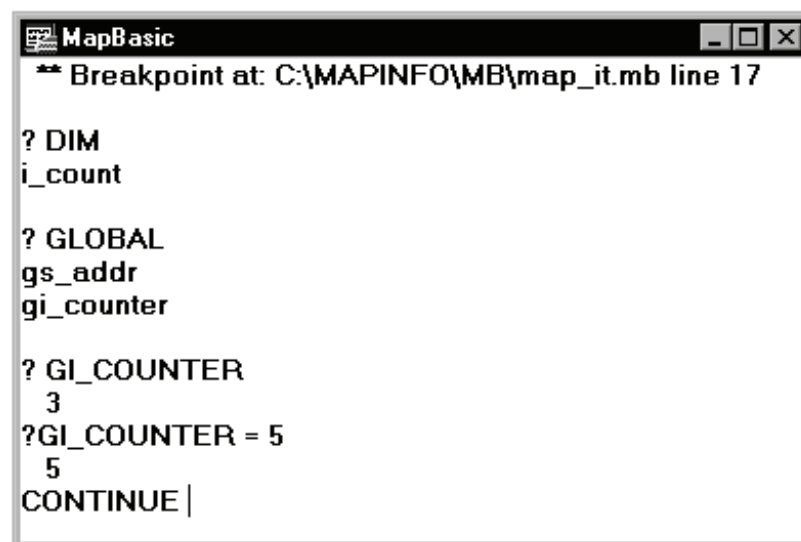
Debugging a MapBasic Program

Some runtime errors are easy to correct. For example, some runtime errors can be caused by modest typing errors (for example, in the example above, the programmer probably meant to enter the table name as STATES instead of STATS). Other errors, however, can be harder to locate. To help you detect and correct bugs in your program, MapBasic provides debugging tools (the **Stop** and **Continue** statements) which work in conjunction with MapInfo's MapBasic Window.

Summary of the Debugging Process

If part of your program is not working correctly, you can use the following procedure to identify where the problem occurs:

1. Within the MapBasic development environment, edit your program, and place a **Stop** statement just before the part of your program that seems to be failing.
2. Recompile and run your program.
When your program reaches the Stop statement, MapBasic temporarily suspends execution of your program and displays a debugging message in the MapBasic window (for example, "Breakpoint at textbox.mb line 23").
3. Within the MapBasic window:
Type ? Dim to see a list of all local variables that are in use.
Type ? Global to see a list of all global variables that are in use.
Type ? *variable_name* to see the current contents of a variable.
Type ? *variable_name* = *new_value* to change the contents of that variable.
4. When you are finished examining and modifying the contents of variables, type Continue in the MapBasic window to resume program execution. Alternately, you can choose the Continue Program command from MapInfo's File menu. Note that while a program is suspended, the File menu contains a Continue Program command instead of a Run Program command.



Limitations of the Stop Statement

In the following cases, MapBasic does not allow you to suspend a program through the **Stop** statement:

- You may not use a **Stop** statement within a custom Function...End Function construct.
- You may not use a **Stop** statement within a **Dialog** control handler, because while the handler is active, the dialog is still on the screen.
- You may not use a **Stop** statement within a **ProgressBar** handler.
- You may not debug one program while another program is running.

- Through the **Run Application** statement, one MapBasic application can “spawn” another application. However, you may not use the **Stop** statement to suspend execution of the spawned application.

Even without using the **Run Application** statement, it is possible to run separate MapBasic programs at one time. For example, if you run the TextBox application, TextBox creates its own custom menu, then remains sleeping until you choose an item from that menu. After loading TextBox, you can run other MapBasic applications. However, you may not use the **Stop** statement while you have multiple applications running simultaneously.

Other Debugging Tools

MapBasic's **Note** and **Print** statements are also helpful when debugging a program. For example, if you wish to observe the contents of a variable as it changes, simply add a **Print** statement to your program:

```
Print "Current value of counter: " + counter
```

to print a message to MapBasic's Message window. The sample program AppInfo.mbx allows you to examine the values of global variables in any MapBasic applications that are running.

Error Trapping

A well-designed program anticipates the possibility of runtime errors and includes precautions whenever possible. Intercepting and dealing with runtime errors is known as error trapping. In MapBasic, error trapping involves using the **OnError** statement.

Veteran BASIC programmers take note: in MapBasic, **OnError** is a single keyword.

At any point during execution, error trapping is either enabled or disabled. By default, all procedures and functions start with error trapping disabled. The **OnError** statement enables error trapping.

Typically, **OnError** specifies a label that must appear at another location in the same procedure or function. The statements following the label are known as the *error-trapping routine*. If an error occurs while an error-trapping routine has been enabled, MapBasic jumps to the specified label and executes the error-trapping routine *instead of* halting the application.

Within the error-trapping routine, you can call the **Err()** function to obtain an Integer code indicating which error occurred. Similarly, **Error\$()** returns a string describing the error message. For a complete listing of potential MapBasic error codes and their descriptions, see the text file **errors.doc** which is included with MapBasic.

Within the error-trapping routine, you can call the **Err()** function to obtain an Integer code indicating which error occurred. Similarly, **Error\$()** returns a string describing the error message. For a complete listing of potential MapBasic error codes and their descriptions, see the text file **Error List**, which is included with MapBasic. Each error-trapping routine ends with a **Resume** statement. The **Resume** statement tells MapBasic which line to go to once the error-trapping routine is finished.

For more about error trapping, see **OnError**, **Resume**, **Err()** and **Error\$()** in the MapBasic *Reference*.

Note: MapBasic can only handle one error at a time. If you enable error-trapping and then an error occurs, MapBasic jumps to your error-handling routine. If another error occurs *within* the error-handling routine (i.e., before the **Resume** statement), your MapBasic application halts.

Example of Error Trapping

The program below opens a table called orders and displays it in Map and Browse windows. An error-trapping routine called **bad_open** handles any errors that relate to the **Open Table** statement. A second error-trapping routine called **not_mappable** handles errors relating to the **Map** statement.

```
Sub orders_setup
  ' At the start, error trapping is disabled
  OnError Goto bad_open
  ' At this point, error trapping is enabled, with
  ' bad_open as the error-handling routine.
  Open Table "orders.tab"
  OnError Goto not_mappable

  ' At this point, error trapping is enabled, with
  ' not_mappable as the new error-handling routine.
  Map From orders
  OnError Goto 0
  Browse * From orders

last_exit:
  Exit Sub
  ' The Exit Sub prevents the program from
  ' unintentionally executing the error handlers.

bad_open:
  ' This routine called if Open statement had an error.
  Note "Couldn't open the table Orders... Halting."
  Resume last_exit

not_mappable:
  ' This routine called if the Map statement had an error
  Note "No map data; data will only appear in Browser."
  Resume Next
End Sub
```

The statement **OnError Goto bad_open** enables error trapping. If an error occurs because of the **Open Table** statement, MapBasic jumps to the error-trapping routine at the label **bad_open**. The error-trapping routine displays an error message, then issues a **Resume** statement to resume execution at the label **last_exit**.

If the **Open Table** statement is successful, the program then issues the statement **OnError Goto not_mappable**. This line resets the error trapping, so that if the **Map** statement generates an error, MapBasic jumps to **not_mappable**. The **not_mappable** error-trapping routine displays a message telling the user why no Mapper window was presented, and then executes a **Resume Next** statement. The **Resume Next** statement tells MapBasic to skip the line that generated the error, and resume with the following statement.

The **OnError Goto 0** statement disables error trapping. Thus, if an error occurs as a result of the **Browse** statement, that error is not trapped, and program execution halts.

Creating the User Interface

The user interface is an important part of every application. MapBasic provides you with all the tools you need to customize MapInfo Professional's user interface.

Sections in this Chapter:

- ♦ Introduction to MapBasic User Interface Principles 109
- ♦ Event-Driven Programming 109
- ♦ Menus 111
- ♦ Standard Dialog Boxes 121
- ♦ Custom Dialog Boxes 123
- ♦ Windows 131
- ♦ ButtonPads (Toolbars) 138
- ♦ Integrating Your Application Into MapInfo Professional. . 145
- ♦ Performance Tips for the User Interface 147

Introduction to MapBasic User Interface Principles

By writing a MapBasic program, you can create a custom user interface for MapInfo Professional. A MapBasic program can control the following elements of the user interface:

- **Menus:** MapBasic programs can add custom menu items to existing menus, remove menus from the menu bar, and create entirely new menus.
- **Dialogs:** MapBasic programs can display custom dialog boxes, tailored to fit the users' needs.
- **Windows:** MapBasic programs can display standard types of MapInfo Professional windows (for example, Map and Browse windows) and customize the contents of those windows. MapBasic can also display messages in a special window (the Message window) and on the MapInfo Professional status bar.
- **ButtonPads** (also known as toolbars): MapBasic programs can add custom buttons to existing ButtonPads, or create entirely new ButtonPads. MapInfo Professional includes a special ButtonPad, Tools, to provide a place where MapBasic utilities can add custom buttons. For example, the ScaleBar application adds its custom button to the Tools pad.

The sample application, OverView, demonstrates many aspects of a custom interface created in MapBasic. When the user runs OverView, MapBasic adds custom items to the Tools menu. If the user chooses the custom Setup Overview menu item, MapBasic displays a custom dialog. If the user chooses a table from this dialog, MapBasic opens a new Map window to display the table.

Event-Driven Programming

MapBasic follows a programming model known as *event-driven programming*. To understand how a MapBasic program can create a custom user interface, you must first understand the basic principles of event-driven programming.

What Is an Event?

In a Graphical User Interface environment, the user controls what happens by typing and by using the mouse. Technically, we say that mouse-clicks and other actions taken by the user generate *events*. There are many different kinds of events; for example, when the user chooses a menu item, we say that the user has generated a menu-choose event, and when the user closes a window, we say the user has generated a window-close event.

What Happens When The User Generates A Menu Event?

When the user generates an event, the software must respond accordingly. Thus, when the user chooses a menu item, the software may need to display a dialog or, depending on which menu item the user chooses, the software may need to take some other action, such as opening or closing a table or a window. In general, when the user generates an event, we say that the software *handles* the event.

If a MapBasic application creates a custom menu, and the user chooses an item from that menu, the MapBasic application handles the menu-choose event. Typically, the MapBasic application handles the event by calling a procedure. In this situation, we say that the procedure acts as an *event-handler*, or *handler* for short.

Thus, creating custom menu items is typically a two-step process:

1. Customize the MapInfo Professional menu structure, using statements such as **Create Menu** or **Alter Menu**.
2. Specify a handler for each custom menu item. A handler can be a sub-procedure that appears elsewhere in your program. Set up each handler procedure to perform whatever tasks are appropriate for the corresponding menu item(s). Alternately, instead of specifying a procedure as the menu item's handler, you can specify that the menu item call a standard MapInfo Professional command. Thus, you could create a custom menu item that invokes the Create Thematic Map command (from MapInfo Professional's Map menu).

As noted in **Chapter 4: Using the Development Environment**, the **Call** statement lets you call a sub-procedure. However, when a sub-procedure acts as an event-handler, you do not issue any **Call** statements. Instead of issuing **Call** statements, you include a **Calling** clause within the **Create Menu** statement.

For example, the TextBox application issues the following **Create Menu** statement:

```
Create Menu "TextBox" As
  "&Create Text Boxes..." Calling create_sub,
  "Close TextBox" Calling Bye,
  "About TextBox..." Calling About
```

This statement creates a custom menu with several menu items, each of which contains a **Calling** clause (for example, Calling create_sub). Each **Calling** clause identifies the name of a procedure that appears elsewhere in the TextBox.MB program. Thus, create_sub, Bye, and About are all sub-procedure names.

When and if the user chooses the Create Text Boxes item from the TextBox menu, MapBasic automatically calls the create_sub procedure. Thus, the create_sub procedure acts as the handler for that menu item.

How Does a Program Handle ButtonPad Events?

Each button on a custom MapBasic ButtonPad has a handler procedure. Like the **Create Menu** statement, the **Create ButtonPad** statement contains a **Calling** clause which lets you designate a handler procedure. When the user works with a custom button, MapBasic calls the sub-procedure that you named in the **Create ButtonPad** statement.

MapBasic lets you create different types of custom buttons. With custom PushButtons, MapBasic calls the button's handler the moment the user chooses the button. With custom ToolButtons, MapBasic only calls the button's handler if the user chooses the tool and then clicks on a window. For more information, see the ButtonPads discussion later in this chapter.

How Does a Program Handle Dialog Events?

Custom MapBasic dialogs can call handler procedures. Thus, if you create a custom dialog that contains a check-box, MapBasic can call a handler procedure each time the user checks or clears the check-box. However, depending on your application, you may not need to create handlers for your dialogs. For a discussion of custom dialogs, see the discussion of Custom Dialogs later in this chapter.

Menus

Menus are an essential element of the graphical user interface. Accordingly, the MapBasic language lets you control every aspect of MapInfo Professional's menu structure. With a few lines of code, you can customize any or all of MapInfo Professional's menus or menu items.

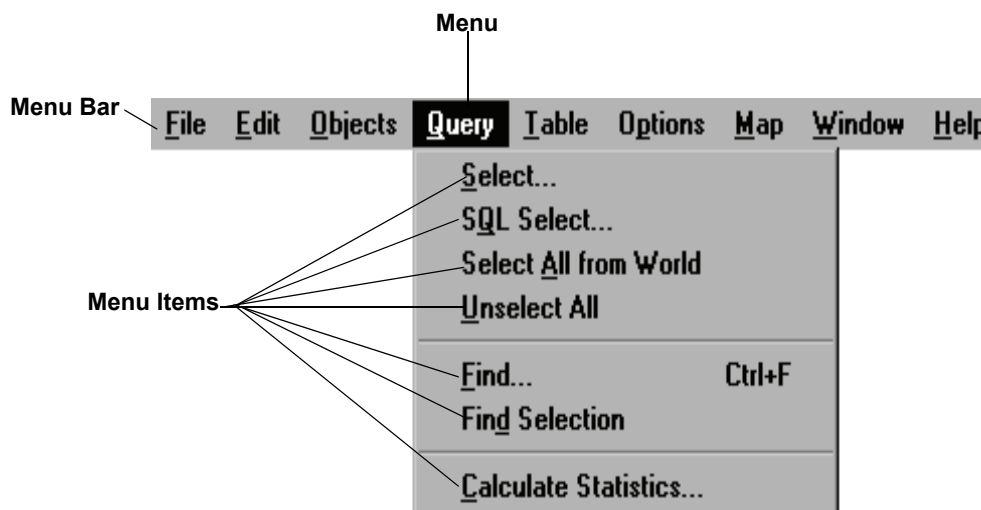
Menu Fundamentals

MapInfo Professional's menu structure consists of the following elements:

The **menu bar** is the horizontal bar across the top of the MapInfo Professional work area. The default MapInfo Professional menu bar contains words such as File, Edit, Objects, Query, etc.

A **menu** is a vertical list of commands that drops down if you click on the menu bar. For example, most applications include a File menu and an Edit menu.

A **menu item** is an individual command that appears on a menu. For example, the File menu typically contains menu items such as Open, Close, Save, and Print. Menu items are sometimes referred to as commands (for example, the File > Save command).



The concepts of menu, menu bar, and menu item are interrelated. Each menu is a set of menu items. For example, the File menu contains items such as Open, Close, Save, etc. The menu bar is a set of menus.

When the user chooses a menu item, some sort of action is initiated. Different menu items invoke different types of actions; some menu items cause dialog boxes to be displayed, while other menu items produce an immediate effect.

The action associated with a menu item is referred to as the menu item's **handler**. A menu item handler can either be a standard MapInfo Professional command code or a custom MapBasic sub-procedure name. In other words, when the user chooses a menu item, MapInfo Professional "handles" the menu-choose event, either by running a standard command code or by calling a sub-procedure from your application.

Adding New Items To A Menu

To add one or more custom items to an existing menu, use the **Alter Menu** statement.

For example, the following statement adds two custom menu items to the Query menu (one item called Annual Report, and another item called Quarterly Report):

```
Alter Menu "Query" Add
  "Annual Report"      Calling report_sub,
  "Quarterly Report"  Calling report_sub_q
```

For each of the custom menu items, the **Alter Menu** statement specifies a **Calling** clause. This clause specifies what should happen when and if the user chooses the menu item. If the user chooses the Annual Report item, MapInfo Professional calls the sub-procedure report_sub.

If the user chooses the Quarterly Report item, MapInfo Professional calls the sub-procedure report_sub_q. These sub-procedures (report_sub and report_sub_q) must appear elsewhere within the same MapBasic application.

You also can create custom menu items that invoke standard MapInfo Professional commands, rather than calling MapBasic sub-procedures. The definitions file menu.def contains a list of definitions of menu codes (for example, M_FILE_NEW and M_EDIT_UNDO). Each definition in that file corresponds to one of the standard MapInfo Professional menu commands (for example, M_EDIT_UNDO corresponds to the Edit menu's Undo command). If a menu item's **Calling** clause specifies one of the menu codes from menu.def, and the user chooses that menu item, MapInfo Professional invokes the appropriate MapInfo Professional command.

For example, the following statement defines a "Color Coded Maps" menu item. If the user chooses Color Coded Maps, MapInfo Professional runs the command code M_MAP_THEMATIC. In other words, if the user chooses the menu item, MapInfo Professional displays the Create Thematic Map dialog, just as if the user had chosen the Map > Create Thematic Map command.

```
Alter Menu "Query" Add
  "Color Coded Maps" Calling M_MAP_THEMATIC
```

Removing Items From A Menu

An application can remove individual menu items. The following statement removes the Delete Table item from MapInfo Professional's Table > Maintenance menu. Note that the identifier M_TABLE_DELETE is a code defined in the menu definitions file, menu.def.

```
Alter Menu "Maintenance" Remove M_TABLE_DELETE
```


If you want to remove several items from a menu, there are two techniques you can use: you can issue an **Alter Menu ... Remove** statement which lists all the items you wish to remove; or you can issue a **Create Menu ...** statement which redefines the menu entirely, including only the items you want.

For example, the following statement creates a simplified version of the Map menu that includes only three items (Layer Control, Previous View, and Options):

```
Create Menu "Map" As
  "Layer Control" Calling M_MAP_LAYER_CONTROL,
  "Previous View" Calling M_MAP_PREVIOUS,
  "Options" Calling M_MAP_OPTIONS
```

Creating A New Menu

To create an all-new menu, use the **Create Menu** statement. For example, the sample application, TextBox, issues the following **Create Menu** statement:

```
Create Menu "TextBox" As
  "&Create Text Boxes..." Calling create_sub,
  "Close TextBox" Calling Bye,
  "About TextBox..." Calling About
```

The **Create Menu** statement creates a new "TextBox" menu. However, the act of creating a menu does not cause the menu to appear automatically. To make the new menu become visible, you must take an additional step.

You could make the TextBox menu visible by adding it to the menu bar, using the **Alter Menu Bar** statement:

```
Alter Menu Bar Add "TextBox"
```

The **Alter Menu Bar Add** statement adds the menu to the right end of the menu bar. The menu produced would look like this:



In practice, adding menus onto the menu bar is sometimes problematic. The amount of space on the menu bar is limited, and every time you add a menu to the menu bar, you fill some of the remaining space. Therefore, for the sake of conserving space on the menu bar, the TextBox application uses a different technique for displaying its menu: instead of adding its menu directly onto the menu bar, the TextBox application uses an **Alter Menu** statement to add its menu as a hierarchical sub-menu, located on the Tools menu.

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

As a result of this statement, the TextBox menu appears as a hierarchical menu located on the Tools menu. The resulting Tools menu looks like this:



Sample programs that are provided with MapInfo Professional, such as ScaleBar and OverView, follow the same convention (placing their menu items on hierarchical menus located off of the Tools menu). Thus, if you run the TextBox application, the ScaleBar application, and the OverView application, all three applications add their commands to the Tools menu.

If each of the sample programs (ScaleBar, etc.) added a menu directly onto the menu bar, the menu bar would quickly become over-crowded. Stacking hierarchical menus onto the Tools menu (or any other menu) is one way of conserving space on the menu bar. Note, however, that some users find hierarchical menus significantly harder to use.

How you design and organize your menus will depend on the nature of your application. Depending on your application, you may need to add one, two, or even several menus to the menu bar.

Regardless of whether you attach your menus to the menu bar or to other menus, MapInfo Professional is limited to 96 menu definitions. In other words, there can never be more than 96 menus defined at one time, including MapInfo Professional's standard menus. This limitation applies even when you are not displaying all of the menus.

Altering A Menu Item

The MapBasic language lets you perform the following operations on individual menu items:

- You can disable (gray out) a menu item, so that the user cannot choose that menu item.
- You can enable a menu item that was formerly disabled.
- You can check a menu item (i.e., add a check-mark to the menu item); however, a menu item must be defined as "checkable" when it is created. To define a menu item as checkable, insert an exclamation point as the first character of the menu item name. For more information, see **Create Menu** in the *MapBasic Reference*.
- You can un-check a menu item (i.e., remove the check-mark)
- You can rename the menu item, so that the text that appears on the menu changes.

To alter a menu item, use the **Alter Menu Item** statement. The **Alter Menu Item** statement includes several optional clauses (Enable, Disable, Check, UnCheck, etc.); use whichever clauses apply to the change you want to make.

The sample program OverView demonstrates the process of creating, then altering, a custom menu. The OverView application creates the following custom menu:

```
Create Menu "OverView" As
  "&Setup OverView"    Calling OverView,
  "(Suspend Tracking"  Calling MenuToggler,
  "(Pick Frame Style"  Calling PickFrame,
  "(-",
  "Close Overview"     Calling Bye,
  "(-",
  "About Overview..." Calling About
```

The Pick Frame Style menu item is initially disabled. (Whenever the name of a menu item begins with the "(" character, that menu item is automatically disabled when the menu first appears.)

When and if the user sets up an overview window, the OverView application enables the Pick Frame Style menu item, using the following statement:

```
Alter Menu Item PickFrame Enable
```

If the user closes the overview window, the application once again disables the Pick Frame menu item, by issuing the following statement:

```
Alter Menu Item PickFrame Disable
```

PickFrame is the name of a sub-procedure in overview.mb. Note that PickFrame appears in both the **Create Menu** statement (in the Calling clause) and in the **Alter Menu Item** statements. When you issue an **Alter Menu Item** statement, you must specify which menu item you want to alter. If you specify the name of a procedure (for example, PickFrame), MapInfo Professional modifies whatever menu item calls that procedure.

Similarly, to enable the Suspend Tracking menu item, issue the following statement:

```
Alter Menu Item MenuToggler Enable
```

You also can use **Alter Menu Item** to change the name of a menu item. For example, the OverView application has a menu item that is initially called Suspend Tracking. If the user chooses Suspend Tracking, the application changes the menu item's name to Resume Tracking by issuing the following statement:

```
Alter Menu Item MenuToggler Text "Resume Tracking"
```

Note that MapInfo Professional enables and disables its own standard menu items automatically, depending on the circumstances. For example, the Window > New Map Window command is only enabled when and if a mappable table is open. Because MapInfo Professional automatically alters its own standard menu items, a MapBasic application should not attempt to enable or disable those menu items.

Re-Defining The Menu Bar

To remove an entire menu from the menu bar, use the **Alter Menu Bar** statement. For example, the following statement causes the Query menu to disappear:

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

You also can use **Alter Menu Bar** to add menus to the menu bar. For example, the following statement adds both the Map menu *and* the Browse menu to the menu bar. (By default, those two menus never appear on the menu bar at the same time. The Map menu ordinarily appears only when a Map is the active window, and the Browse menu ordinarily appears only when a Browser window is active.)

```
Alter Menu Bar Add "Map", "Browse"
```

The **Alter Menu Bar Add** statement always adds menus to the right end of the menu bar. One minor disadvantage of this behavior is the fact that menus can end up located to the right of the Help menu. Most software packages arrange the menu bar so that the last two menu names are **Window** and **Help**. Therefore, you may want to insert your custom menu to the left of the **Window** menu. The following statements show how to insert a menu to the left of the Window menu:

```
Alter Menu Bar Remove ID 6, ID 7  
Alter Menu Bar Add "Tools", ID 6, ID 7
```

The first statement removes the Window menu (ID 6) and Help menu (ID 7) from the menu bar. The second statement adds the Tools menu, the Window menu, and the Help menu to the menu bar. The end result is that the Tools menu is placed to the left of the Window menu.

For complete control over the menu order, use the **Create Menu Bar** statement. For example, this statement re-defines the menu bar to include the File, Edit, Map, Query, and Help menus (in that order):

```
Create Menu Bar As "File", "Edit", "Map", "Query", "Help"
```

For a list of MapInfo Professional's standard menu names ("File", "Query" etc.) see **Alter Menu** in the MapBasic *Reference* or online Help. To restore MapInfo Professional's standard menu definitions, issue a **Create Menu Bar As Default** statement.

Specifying Language-Independent Menu References

Most of the preceding examples refer to menus by their names (for example, "File"). There is an alternate syntax for referring to MapInfo Professional's standard menus: you can identify standard menus by ID numbers. For example, in any menu-related statement where you might refer to the File menu as "File", you could instead refer to that menu as ID 1. Thus, the following statement removes the Query menu (which has ID number 3) from the menu bar:

```
Alter Menu Bar Remove ID 3
```

If your application will be used in more than one country, you may want to identify menus by their ID numbers, rather than by their names. When the MapInfo Professional software is localized for non-English speaking countries, the names of menus are changed. If your application tries to alter the "File" menu, and you run your application on a non-English version of MapInfo Professional, your application may generate an error (because in a non-English version of MapInfo Professional, "File" may not be the name of the menu). For a listing of the ID numbers that correspond to MapInfo Professional's standard menus, see **Alter Menu** in the MapBasic *Reference* or online Help.

Customizing MapInfo Professional's Shortcut Menus

MapInfo Professional 4.0 provides shortcut menus. These menus appear if the user clicks the right mouse button. To manipulate shortcut menus, use the same statements you would use to manipulate conventional menus: **Alter Menu**, **Alter Menu Item**, and **Create Menu**.

Each shortcut menu has a unique name and ID number. For example, the shortcut menu that appears when you right-click a Map window is called "MapperShortcut" and has an ID of 17. For a listing of the names and ID numbers of the shortcut menus, see **Alter Menu** in the MapBasic *Reference* or online Help.

To destroy a shortcut menu, use the **Create Menu** statement to re-define the menu, and specify the control code "(" as the new menu definition. For example:

```
Create Menu "MapperShortcut" ID 17 As "("
```

Assigning One Handler Procedure To Multiple Menu Items

The **Create Menu** and **Alter Menu** statements provide an optional **ID** clause, which lets you assign a unique ID number to each custom menu item you create. Menu item IDs are optional. However, if you intend to have two or more menu items calling the same handler procedure, you will probably want to assign a unique ID number to each of your custom menu items.

In situations where two or more menu items call the same handler procedure, the handler procedure generally calls **CommandInfo()** to determine which item the user chose. For example, the following statement creates two custom menu items that call the same handler:

```
Alter Menu "Query" Add
  "Annual Report"      ID 201 Calling report_sub,
  "Quarterly Report"   ID 202 Calling report_sub
```

Both menu items call the procedure `report_sub`. Because each menu item has a unique ID, the handler procedure can call **CommandInfo()** to detect which menu item the user chose, and act accordingly:

```
Sub report_sub
  If CommandInfo(CMD_INFO_MENUITEM) = 201 Then
    '
    ' ... then the user chose Annual Report...
    '
  ElseIf CommandInfo(CMD_INFO_MENUITEM) = 202 Then
    '
    ' ... then the user chose Quarterly Report...
    '
  End If
End Sub
```

Menu item IDs also give you more control when it comes to altering menu items. If an **Alter Menu Item** statement identifies a menu item by the name of its handler procedure, MapBasic modifies *all* menu items that call the same procedure. Thus, the following statement disables both of the custom menu items defined above (which may not be the desired effect):

```
Alter Menu Item report_sub Disable
```

Depending on the nature of your application, you may want to modify only one of the menu items. The following statement disables only the Annual Report menu item, but has no effect on any other menu items:

```
Alter Menu Item ID 201 Disable
```

Menu item ID numbers can be any positive Integer.

Simulating Menu Selections

To activate a MapInfo Professional command as if the user had chosen that menu item, use the **Run Menu Command** statement. For example, the following statement displays MapInfo Professional's Open Table dialog, as if the user had chosen File > Open Table:

```
Run Menu Command M_FILE_OPEN
```

The code M_FILE_OPEN is defined in menu.def.

Defining Shortcut Keys And Hot Keys

Shortcut keys are keystroke combinations that let the user access menus and menu items directly from the keyboard, without using the mouse. Typically, a shortcut key appears as an underlined letter in the name of the menu or menu item. For example, in Windows, the shortcut keystroke to activate the MapInfo Professional File menu is <Alt-F>, as indicated by the underlined letter, F. To assign a shortcut key to a menu item, place an ampersand (&) directly before the character that you want to define as the shortcut key.

The following program fragment shows how a MapBasic for Windows program defines the C key (in Create Text Boxes) as a shortcut key. If this program runs on MapInfo Professional for Macintosh, the ampersand is ignored.

```
Create Menu "TextBox" As
  "&Create Text Boxes..." Calling create_sub,
  ...
```

Hot keys are keystroke combinations that let the user execute menu commands without activating the menu. Unlike shortcut keys that let you traverse through the menu structure using the keyboard, hot keys let you avoid the menu completely. The following program fragment adds the hot key combination <Control-Z> to a custom menu item:

Hot keys are keystroke combinations that let the user execute menu commands without activating the menu. The following program fragment adds the hot key combination <Command-Z> to a custom menu item:

```
Alter Menu "Query" Add
  "New Report" + Chr$(9) + "CTRL-Z/W^%122" Calling new_sub
```

The instruction + Chr\$(9) tells MapBasic to insert a tab character. The tab character is used for formatting, so that all of the menu's hotkey descriptions appear aligned.

The text CTRL-Z appears on the menu, so that the user can see the menu item has a hot key.

The instruction `/w^%122` defines the hot key as <Control-Z>. The code `/w^%122` is a hot key code recognized by MapInfo Professional: `/w` specifies that the code is for MapInfo Professional for Windows, the caret (^) specifies that the user should hold down the Ctrl key, and the `%122` specifies the letter “z” (122 is the ASCII character code for z).

```
Alter Menu "Query" Add
  "New Report /Mz" Calling new_sub
```

The instruction `/Mz` defines the hot key as Command-Z. `/M` specifies that the code is for MapInfo Professional for Macintosh, and `z` specifies the z key.

For a listing of codes that control menu hot keys, see **Create Menu** in the MapBasic *Reference* or online Help.

Controlling Menus Through the MapInfo Professional Menus File

The default menu structure of MapInfo Professional is controlled by the MapInfo Professional menus file. If you want to customize MapInfo Professional's menu structure, you can do so by altering the menus file.

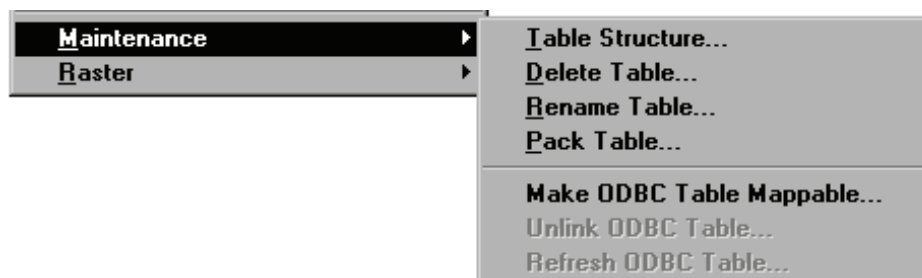
With MapInfo Professional for Windows, the menus file is called **MAPINFOW.MNU**. With MapInfo Professional for Macintosh, the menus file is called **MapInfo menus**.

Since the menus file is a text file, you can view it in any text editor. If you examine the menus file, you will see that it bears a strong resemblance to a MapBasic program. If you change the menu definitions in the menus file, the menus will look different the next time you run MapInfo Professional. In other words, altering the menus file gives you a way of customizing the menu structure without using a compiled MapBasic application.

WARNING: Before you make any changes to the menus file, make a backup of the file. If the menus file is corrupted or destroyed, you will not be able to run MapInfo Professional (unless you can restore the menus file from a backup). If you corrupt the menus file, and you cannot restore the file from a backup, you will need to re-install MapInfo Professional.

The menus file contains several **Create Menu** statements. These statements define MapInfo Professional's standard menu definitions (File, Edit, etc.). If you wish to remove one or more menu items from a menu, you can do so by removing appropriate lines from the appropriate **Create Menu** statement.

For example, MapInfo Professional's Table > Maintenance menu usually contains a Delete Table command, as shown below.



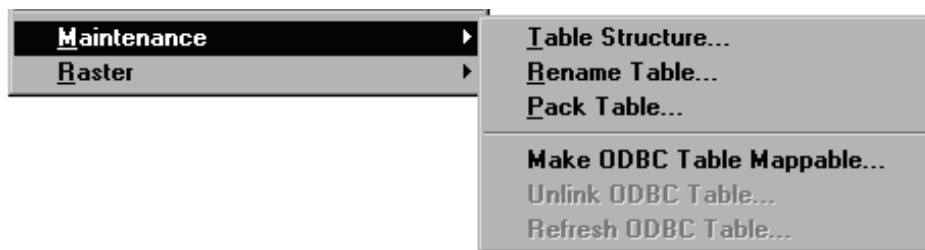
If you examine the menus file, you will see that the Maintenance menu is defined through a **Create Menu** statement that looks like this:

```
Create Menu "&Maintenance" As
  "&Table Structure..." Calling 404,
  "&Delete Table..."      Calling 409,
  "&Rename Table..."      Calling 410,
  "&Pack Table..."        Calling 403,
  . . .
```

Because the Delete Table command is potentially dangerous, you might want to re-define the Maintenance menu to eliminate Delete Table. To eliminate the Delete Table command from the menu, remove the appropriate line ("&Delete Table..." Calling 409) from the menus file. After you make this change, the **Create Menu** statement will look like this:

```
Create Menu "&Maintenance" As
  "&Table Structure..." Calling 404,
  "&Rename Table..."    Calling 410,
  "&Pack Table..."      Calling 403,
  . . .
```

The next time you run MapInfo Professional, the Table > Maintenance menu will appear without a Delete Table item.



Similarly, if you wish to remove entire menus from the MapInfo Professional menu bar, you can do so by editing the **Create Menu Bar** statement that appears in the menus file.

If MapInfo Professional is installed on a network, and you modify the menus file in the directory where MapInfo Professional is installed, the changes will apply to all MapInfo Professional users on the network. In some circumstances, you may want to create different menu structures for different network users. For example, you may want to eliminate the Delete Table command from the menu that appears for most of your users, but you may want that command to remain available to your network system administrator.

To assign an individual user a customized menu structure, place a customized version of the menus file in that user's "home" directory. For Windows users, the home directory is defined as the user's private Windows directory (i.e., the directory where WIN.INI resides).

To assign an individual user a customized menu structure, place a customized version of the menus file in that user's "home" directory/folder. For Macintosh users, the home directory is defined as the location of the user's System folder. The menus file can be placed directly in the System folder, or in the Preferences folder within the System folder.

When a user runs MapInfo Professional, it checks to see if a copy of the menus file exists in the user's home directory. If a copy of the menus file is present in the user's home directory, MapInfo Professional loads that set of menus. If there is no menus file in the user's home directory, MapInfo Professional loads the menus file from the directory where it is installed.

Thus, if you want different users to see two different versions of the menu structure, create two different versions of the menus file. Place the version that applies to most of your users in the directory where MapInfo Professional is installed. Place the version that applies only to individual users in the home directories of the individual users.

Standard Dialog Boxes

Dialog boxes are an essential element of the user interface. MapBasic provides several different statements and functions that let you create dialogs for your application.

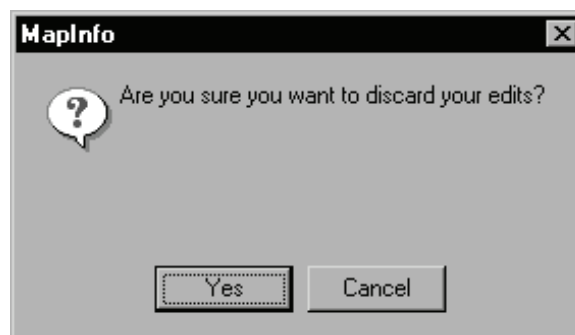
Displaying a Message

Use the **Note** statement to display a simple dialog box with a message and an OK button.



Asking a Yes-or-No Question

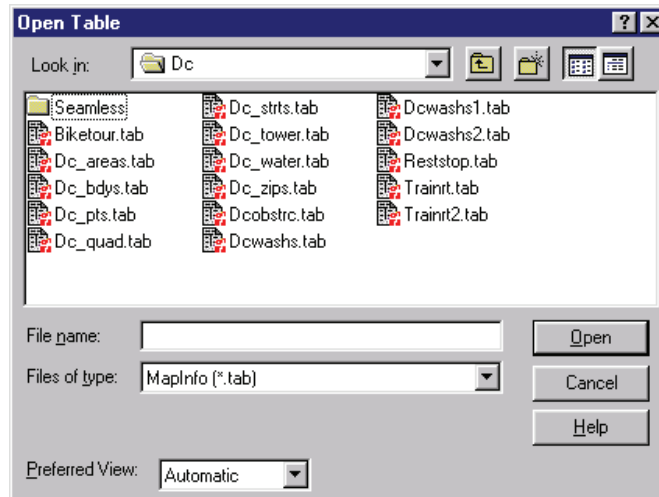
Use the **Ask()** function to display a dialog with a prompt and two buttons. The two buttons usually say OK and Cancel, but you can customize them to suit your application. If the user chooses the OK button, the function returns a TRUE value, otherwise, the function returns FALSE.



Selecting a File

Call the **FileOpenDlg()** function to display a standard File Open dialog. If the user chooses a file, the function returns the name of the chosen file. If the user cancels out of the dialog, the function returns an empty string.

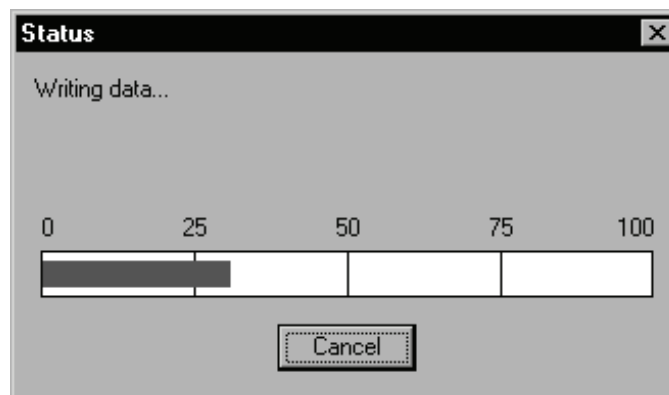
The **FileOpenDlg()** function produces a dialog that looks like this:



The **FileSaveAsDlg()** function displays a standard File Save As dialog, and returns the file name entered by the user.

Indicating the Percent Complete

Use the **ProgressBar** statement to display a standard percent-complete dialog, containing a progress bar and a Cancel button.



Displaying One Row From a Table

MapInfo Professional does not provide a standard dialog that displays one row from a table. However, you can use MapInfo Professional's Info window to display a row. Instructions on managing the Info window (and other windows as well) appear later in this chapter.

For more information about the statements and functions listed above, see the *MapBasic Reference*. If none of the preceding statements meets your needs, use the **Dialog** statement to create a custom dialog, as described in the following section.

Custom Dialog Boxes

The **Dialog** statement lets you create custom dialogs. When you issue a **Dialog** statement, MapInfo Professional displays the dialog and lets the user interact with the dialog. When the user dismisses the dialog (for example, by clicking the OK or Cancel button), MapInfo Professional executes any statements that follow the **Dialog** statement. After the **Dialog** statement, you can call the **CommandInfo()** function to tell whether the user chose OK or Cancel.

Everything that can appear on a dialog is known as a *control*. For example, every OK button is a control, and every Cancel button is also a control. To add controls to a dialog, include **Control** clauses within the **Dialog** statement. For example, the following statement creates a dialog with four controls: a label (known as a StaticText control); a box where the user can type (known as an EditText control); an OK push-button (known as OKButton control) and a Cancel push-button (CancelButton control).

```
Dim s_searchfor As String

Dialog
  Title "Search"
  Control StaticText
    Title "Enter string to find:"
  Control EditText
    Into s_searchfor
  Control OKButton
  Control CancelButton
  Control CancelButton
  Control OKButton
If CommandInfo(CMD_INFO_DLG_OK) Then
  ' ... then the user clicked OK -- in which case,
  ' the String variable: s_searchfor will contain
  ' the value entered by the user.
End If
```

This **Dialog** statement produces the following dialog:



Sizes and Positions of Controls

If you want to change the size of a dialog control, you can include the optional **Width** and **Height** clauses within the **Control** clause. If you want to change the position of a dialog control, you can include the optional **Position** clause.

For example, you might not like the default placement of the buttons in the dialog shown above. To control the button placement, you could add **Position** clauses, as shown below:

```
Dialog
  Title "Search"
  Control StaticText
    Title "Enter string to find:"
  Control EditText
    Into s_searchfor

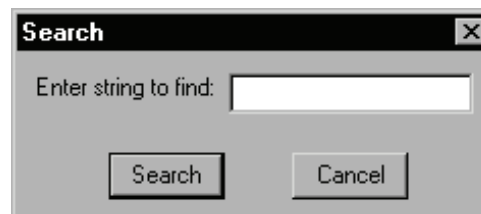
  Control OKButton
    Title "Search"
    Position 30, 30

  Control CancelButton
    Position 90, 30

  Control CancelButton
    Position 80, 30

  Control OKButton
    Title "Search"
    Position 120, 30
```

Because two of the **Control** clauses include **Position** clauses, the dialog's appearance changes:



Positions and sizes are stated in terms of **dialog units**, where each dialog unit represents one quarter of a character's width or one eighth of a character's height. The upper-left corner of the dialog has the position 0, 0. The following **Position** clause specifies a position in the dialog five characters in from the left edge of the dialog, and two characters from the top edge of the dialog:

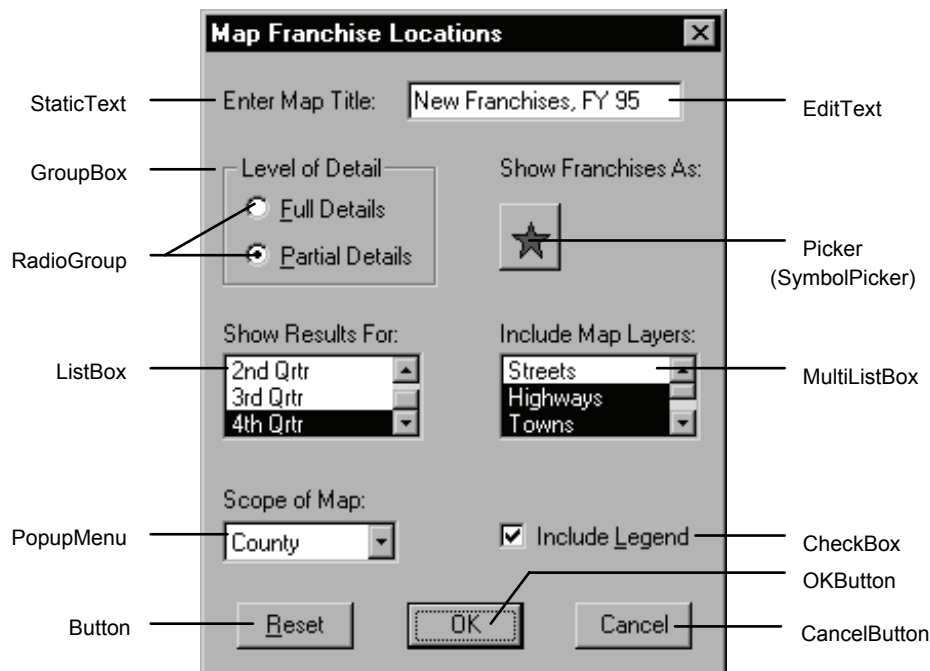
```
Position 20, 16
```

A horizontal position of 20 specifies a position five characters to the right, since each dialog unit represents one fourth of the width of a character. A vertical position of 16 specifies a position two characters down, since each dialog unit spans one eighth of the height of a character.

You can include a **Position** clause for every control in the dialog. You also can specify **Width** and **Height** clauses to customize a control's size.

Control Types

The previous examples contained four types of controls (StaticText, EditText, OKButton, and CancelButton). The following illustration shows all of MapBasic's dialog control types.



StaticText is a non-interactive control that lets you include labels in the dialog box. For example:

```
Control StaticText
  Title "Enter map title:"
  Position 5, 10
```

An **EditText** control is a boxed area where the user can type. For example:

```
Control EditText
  Value "New Franchises, FY 95"
  Into s_title
  ID 1
  Position 65, 8   Width 90
```

A **GroupBox** control is a rectangle with a label at the upper left corner. Use GroupBoxes for visual impact, to convey that other dialog controls are related. For example:

```
Control GroupBox
  Title "Level of Detail"
  Position 5, 30   Width 70   Height 40
```

A **RadioGroup** control is a set of "radio buttons" (i.e., a list of choices where MapBasic only allows the user to select one of the buttons at a time). For example:

```
Control RadioGroup
  Title "&Full Details;&Partial Details"
  Value 2
  Into i_details
  ID 2
  Position 12, 42   Width 60
```

There are four types of Picker controls: **PenPicker**, **BrushPicker**, **FontPicker**, and **SymbolPicker**. Each Picker control lets the user select a graphical style (line, fill, font, or symbol). The illustration shown above includes a **SymbolPicker** control, showing a star-shaped symbol. For example:

```
Control SymbolPicker
  Position 95, 45
  Into sym_variable    ID 3
```

A **ListBox** control is a scrollable list from which the user can select one item. MapBasic automatically appends a vertical scroll bar to the right edge of the ListBox if there are too many list items to be displayed at one time. For example:

```
Control ListBox
  Title "First Qtrtr;2nd Qtrtr;3rd Qtrtr;4th Qtrtr"
  Value 4
  Into i_quarter
  ID 4
  Position 5, 90    Width 65    Height 35
```

A **MultiListBox** is similar to a ListBox, except that the user can shift-click or control-click to select two or more items from the list. For example:

```
Control MultiListBox
  Title "Streets;Highways;Towns;Counties;States"
  Value 3
  ID 5
  Position 95, 90    Width 65    Height 35
```

A **PopupMenu** appears as a text item with a down arrow at the right edge. As the user clicks on the control, a menu pops up, allowing the user to make a selection. For example:

```
Control PopupMenu
  Title "Town;County;Territory;Entire state"
  Value 2
  Into i_scope
  ID 6
  Position 5, 140
```

A **CheckBox** is a label with a box. The user can check or clear the box by clicking on the control. For example:

```
Control CheckBox
  Title "Include &Legend"
  Into l_showlegend
  ID 7
  Position 95, 140
```

Button controls are perhaps the most common type of control that you will use, since almost every dialog box has at least one button. MapBasic provides special control types **OKButton** and **CancelButton** for creating OK and Cancel buttons.

```
Control Button
  Title "&Reset"
  Calling reset_sub
  Position 10, 165

Control OKButton
  Position 65, 165
  Calling ok_sub

Control CancelButton
  Position 120, 165
```

Each dialog should have no more than one **OKButton** or **CancelButton** control. Both controls are optional. However, as a general rule, every dialog should have an OK and/or a Cancel button, so that the user has a way of dismissing the dialog. If either control has a handler, MapBasic executes the handler procedure and then resumes executing the statements that follow the **Dialog** statement.

Every type of control is described in detail in the MapBasic *Reference* and online Help. For example, to read about ListBox controls, see **Control Listbox**.

Specifying a Control's Initial Value

Most types of controls have an optional **Value** clause. This clause specifies how the control is set when the dialog first appears. For example, if you want the fourth item in a ListBox control to be selected when the dialog first appears, add a **Value** clause to the **ListBox** clause:

```
Value 4
```

If you omit the **Value** clause, MapInfo Professional uses a default value. For example, CheckBox controls are checked by default. For more information about setting a **Value** clause, see the appropriate **Control** description (for example, **Control CheckBox**) in the MapBasic *Reference*.

Reading a Control's Final Value

Most types of controls allow an optional **Into** clause. This clause associates a program variable with the control, so that MapInfo Professional can store the dialog data in the variable. If you create a control with an **Into** clause, and if the user terminates the dialog by clicking the OK button, MapInfo Professional stores the control's final value in the variable.

The **Into** clause must name a local or global variable in your program. The variable that you specify must be appropriate for the type of control. For example, with a CheckBox control, the variable must be Logical (TRUE meaning checked, FALSE meaning clear). See the MapBasic *Reference* for more information about the type of variable appropriate for each control.

Note: MapInfo Professional only updates the **Into** variable(s) after the dialog terminates, and only if the dialog terminates because the user clicked OK. If you need to read the value of a control from within a dialog handler procedure, call the **ReadControlValue()** function.

Responding to User Actions by Calling a Handler Procedure

Most types of controls can have *handlers*. A handler is a sub-procedure that MapBasic calls automatically when and if the user clicks that control. The optional **Calling handler** clause specifies a control's handler; *handler* must be the name of a sub-procedure that takes no parameters. When the user clicks on a control that has a handler procedure, MapBasic calls the procedure. When the procedure finishes, the user can continue interacting with a dialog (except in the case of OKButton and CancelButton controls, which automatically dismiss the dialog).

Handler procedures allow your program to issue statements while the dialog is on the screen. For example, you may want your dialog to contain a "Reset" button. If the user clicks on the Reset button, your program will reset all controls in the dialog to their default values. To create such a dialog, you would need to assign a handler procedure to the "Reset" Button control. Within the handler procedure, you would issue **Alter Control** statements to reset the dialog's controls.

A **ListBox** or **MultiListBox** control handler can be set up to respond one way to single-click events while responding differently to double-click events. The handler procedure can call the **CommandInfo(CMD_INFO_DLG_DBL)** function to determine whether the event was a single- or double-click. For an example of this feature, see the Named Views sample program (nviews.mb). The Named Views dialog presents a list of names; if the user double-clicks on a name in the list, the handler procedure detects that there was a double-click event, and dismisses the dialog. In other words, the user can double-click on the list, rather than single-clicking on the list and then clicking on the OKButton.

If two or more controls specify the same procedure name in the **Calling** clause, the named procedure acts as the handler for both of the controls. Within the handler procedure, call the **TriggerControl()** function to determine the ID of the control that was used.

Most dialog controls can have handler procedures (only GroupBox, StaticText, and EditText controls cannot have handlers). You also can specify a special handler procedure that is called once when the dialog first appears. If your **Dialog** statement includes a **Calling** clause that is not part of a **Control** clause, the **Calling** clause assigns a handler procedure to the dialog itself.

The **Alter Control** statement may only be issued from within a handler procedure. Use **Alter Control** to disable, enable, show, hide, rename, or reset the current setting of a control. The **Alter Control** statement can also set which EditText control has the focus (i.e., which control is active). For more information, see **Alter Control** in the MapBasic *Reference* or online Help.

Enabled / Disabled Controls

When a control first appears, it is either enabled (clickable) or disabled (grayed out). By default, every control is enabled. There are two ways to disable a dialog control:

- Include the optional **Disable** keyword within the **Dialog** statement's **Control** clause. When the dialog appears, the control is disabled.
- From within a handler procedure, issue an **Alter Control** statement to disable the control. If you want the control to be disabled as soon as the dialog appears, assign a handler procedure to the dialog itself, by including a **Calling** clause that is not within a **Control** clause. This handler will be called once, when the dialog first appears. Within the handler, you can issue **Alter Control** statements. This technique is more involved, but it is also more flexible. For

example, if you want a control to be disabled, but only under certain conditions, you can place the **Alter Control** statement within an **If...Then** statement.

Note: If you are going to use an **Alter Control** statement to modify a dialog control, you should assign an ID number to the control by including an **ID** clause in the **Dialog** statement. For an example, see **Alter Control** in the MapBasic *Reference* or online Help.

Letting the User Choose From a List

The **ListBox** control presents a list of choices. There are two ways you can specify the list of items that should appear in a **ListBox** control:

- Build a String expression that contains all of the items in the list, separated by semicolons. For example:

```
Control ListBox
  Title "First Qrtr;2nd Qrtr;3rd Qrtr;4th Qrtr;Year in Review"
```

- Declare an array of String variables, and store each list item in one element of the array. In the **Control** clause, specify the keywords **From Variable**. For example, if you have created a String array called `s_list`, you could display the array in a **ListBox** control using this syntax:

```
Control ListBox
  Title From Variable s_list
```

You can use the **From Variable** syntax in all three of MapBasic's list controls (**ListBox**, **MultiListBox**, and **PopupMenu**).

Managing MultiListBox Controls

If your dialog contains a **MultiListBox** control, you must use a handler procedure to determine what list item(s) the user selected from the list. In most cases, a dialog with a **MultiListBox** control contains an **OKButton** control with a handler procedure. The **OKButton**'s handler procedure calls the **ReadControlValue()** function within a loop. The first **ReadControlValue()** call returns the number of the first selected list item; the next call returns the number of the next selected list item; etc. When **ReadControlValue()** returns zero, the list of selected items has been exhausted. If **ReadControlValue()** returns zero the first time you call it, none of the list items are selected.

Within a handler procedure, you can de-select all items in a **MultiListBox** control by issuing an **Alter Control** statement, and assigning a value of zero to the control. To add a list item to the set of selected items, issue an **Alter Control** statement with a positive, non-zero value. For example, to select the first and second items in a **MultiListBox** control, you could issue the following statements:

```
Alter Control 1 Value 1
Alter Control 1 Value 2
```

Note that both the **ReadControlValue()** function and the **Alter Control** statement require a control ID. To assign a control ID to a **MultiListBox** control, include the optional **ID** clause in the **Control MultilistBox** clause.

Specifying Shortcut Keys for Controls

When a MapBasic application runs on MapInfo Professional for Windows, the application dialogs can assign shortcut keys to the various controls. A shortcut key is a convenience that lets the user activate a dialog control using the keyboard instead of the mouse.

To specify a shortcut key for a control, include the ampersand character (&) in the control's title immediately before the character that is to be used as a shortcut key character. For example, the following **Control** clause creates a Button control with R as the shortcut key:

```
Control Button
  Title "&Reset"
  Calling reset_sub
```

Because an ampersand appears in the Button control's title, the user is able to activate the Reset button by pressing Alt-R. If you want to display an ampersand character in a control, use two successive ampersand characters (&&).

You cannot specify a shortcut key for an EditText control. However, if you place a StaticText label to the left of an EditText control, and you specify a shortcut key for the StaticText label, the user can set the focus on the EditText control by pressing the shortcut key of the StaticText label.

Dialog shortcut key designations are ignored when a MapBasic application runs in any operating environment other than Windows.

Modal vs. Modeless Dialog Boxes

The **Dialog** statement creates a *modal* dialog box. In other words, the user must dismiss the dialog box (for example, by clicking OK or Cancel) before doing anything else with MapInfo Professional.

Some dialog boxes are *modeless*, meaning that the dialog can remain on the screen while the user performs other actions. For example, MapInfo Professional's Image Registration dialog box is modeless. The **Dialog** statement cannot create modeless dialog boxes. If you want to create modeless dialog boxes, you may need to develop an application in another programming environment, such as Microsoft Visual Basic, and call that application from within your MapBasic program (for example, using the **Run Program** statement).

Terminating a Dialog Box

After a MapBasic program issues a **Dialog** statement, it will continue to be displayed until one of four things happens:

- The user clicks the dialog's **OKButton** control (if the dialog has one).
- The user clicks the dialog's **CancelButton** control (if the dialog has one).
- The user otherwise cancels the dialog (for example, by pressing the Escape key).
- The user clicks a control that has an associated handler procedure that issues a **Dialog Remove** statement.

Ordinarily, a dialog terminates when the user clicks an **OKButton** or **CancelButton** control. There are times when the user should be allowed to continue using a dialog after pressing OK or Cancel. For example, in some dialogs if the user presses Cancel, the application asks the user to verify the cancellation (Are you sure you want to lose your changes?). If the user's response is No, the application should resume using the original dialog.

The **Dialog Preserve** statement lets you allow the user to continue using a dialog after the **OKButton** or **CancelButton** is clicked. You can only issue a **Dialog Preserve** statement from within the handler sub-procedure of either the **OKButton** or **CancelButton** control.

The **Dialog Remove** statement halts a dialog prematurely. When a control's handler procedure issues a **Dialog Remove** statement, the dialog halts immediately. **Dialog Remove** is only valid from within a dialog control's handler procedure. **Dialog Remove** can be used, for instance, to terminate a dialog when the user double-clicks a **ListBox** control. The Named Views sample program (NIEWS.MB) provides an example of allowing the user to double-click in a list.

Windows

A MapBasic application can open and manipulate any of MapInfo Professional's standard window types (Map windows, Browse windows, etc.).

To open a new document window, issue one of these statements: **Map**, **Browse**, **Graph**, **Layout**, or **Create Redistrict**. Each document window displays data from a table, so you must have the proper table(s) open before you open the window.

To open one of MapInfo Professional's other windows (for example, the Help window or the Statistics window), use the **Open Window** statement.

Many window settings can be controlled through the **Set Window** statement. For example, you could use the **Set Window** statement to set a window's size or position. There are also other statements that let you configure attributes of specific window types. For example, to control the order of layers in a Map window, you would issue a **Set Map** statement. To control the display of a grid in a Browse window, you would issue a **Set Browse** statement.

Each document window (Map, Browser, Layout, Graph, or Redistrict) has an Integer identifier, or window ID. Various statements and functions require a window ID as a parameter. For example, if two or more Map windows are open, and you want to issue a **Set Map** statement to modify the window, you should specify a window ID so that MapInfo Professional knows which window to modify.

To obtain the window ID of the active window, call the **FrontWindow()** function. Note that when you first open a window (for example, by issuing a **Map** statement), that new window is the active window. For example, the OverView sample program issues a **Map** statement to open a Map window, and then immediately calls the **FrontWindow()** function to record the ID of the new Map window. Subsequent operations performed by the OverView application refer to the ID.

Note: A window ID is not a simple, ordinal number, such as 1, 2, etc. The number 1 (one) is not a valid window ID. To obtain a window ID, you must call a function such as **FrontWindow()** or **WindowID()**. For example, to obtain the window ID of the first window that is open, call **WindowID(1)**. To determine the number of open windows, call **NumWindows()**.

The **WindowInfo()** function returns information about an open window. For example, if you want to determine whether the active window is a Map window, you can call **FrontWindow()** to determine the active window's ID, and then call **WindowInfo()** to determine the active window's window type.

To close a window, issue a **Close Window** statement.

Specifying a Window's Size and Position

There are two ways to control a window's size and position:

- Include the optional **Position**, **Width**, and **Height** clauses in the statement that opens the window. For example, the following **Map** statement not only opens a Map window, it also specifies the window's initial size and position:

```
Map From world
  Position (2,1) Units "in"
  Height 3 Units "in"
  Width 4 Units "in"
```
- Issue a **Set Window** statement to control a window's size or position after the window is open. Note that the **Set Window** statement requires an Integer window ID.

Map Windows

A Map window displays mappable objects from one or more tables. When opening a Map window, you must specify the tables that you want to display; each table must already be open.

The following statement opens a Map window:

```
Map From world, worldcap, grid30
```

This example maps the objects from the World, Worldcap, and Grid30 tables.

To add layers to a Map window, issue an **Add Map Layer** statement. To remove map layers from a Map window, issue a **Remove Map Layer** statement. If you want to temporarily hide a map layer, you do not need to remove it from the map; instead, you can use the **Set Map** statement to set that layer's Display attribute to off.

The **Set Map** statement is a very powerful statement that can control many aspects of a Map window. By issuing **Set Map** statements, your program can control map attributes that the user would control through the Map > Layer Control and Map > Options commands. For more information, see **Set Map** in the *MapBasic Reference*.

Use the **Shade** statement to create a thematic map (a map that uses color coding or other graphical devices to display information about the data attached to the map). The **Shade** statement lets you create the following of MapInfo Professional's styles of thematic maps: ranges, bar charts, pie charts, graduated symbols, dot density, or individual values. When you create a thematic map, MapInfo Professional adds a thematic layer to the affected window. To modify a thematic map, use the **Set Shade** statement.

As of version 5.0 use the **Create Grid** Statement to create an important new thematic type that enables analysis unconstrained by pre-existing geographic boundaries. Surface themes provide a continuous color visualization for point data sets that you previously looked at as a point thematic or graduated symbol. An inverse distance weighted interpolator populates the surface values from your MapInfo Professional point table. This powerful new thematic can be used in many industries like telco, retail analysis, insurance, traditional GIS areas, and many more. This new theme and grid format is supported by open APIs for additional grid formats and interpolators which allows customization by our developer community. Refer to the **Create Grid** statement in the *MapBasic Reference*. To modify a surface thematic, use the **Inflect** clause of the **Set Map** statement.

To change a Map window's projection, you can issue a **Set Map** statement with a **CoordSys** clause. Alternately, you can display a map in a specific projection by saving your table(s) in a specific projection (using the **Commit Table ... As** statement).

To control whether scroll bars appear on a Map window, issue a **Set Window** statement.

Using Animation Layers to Speed Up Map Redraws

If the **Add Map Layer** statement includes the **Animate** keyword, the 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, even if the map is very complex.

The animation layer is useful in realtime 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 and makes the table an animation layer:

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

Animation layers have the following restrictions:

- When you add an animation layer, it does not appear in the Layer Control dialog box.
- The user cannot interact with the animation layer by clicking in the Map window. For example, the user cannot use the Info tool to click on a point in the animation layer.
- Each Map window can have only one animation layer. The animation layer automatically becomes the map's top layer. If you attempt to add an animation layer to a Map window that already has an animation layer, the new animation layer replaces the old one.
- Workspace files do not preserve information about animation layers.
- To terminate the animation layer processing, issue a **Remove Map Layer Animate** statement.

Sample Program

To see a demonstration of animation layers, run the sample program ANIMATOR.MBX.

Performance Tips for Animation Layers

The purpose of the animation layer feature is to allow fast updates to small sections of the Map window. To get the best redraw speed possible:

- Avoid displaying the Map window in a Layout window. If the Map window that has the animation layer is displayed in a Layout window, screen updates may not be as fast.
- Make sure that the layer you are using as an animation layer is only displayed once in the Map window.

For example, suppose you are working with two tables: Roads (a table containing a street map), and Trucks (a table containing point objects, each of which represents a delivery truck). Suppose your Map window *already contains* both layers. If you want to turn the Trucks layer into an animation layer, you need to issue the following statement:

```
Add Map Layer Trucks Animate
```

However, you now have a problem: the Trucks layer now appears in the Map window twice—once as a conventional map layer, and once as an animation layer. Because the Trucks layer is still being displayed as a conventional layer, MapInfo Professional will not be able to perform fast screen updates. In other words, updates to the Map window will redraw as slowly as before, which defeats the purpose of the animation layer feature.

The following example demonstrates how to handle this situation. Before you add the Trucks layer as an animation layer, turn off the display of the “conventional” Trucks layer:

```
'temporarily prevent screen updates
  Set Event Processing Off

'set the original Trucks layer so it won't display
  Set Map Layer "Trucks" Display Off

'add the Trucks layer to the map, as an animation layer
  Add Map Layer Trucks Animate

' allow screen updates again
  Set Event Processing On

' At this point, there are two Trucks layers in the
' Map window. However, the "conventional" Trucks layer
' is not displayed, so it will not slow down the display
' of the "animated" Trucks layer.
```

Browser Windows

A Browser window displays columns of table data. The following statement opens a simple Browser window that displays all the columns in the World table:

```
Browse * From world
```

The asterisk specifies that every column in the table should appear in the Browser. To open a Browser window that displays only some of the columns, replace the asterisk with a list of column expressions. For example, the following statement opens a Browser window that shows only two columns:

```
Browse country, capital From world
```

The **Browse** statement can specify column expressions that calculate derived values. For example, the following statement uses the **Format\$()** function to create a formatted version of the World table's Population column. As a result, the second column in the Browser will contain commas to make the population statistics more readable.

```
Browse country, Format$(Population, ",#") From world
```

If the **Browse** statement specifies a simple column name (for example, country), the Browser window allows the user to edit the column values (unless the table is read-only). However, if the **Browse** statement specifies an expression that is more complex than just a column name, the corresponding column in the Browser window is read-only. Thus, if you want to create read-only columns in a Browser window, you can do so by browsing an expression, rather than a simple column name.

The expressions that you specify in the **Browse** statement appear as column headers across the top of the Browser window. The following statement shows how you can override the default column expression with an alias column header:

```
Browse country, Format$(Population, ",#") "Pop" From world
```

Because the String expression "Pop" appears after the column expression, "Pop" will be the column header that appears on the Browser window.

You can also set the initial default position of the Browser window. The following example positions the initial display so that the second column of the fifth row is in the upper left position of the Browser display:

```
Browse * From world Row 5 Column 2
```

Graph Windows

A Graph window contains a graph containing labels and values computed from a table. This sample displays a graph using one column for labels and another for data:

```
Graph country, population From world
```

The first item after the keyword **Graph** is the name of the column that provides labels for the data. Each following item is an expression that provides the graph with data. The example above is a simple expression in which the data is one column of the table. You can use any valid numeric expression.

Layout Windows

A Layout window represents a page layout. To open a Layout window, use the **Layout** statement.

Most Layout windows contain one or more frame objects. To create a frame object, issue a **Create Frame** statement. Layout windows also can contain any type of Map object. For example, to place a title on the page layout, create a text object by issuing a **Create Text** statement.

A Layout window can be treated as a table. For example, you can add objects to a Layout by issuing an **Insert** statement that refers to a table name such as "Layout1." However, strictly speaking, the objects that appear on a layout are not saved in table format (although they are saved in workspace files). For more information on accessing a Layout window as if it were a table, see [Chapter 8: Working With Tables](#).

Objects stored on Layout windows must use a Layout coordinate system, which defines object coordinates in terms of "paper" units such as inches or millimeters. For more information on Layout coordinates, see [Chapter 10: Graphical Objects](#).

Redistrict Windows

Use the **Create Redistrict** statement to begin a redistricting session. The **Create Redistrict** statement lets your program control all redistricting options that the user might configure through the Window > New Redistrict Window dialog.

Once a redistricting session has begun, you can control the Districts Browser by issuing **Set Redistrict** statements. To perform actions from the Redistrict menu, use the **Run Menu Command** statement.

For example, to assign objects to a district (as if the user had chosen Redistrict > Assign Selected Objects), issue the following statement:

```
Run Menu Command M_REDISTRICT_ASSIGN
```

To end a redistricting session, close the Districts Browser by issuing a **Close Window** statement. Note that values in the base table change as objects are re-assigned from district to district. After a redistricting session, you must save the base table if you want to retain the map objects' final district assignments. To save a table, issue a **Commit** statement.

For more information about redistricting, see the MapInfo Professional documentation.

Message Window

You can use MapBasic's **Print** statement to print text to the Message window. For example, the following statement prints a message to the Message window:

```
Print "Dispatcher is now on line."
```

Customizing the Info Window

The Info window displays a row from a table. The user can edit a row by typing into the Info window. To control and customize the Info window, use the **Set Window** statement. The following picture shows a customized Info window:



The following program creates the customized Info window shown above.

```
Include "mapbasic.def"
Open Table "World" Interactive

Select
  Country, Capital, Inflat_Rate + 0 "Inflation"
From World
Into World_Query

Set Window Info
  Title "Country Data"
  Table World_Query Rec 1
  Font MakeFont("Arial", 1, 10, BLACK, WHITE)
  Width 3 Units "in" Height 1.2 Units "in"
  Position (2.5, 1.5) Units "in"
  Front
```

Note the following points about this example:

- Ordinarily, the Info window's title bar reads "Info Tool." This program uses the **Title** clause to make the title bar read "Country Data."
- To specify which row of data appears in the window, use the **Set Window** statement's **Table ... Rec** clause. The example above displays record number 1 from the World_Query table. (World_Query is a temporary table produced by the **Select** statement.)
- The Info window displays a box for each field in the record; the scroll-bar at the right edge of the window allows the user to scroll down through the fields. To limit the number of fields displayed, the example above uses a **Select** statement to build a temporary query table, World_Query. The World_Query table has only three columns; as a result, the Info window displays only three fields.

To make some, but not all, of the fields in the Info window read-only:

1. Use a **Select** statement to produce a temporary query table.
2. Set up the **Select** statement so that it calculates expressions instead of simple column values. The **Select** statement shown above specifies the expression "Inflat_Rate + 0" for the third column value. (The "Inflation" string that follows the expression is an alias for the expression.)

```
Select
  Country, Capital, Inflat_Rate + 0 "Inflation"
```

3. In the **Set Window Info** statement, use the **Table... Rec** clause to specify which record is displayed. Specify a row from the query table, as in the example above. When a column in the query table is defined with an expression, the corresponding box in the Info window is read-only. (In the example above, the Inflation field is read-only.)
4. When the user types a new value into the Info window, MapInfo Professional automatically stores the new value in the temporary query table, *and* in the base table on which the query was based. You do not need to issue additional statements to apply the edit to the table. (However, you do need to issue a **Commit** statement if you want to save the user's edits.)

To make all fields in the Info window read-only, issue the following statement:

```
Set Window Info ReadOnly
```

Note: All of the fields in the Info window are read-only when you display a table that is a join (such as a StreetInfo table) or a query table that uses the Group By clause to calculate aggregate values.

ButtonPads (Toolbars)

A ButtonPad is a resizable, floating window which contains one or more buttons. The user can initiate various types of actions by choosing buttons from a ButtonPad.

The terms “ButtonPad” and “toolbar” mean exactly the same thing. The MapInfo Professional user interface refers to toolbars. For example, MapInfo Professional’s Options menu has a Toolbars command, which lets the MapInfo Professional user show or hide toolbars. Meanwhile, the MapBasic language syntax refers to toolbars as ButtonPads. For example, use the **Alter ButtonPad** statement to show or hide a toolbar.

MapInfo Professional provides several standard ButtonPads, such as the Main ButtonPad. A MapBasic program can add custom buttons to existing ButtonPads, or create entirely new ButtonPads.

What Happens When The User Chooses A Button?

Like menu items, custom buttons have handler procedures. When a user works with a custom button, MapBasic automatically calls that button’s handler procedure. Thus, if you want MapBasic to display a custom dialog each time the user clicks on a button, create a sub procedure which displays the dialog, and make that procedure the handler for the custom button.

A MapBasic program can create three different types of buttons: ToolButtons, ToggleButtons, and PushButtons. The button type dictates the conditions under which MapBasic calls that button’s handler.

- **PushButton:** When the user clicks on a PushButton, the button springs back up, and MapBasic calls the PushButton’s handler procedure.
The Layer Control button is an example of a PushButton. Clicking on the Layer Control button has an immediate effect (a dialog displays), but there is no lasting change to the status of the button.
- **ToggleButton:** When the user clicks on a ToggleButton, the button toggles between being checked (pushed in) and being unchecked (not pushed in). MapBasic calls the button’s handler procedure each time the user clicks on the ToggleButton.
The Show/Hide Legend Window button is an example of a ToggleButton. Clicking on the button has an immediate effect: showing or hiding the Legend Window. Furthermore, there is a lasting change to the button’s status: the button toggles in or out.
- **ToolButton:** When the user clicks on a ToolButton, that button becomes the active tool, and remains the active tool until the user chooses a different tool. MapBasic calls the button’s handler procedure if the user clicks in a Map, Browse, or Layout window while the custom button is the selected tool.
The Magnify tool is an example of a ToolButton. Choosing the tool does not produce any immediate effects; however, choosing the tool and then clicking in a Map window does have an effect.

MapBasic Statements Related To ButtonPads

The following statements and functions let you create and control custom buttons and ButtonPads:

Create ButtonPad

This statement creates a new ButtonPad.

Alter ButtonPad

After creating a custom ButtonPad, your program can alter various attributes of the ButtonPad. The **Alter ButtonPad** statement lets you reposition, show, or hide a ButtonPad, or add or remove buttons to or from a ButtonPad.

The **Alter ButtonPad** statement lets you modify any ButtonPad, even standard pads, such as Main. If your application needs only one or two custom buttons, you may want to add those buttons to the standard Main ButtonPad, instead of creating a new ButtonPad.

Alter Button

This statement modifies the status of a single button. Use the **Alter Button** statement to disable (de-activate) or enable (activate) a button, or to change which button is currently selected.

CommandInfo()

Use the **CommandInfo()** function within a button's handler procedure to query information about how the user has used the custom button. For example, if the user chooses a ToolButton and then clicks in a Map window, the **CommandInfo()** function can read the x- and y-coordinates of the location where the user clicked.

If you create two or more buttons that call the same handler procedure, that procedure can call **CommandInfo(CMD_INFO_TOOLBTN)** to determine which button is in use.

Thus, within a button's handler procedure, you might call **CommandInfo()** several times: Once to determine which button the user has chosen; once to determine the x-coordinate of the location where the user clicked; once to determine the y-coordinate; and once to determine whether or not the user held down the shift key while clicking.

ToolHandler

ToolHandler, a special procedure name, gives you an easy way to add one button to the Main ButtonPad. If your MapBasic program includes a procedure named ToolHandler, MapBasic automatically adds one ToolButton to the Main ButtonPad. Then, if the user chooses the ToolButton, MapBasic automatically calls the ToolHandler procedure each time the user clicks in a Map, Browse, or Layout window.

A MapBasic program cannot customize the button icon or draw mode associated with the ToolHandler procedure; the icon and cursor always use a simple + shape. If you need to specify a custom icon or cursor, use the **Create ButtonPad** or **Alter ButtonPad** statement instead of a ToolHandler procedure.

If the user runs multiple MapBasic applications at one time, and each application has its own ToolHandler, each application adds its own button to the Main ButtonPad.

Creating A Custom PushButton

The following program creates a custom ButtonPad containing a PushButton. The `button_prompt` procedure is the button's handler; therefore, whenever the user clicks the custom PushButton, MapBasic automatically calls the `button_prompt` procedure.

```
Include "icons.def"
Declare Sub Main
Declare Sub button_prompt

Sub Main
  Create ButtonPad "Custom" As
    PushButton
      Icon    MI_ICON_ZOOM_QUESTION
      Calling button_prompt
      HelpMsg "Displays the query dialog\nQuery"
    Show
End Sub

Sub button_prompt
  ' This procedure called automatically when
  ' the user chooses the button.
  ' ...
End Sub
```

The Main procedure contains only one statement: **Create ButtonPad**. This statement creates a custom ButtonPad, called "Custom," and creates one custom button on the ButtonPad.

The **PushButton** keyword tells MapBasic to make the custom button a PushButton.

The **Icon** clause tells MapBasic which icon to display on the custom button. The identifier, `MI_ICON_ZOOM_QUESTION`, is defined in the file `icons.def`. To see a list of standard MapInfo Professional icon identifiers, examine `icons.def`.

The **Calling** clause tells MapBasic to call the `button_prompt` procedure whenever the user clicks on the custom button.

The **HelpMsg** clause defines both a status bar help message and a ToolTip help message for the button. Help messages are discussed later in this chapter.

Adding A Button To The Main ButtonPad

The preceding example used the **Create ButtonPad** statement to create an all-new ButtonPad. MapBasic can also add custom buttons to MapInfo Professional's default ButtonPads, such as Main. To add a button to an existing ButtonPad, use the **Alter ButtonPad** statement, instead of the **Create ButtonPad** statement, as shown in the following example:

```
Alter ButtonPad "Main"
  Add Separator
  Add PushButton
    Icon    MI_ICON_ZOOM_QUESTION
    Calling button_prompt
    HelpMsg "Displays the query dialog\nQuery"
  Show
```

The **Add PushButton** clause adds a custom button to the Main ButtonPad, while the **Add Separator** clause places an empty space between the new button and the previous button. The **Add Separator** clause is optional; use it when you want to separate buttons into distinct groups.

MapInfo Professional includes a special ButtonPad, called Tools, so that MapBasic utility programs will have a place where they can add custom buttons. For example, the ScaleBar utility adds its button to the Tools ButtonPad.

Creating A Custom ToolButton

The preceding examples created custom PushButtons. MapBasic also can create custom ToolButtons, which act like MapInfo Professional tools, such as the Magnify and Line tools. If a program creates a custom ToolButton, the user can choose that tool, then use that tool to click, and sometimes drag, on a Map, Browse, or Layout window.

The following example creates a custom ToolButton. After selecting the tool, the user can click and drag in a Map window. As the user drags the mouse, MapInfo Professional displays a dynamically-changing line connecting the current cursor position to the location where the user clicked.

```
Include "icons.def"
Include "mapbasic.def"
Declare Sub Main
Declare Sub draw_via_button

Sub Main
  Create ButtonPad "Custom" As
    ToolButton
      Icon MI_ICON_LINE
      DrawMode DM_CUSTOM_LINE
      Cursor MI_CURSOR_CROSSHAIR
      Calling draw_via_button
      HelpMsg "Draws a line on a Map window\nDraw Line"
  Show
End Sub
Sub draw_via_button
  Dim x1, y1, x2, y2 As Float
  If WindowInfo(FrontWindow(), WIN_INFO_TYPE) <> WIN_MAPPER Then
    Note "This tool may only be used on a Map window. Sorry!"
    Exit Sub
  End If

  ' Determine map location where user clicked:
  x1 = CommandInfo(CMD_INFO_X)
  y1 = CommandInfo(CMD_INFO_Y)
  x2 = CommandInfo(CMD_INFO_X2)
  y2 = CommandInfo(CMD_INFO_Y2)

  ' Here, you could create objects based on x1, y1, x2, and y2.
End Sub
```

In this example, the **Create ButtonPad** statement includes the **ToolButton** keyword instead of the **PushButton** keyword. This tells MapBasic to make the custom button act like a drawing tool.

The button definition includes a **DrawMode** clause, which tells MapBasic whether the user can drag after clicking with the tool. The example above uses the DM_CUSTOM_LINE drawing mode; therefore, the user is able to click and drag with the custom tool, just as you can click and drag when using

MapInfo Professional's standard Line tool. When a tool uses the DM_CUSTOM_POINT mode, the user cannot drag after clicking. For a listing of all available drawing modes, see **Alter ButtonPad** in the MapBasic *Reference* or online Help.

The **DrawMode** also controls what the user sees while dragging. With the DM_CUSTOM_LINE mode, MapBasic draws a line between the cursor location and the point where the user first clicked. With the DM_CUSTOM_RECT mode, MapBasic draws a rectangular marquee while the user drags the mouse. Regardless of which **DrawMode** is used with a ToolButton, MapInfo Professional calls the button's handler procedure after the user clicks and releases the mouse button. The handler procedure can call **CommandInfo()** to determine where the user clicked. Note: If the user cancels the operation by pressing the Esc key, MapInfo Professional does not call the handler procedure.

Choosing Icons for Custom Buttons

When you define a custom button, you control the icon that appears on the button. To specify which icon you want to use, use the **Icon** clause.

The keyword **Icon** is followed by a code from ICONS.DEF. For example, the following statement defines a custom button that uses the icon for MapInfo Professional's Info button. The code MI_ICON_INFO is defined in ICONS.DEF.

```
Alter ButtonPad "Main"
  Add Separator
  Add PushButton
    Icon    MI_ICON_INFO
    Calling procedure_name
```

Note: MapInfo Professional 4.0 provides many built-in icons, most of which are not used in MapInfo Professional's standard user interface. To see a demonstration of the built-in icons, run the sample program Icon Sampler (ICONDEMO.MBX) and then choose an item from the Icon Sampler menu. To see the code for a particular icon, position the mouse over that icon.

The button's ToolTip shows you the icon code. You also can copy an icon's code to the clipboard:

1. Run the Icon Sampler application (ICONDEMO.MBX).
2. Choose an item from the Icon Sampler menu. A custom ButtonPad appears.



3. Click on the button whose icon you want to use. A dialog box appears.



4. Press Ctrl-C (the Windows shortcut for the Copy command).
5. Click OK to dismiss the dialog box.
6. Switch to MapBasic. Press Ctrl-V (the shortcut for Paste) to paste the code into your program.

Selecting Objects by Clicking With a ToolButton

If the user chooses a custom ToolButton and then clicks on a map object, the object is not selected; instead, MapInfo Professional calls the custom ToolButton's handler procedure. If you need to select the object on which the user clicked, issue a **Select** statement from within the handler procedure.

The following handler procedure selects the town boundary region where the user clicked. To determine the coordinates where the user clicked, call **CommandInfo()**. Then, to select objects at that location, issue a **Select** statement with a **Where** clause, and specify a geographic operator such as **Contains**. The following example selects all the town regions that contain the location where the user clicked.

```
Sub t_click_handle
  Dim fx, fy As Float

  fx = CommandInfo(CMD_INFO_X)
  fy = CommandInfo(CMD_INFO_Y)
  Select * From towns
    Where obj Contains CreatePoint(fx, fy)

End Sub
```

Note: Instead of using a **Select** statement, you could call the **SearchPoint()** or **SearchRect()** function to perform a search, and then call **SearchInfo()** to process the search results. For an example of this technique, see **SearchInfo()** in the *MapBasic Reference* or online Help.

Another approach would be to define a procedure called **SelChangedHandler**. If the user is running an application that contains a **SelChangedHandler** procedure, MapInfo Professional automatically calls that procedure every time the selection changes. The user could select objects by pointing and clicking with MapInfo Professional's standard Select tool (the arrow-shaped icon at the upper left corner of MapInfo Professional's Main ButtonPad), and your application could respond by issuing statements within the **SelChangedHandler** procedure.

Including Standard Buttons in Custom ButtonPads

You can include any of MapInfo Professional's standard buttons (such as the Select button) on custom ButtonPads. For example, the following statement creates a custom ButtonPad containing two buttons: The standard MapInfo Professional Select button, and a custom button.

```
Create ButtonPad "ToolBox" As
  ' Here is the standard Select button...
  ToolButton
    Icon    MI_ICON_ARROW
    Calling M_TOOLS_SELECTOR
    HelpMsg "Select objects for editing\nSelect"

  ' Here is a custom ToolButton...
  ToolButton
    Icon    MI_ICON_LINE
    DrawMode DM_CUSTOM_LINE
    Calling sub_procedure_name
    HelpMsg "Draw New Delivery Route\nNew Route"
```

The first button's **Calling** clause specifies `M_TOOLS_SELECTOR`, which is a numeric code defined in `MENU.DEF`. This code represents MapInfo Professional's Select button. Every standard MapInfo Professional button has a corresponding code in `MENU.DEF`. Because the second button is a custom button, its **Calling** clause specifies the name of a procedure, rather than a numeric code.

Note that the custom button includes a **DrawMode** clause, but the Select button does not. When you place a standard button on a custom pad, you should omit the **DrawMode** clause, because each of MapInfo Professional's standard buttons already has a pre-defined draw mode. You should only specify a **DrawMode** clause when creating a custom `ToolButton`.

CAUTION: **Caution: ToolButtons and ToggleButtons are not interchangeable. You cannot convert one type of button to another type merely by replacing the ToolButton keyword with the ToggleButton keyword (or vice versa). ToolButtons return x/y coordinates in response to the user clicking on a window. ToggleButtons, however, do not return coordinates, and they respond as soon as the user clicks on the button.**

If you include standard MapInfo Professional buttons in your custom `ButtonPads`, make sure that you do not accidentally change a `ToolButton` to a `ToggleButton`. To see how MapInfo Professional's standard buttons are defined, view the MapInfo Professional menus file, `MAPINFOW.MNU`. (On the Macintosh, this file is called MapInfo Professional Menus.) The menus file contains the **Create ButtonPad** statements that define MapInfo Professional's `ButtonPads`.

Note: You can copy button definitions out of `MAPINFOW.MNU` and paste them into your programs.

Assigning Help Messages to Buttons

Your users may not understand the purpose of a toolbar button just by looking at its icon. Therefore, MapBasic lets you create two types of on-screen help messages to assist your users:

- Status bar help. Used to show a brief description of the button, this type of help message appears on the MapInfo Professional status bar (assuming that the status bar is currently visible).
- ToolTip help. Used to show the name of the button, this type of help message appears next to the mouse cursor.

In earlier versions of MapInfo Professional, status bar help only appeared when the user clicked on a button. In version 4.0 and later, both the status bar help and ToolTip help appear when the user leaves the mouse cursor positioned over a toolbar button.

Both types of help messages are defined through the **HelpMsg** clause, in the **Create ButtonPad** and **Alter ButtonPad** statements. Within the **HelpMsg** clause, you specify one string that contains the status bar help message, followed by the letters `\n`, followed by the ToolTip message.

For example:

```
Create ButtonPad "Custom" As
  PushButton
    Icon    MI_ICON_ZOOM_QUESTION
    Calling generate_report
    HelpMsg "This button generates reports\nGenerate Report"
  Show
```

In this example, the custom button's status bar help message is "This button generates reports" and its ToolTip message is "Generate Report." To show or hide the status bar, use the **StatusBar** statement.

Docking a ButtonPad to the Top of the Screen

Use the **Alter ButtonPad** statement to attach a toolbar to the top edge of the screen. (This is sometimes known as “docking” the toolbar.) For example, the following statement docks the Main toolbar:

```
Alter ButtonPad "Main" Fixed
```

The keyword **Fixed** specifies that the pad should be docked to the top of the screen. To change a toolbar from docked to floating, specify **Float** instead of **Fixed**. The **Fixed** and **Float** keywords can also be used within the **Create ButtonPad** statement, so that you can set the docked status at the moment you create the toolbar.

To determine whether a toolbar is currently docked, call the **ButtonPadInfo()** function.

Other Features of ButtonPads

MapBasic also offers the following ButtonPad-related features:

- **Enabled/Disabled Buttons.** A MapBasic program can disable or enable custom buttons as needed. For details, see the MapBasic *Reference*, **Alter ButtonPad**.
- **Custom Button Icons.** You can use a resource editor to create custom icons, and use those custom icons on MapBasic ButtonPads.

Custom Draw Cursors. The cursor is the shape that moves as you move the mouse. By default, all custom MapBasic buttons use a simple cursor, shaped like a pointer. However, you can use a resource editor to create custom cursors.

The MapBasic development environment does not include a resource editor. However, MapBasic programs can incorporate bitmaps and cursors created using other resource editors. For more information about creating custom icons and cursors, see **Chapter 12: Integrated Mapping**.

Integrating Your Application Into MapInfo Professional

The preceding sections have discussed how a MapBasic application can customize the user interface by creating custom menus, dialogs, windows and ButtonPads. Once you have completed your application, however, one issue will remain: what steps does the user have to take to run your application, so that your customized user-interface will take effect?

Any MapInfo Professional user can run a MapBasic application by choosing Tools > Run MapBasic Program. However, you may want to set up your application so that it runs automatically, instead of forcing your users to choose File > Run MapBasic Program every time they run MapInfo Professional. If you are creating what is known as a turn-key system, you probably want your application to run automatically, as soon as the user launches MapInfo Professional.

Using Windows you can change the command line of a shortcut icon in a similar manner. Right-click the shortcut icon, choose Properties, and click on the Shortcut tab.

Ordinarily, MapInfo Professional displays the Quick Start dialog as soon as the user runs it (unless the user has cleared the Display Quick Start Dialog check box in the Startup Preferences dialog). However, if you add the name of a MapBasic application to the command that launches MapInfo Professional, then the Quick Start dialog will not appear. Depending on the nature of your application, this behavior may or may not be desirable. If you want your application to run automatically, without disabling the Quick Start dialog, you may need to use a different method for loading your application. Instead of modifying the MapInfo Professional command line, you may want to create a special workspace, called the Startup workspace.

If the user launches MapInfo Professional by double-clicking the MapInfo icon, the Quick Start dialog box displays automatically (unless the user has cleared the Display Quick Start Dialog check box in the Startup Preferences dialog). However, when the user launches MapInfo Professional by double-clicking on a MapInfo document, the Quick Start dialog does not appear. Depending on the nature of your application, this behavior may or may not be desirable. If you want your application to run automatically, without disabling the Quick Start dialog, you may need to use a different method for loading your application. You may want to create a special workspace, called the Startup workspace.

Loading Applications Through the Startup Workspace

“Startup” is a special name for a workspace. If a startup workspace exists on the user’s system, MapInfo Professional loads the workspace automatically. If the startup workspace contains a **Run Application** statement, MapInfo Professional runs the specified application.

For example, if you want to run the ScaleBar application, you could create a startup workspace that looks like this:

```
!Workspace
!Version 600
!Charset Neutral
Run Application "scalebar.mbx"
```

The first three lines are required for MapInfo Professional to recognize the file as a workspace. The fourth line, in this example, launches a MapBasic application by executing a **Run Application** statement.

The presence of a startup workspace has no effect on the display of the Quick Start dialog. MapInfo Professional loads the startup workspace (if there is one), and then displays the Quick Start dialog (unless the user has configured the system so that the Quick Start dialog never displays).

On Windows, the startup workspace has the name STARTUP.WOR and can be located in the directory in which MapInfo Professional is installed or in the user’s private Windows directory (the directory where WIN.INI is stored). If a STARTUP.WOR exists in both directories, both workspaces will be executed when the user starts MapInfo Professional.

In a networked environment, if you want the startup workspace to apply to all MapInfo Professional users on the network, you should place the startup workspace file in the directory where MapInfo Professional is installed. If you do not want all the network users to run the same startup workspace file, you should use the alternate location for the startup workspace (for example, on Windows, place the workspace in the users’ private Windows directories).

Manipulating Workspaces through MapBasic

Since workspaces are simply text files, you can create and edit a startup workspace using any text editor. Furthermore, since a MapBasic program can perform file input/output, your MapBasic program can automate the maintenance of the startup workspace.

To see how a MapBasic program can manipulate a workspace file, try this:

1. Choose MapInfo Professional's Tools > Run MapBasic Program command to run the TextBox application.
2. Choose Tools > TextBox > About TextBox to display the About TextBox dialog.
3. Click on the Auto-Load button on the About TextBox dialog. MapInfo Professional displays a dialog that lets you activate automatic loading of the TextBox application.
4. Choose OK on the Enable Automatic Loading dialog. MapInfo Professional displays a message indicating that the TextBox application is now configured to run automatically. Choose OK on the About TextBox dialog.
5. Exit MapInfo Professional, then restart it. Note that in this new MapInfo Professional session, the TextBox application runs automatically; you do not need to choose Tools > Run MapBasic application.

When you choose OK in step 4, the TextBox application adds a **Run Application** statement to the startup workspace file. If the startup workspace file does not exist, the TextBox application creates it.

The maintenance of the startup workspace is handled by functions and procedures in the program module `auto_lib.mb`. Many of the sample programs that are bundled with MapInfo Professional contain the same functionality; for example, a MapInfo Professional user can set up the ScaleBar application to run automatically by choosing the Auto-Load button on the About ScaleBar dialog.

The `auto_lib.mb` program module is one of the sample programs included with MapBasic. If you want your application to include the Auto-Load feature, follow the instructions that appear in the comments at the top of `auto_lib.mb`.

Performance Tips for the User Interface

Animation Layers

If you are making frequent updates to objects in a Map window, using an Animation Layer can make the window redraw more quickly. Animation Layers are described earlier in this chapter.

Avoiding Unnecessary Window Redraws

Whenever your application alters a Map window (or alters an object in the window), MapInfo Professional redraws the window. If your application makes several alterations, the Map window will redraw several times, which can annoy your users.

There are two ways to suppress unnecessary window redraws:

- To suppress unnecessary redrawing of one Map window, use the **Set Map ... Redraw Off** statement. Then issue all statements that affect the Map window. When you are finished updating the map, issue a **Set Map ... Redraw On** statement to allow the window to redraw. The window will redraw once, showing all changes you made.
- To suppress unnecessary redrawing of all MapInfo Professional windows, use the **Set Event Processing Off** statement. When you are finished updating various windows, issue a **Set Event Processing On** statement, and the screen will redraw once.

Purging the Message Window

The **Print** statement prints text to the Message window.

Note: Printing large amounts of text to the Message window can dramatically slow down subsequent **Print** statements.

If your program prints large amounts of text to the message window, you should periodically clear the Message window by issuing a **Print Chr\$(12)** statement.

Suppressing Progress Bar Dialogs

If your application minimizes MapInfo Professional, you should suppress progress bars by using the **Set ProgressBars Off** statement.

When a progress bar displays while MapInfo Professional is minimized, the progress bar is frozen for as long as it is minimized. If you suppress the display of progress bars, the operation can proceed, even if MapInfo Professional is minimized.

Working With Tables

MapBasic provides you with a full complement of statements and functions for working with tables. For instance, you can modify the structure of a table using the **Alter Table** statement, or locate a row in a table using **Fetch**. The **Import** statement lets you create a MapInfo table from a text file and the **Export** statement lets you export a table to a different format.

This chapter introduces you to the MapBasic statements and functions that let you manage your MapInfo tables. Refer to the MapBasic *Reference* for more information about each statement and function.

Sections in this Chapter:

- ♦ Opening Tables Through MapBasic 150
- ♦ Creating New Tables. 157
- ♦ Accessing the Cosmetic Layer 162
- ♦ Accessing Layout Windows 162
- ♦ Multi-User Editing. 163
- ♦ Files that Make Up a Table. 166
- ♦ Raster Image Tables. 167
- ♦ Working With Metadata 169
- ♦ Working With Seamless Tables. 171
- ♦ Accessing DBMS Data 173
- ♦ Accessing/Updating Remote Databases with Linked Tables .
175
- ♦ Performance Tips for Table Manipulation 176

Opening Tables Through MapBasic

A table must be open before a MapBasic application can access the table. Use the **Open Table** statement to open a table. For example, the following statement opens the World table:

```
Open Table "C:\mapinfo\data\world"
```

Notice that the **Browse** statement identifies the table by its alias (Earth). The table's alias name remains in effect for as long as the table is open. The table has not been permanently renamed. To permanently rename a table, use the **Rename Table** statement.

If you include the optional **Interactive** clause in the **Open Table** statement, and if the table you specify cannot be located in the directory that you specify, MapInfo displays a dialog prompting the user to locate the table. If you omit the **Interactive** keyword and the table cannot be located, the **Open Table** statement generates an error.

Determining Table Names at Runtime

When referring to a table in MapBasic, you can either use a string expression or hard-code the table name into your program. For example, if the tables States, Pipeline and Parcels are open when your program is run, you can specify their names explicitly in your program:

```
Select * From States  
Browse * From Pipeline  
i = NumCols(Parcels)
```

You may or may not want to limit your program to work with specific table names. For example, you might want to prompt the user to choose a table from a list of open tables. Since you wouldn't know the name of the selected table ahead of time, you couldn't hard-code it into the program.

You can use a string variable to store the name of a table. Assuming that a table called Zoning is open, you can do the following:

```
Dim work_table As String  
work_table = "Zoning"  
Browse * From work_table
```

Opening Two Tables With The Same Name

MapInfo assigns a non-default table alias if you attempt to open two tables that have the same alias. For example, if you open the table "C:\data1994\sites", MapInfo assigns the table its default alias ("sites"); but if you then attempt to open a different table that has an identical default alias (for example, "C:\backup\sites"), MapInfo must assign a non-default alias to the second table, so that the two tables can be differentiated. In this example, MapInfo might assign the second table an alias such as "sites_2."

If you include the optional **Interactive** keyword in the **Open Table** statement, MapInfo will display a dialog to let the user specify the table's non-default alias. If you omit the **Interactive** keyword, MapInfo assigns the alias table name automatically.

As a result of this behavior, you may not be able to make assumptions about the alias name with which a table was opened.

However, you can use the **TableInfo()** function to determine the alias under which a table was opened, as shown in the following example:

```
Include "mapbasic.def"
Dim s_filename As String
Open Table "states" Interactive
s_filename = TableInfo(0, TAB_INFO_NAME)
Browse * from s_filename
```

The function call **TableInfo(0, TAB_INFO_NAME)** returns the alias name of the most recently opened table.

Opening Non-Native Files As Tables

You can access “non-native” files (dBASE, Lotus, Excel, or text files) as tables, even though they are not stored in the MapInfo table format. However, before you access a non-native file through MapBasic, you must register the file. When you register a file, MapInfo builds a table (.tab) file to accompany the non-native file. You only need to register each file once. After you have registered a file, you can treat the file as a table.

The following statement registers a dBASE file:

```
Register Table "income.dbf" Type DBF
```

After you have registered a file, the file is considered a table, and you can open it the same way you would open any MapInfo table: by issuing an **Open Table** statement.

```
Open Table "income" Interactive
```

MapInfo’s ability to query a table is not affected by the table’s source. For example, you can issue a SQL Select statement to extract data from a table, regardless of whether the table was based on a spreadsheet or a database file.

However, MapInfo’s ability to modify a table does depend in part on the table’s source. If a table is based on a .dbf file, MapInfo can modify the table; when you **Update** such a table in MapInfo, you are actually modifying the original .dbf file. However, MapInfo cannot modify tables that are based on spreadsheets or ASCII (text) files. If you need to modify a table, but MapInfo cannot modify the table because it is based on a spreadsheet or ASCII file, make a copy of the table (using the **Commit Table ... As** statement) and modify the copy.

Creating A Report File From An Open MapInfo Table

High quality reports of tabular data, processed within MapInfo, can be produced using the industry standard report writer, from Seagate Crystal Reports. Crystal provides a highly intuitive environment for developing professional reports. See the **Create Report From Table** and **Open Report** statements in the MapBasic *Reference*.

Reading Row-And-Column Values From a Table

MapBasic programs can access specific column values from specific rows in a table, through the following procedure:

1. Use a **Fetch** statement to specify which row in the table you want to query. This action sets which row is current.
2. Use a table-reference expression (for example, *tablename.columnname*) to access a specific column in the current row.

For example, the following program reads the contents of the Country column from the first row of the World table:

```
Dim s_name As String
Open Table "world" Interactive
Fetch First From world
s_name = world.Country
```

Every open table has a current-row setting; this setting is known as the *row cursor* (not to be confused with the *mouse cursor*, which is the shape that moves across the screen as you move the mouse). When you issue a **Fetch** statement, you position the row cursor on a specific row in the table. Subsequent table references (for example, *world.country*) extract data from whichever row is specified by the cursor.

The **Fetch** statement provides several different ways of positioning the cursor. You can move the cursor forward or backward one row at a time, position the cursor on a specific row number, or set the cursor on the first or last row in the table. To determine whether a **Fetch** statement has attempted to read past the end of a table, call the **EOT()** function. For more information on the **Fetch** statement or the **EOT()** function, see the *MapBasic Reference*.

The MapBasic language recognizes three different types of expressions that reference specific column values:

Column reference syntax	Example
<i>tablename.columnname</i>	<i>world.country</i>
<i>tablename.COLn</i>	<i>world.COL1</i>
<i>tablename.COL(n)</i>	<i>world.COL(i)</i>

The preceding example used the *tablename.columnname* syntax (for example, *world.country*).

Another type of column reference is *tablename.col#*. In this type of expression, a column is specified by number, not by name (where **col1** represents the first column in the table). Since Country is the first column in the World table, the assignment statement above could be rewritten as follows:

```
s_name = world.col1
```


A third type of column reference takes the form *tablename.col(numeric expression)*. In this type of reference, the column number is specified as a numeric expression within parentheses. The preceding assignment statement could be rewritten as follows:

```
Dim i As Integer
i = 1
s_name = world.col(i)
```

Using this syntax, you can write a MapBasic program that determines, at runtime, which column to reference.

The *tablename* in a table reference is optional in statements in which the table name is already part of the statement. For instance, in the **Browse** statement you are required to specify column names and then the table name. Since the table name is explicitly specified in the statement (in the **From** clause), the column references at the beginning of the line do not need to include the *tablename*.

```
Select Country, Population/1000000 From World
Browse Country, Col2 From Selection
```

The **Select** statement also has a **From** clause, where you name the table(s) to be queried. Column names that appear within a **Select** statement do not need the *tablename*. prefix if the **Select** statement queries a single table. However, if a **Select** statement's **From** clause lists two or more tables, column references must include the *tablename*. prefix. For a general introduction to using the SQL **Select** statement, see the *MapInfo User Guide*, or see **Select** in the *MapBasic Reference*.

There are instances in which you *must* use the COLn or the COL(n) column referencing method. In the example above, the **Select** statement identifies two columns; the latter of these columns is known as a *derived* column, since its values are derived from an equation (Population/1000000). The subsequent **Browse** statement can refer to the derived column only as col2 or as col(2), because the derived expression Population/1000000 is not a valid column name.

Alias Data Types as Column References

The preceding examples have used explicit, “hard-coded” column names. For example, the following statement identifies the Country column and the Population column explicitly:

```
Select Country, Population/1000000 From World
```

In some cases, column references cannot be specified explicitly, because your application will not know the name of the column to query until runtime. For example, if your application lets the user choose a column from a list of column names, your application will not know until runtime what column the user chose.

MapBasic provides a variable type, Alias, that you can use to store column expressions that will be evaluated at runtime. As with String variables, you can assign a text string to an Alias variable. MapBasic interprets the contents of the Alias variable as a column name whenever an Alias variable appears in a column-related statement. For example:

```
Dim val_col As Alias
val_col = "Inflat_Rate"
Select * From world Where val_col > 4
```

MapBasic substitutes the contents of val_col (the alias, Inflat_Rate) into the **Select** statement in order to select all the countries having an inflation rate greater than 4 percent.

Note: The maximum length of the alias is 32 characters.

In the example below, the sub-procedure **MapIt** opens a table, maps it, and selects all records from a specified column that have a value greater than or equal to a certain value. MapIt uses an Alias variable to construct column references that will be evaluated at runtime.

```

Include "mapbasic.def"
Declare Sub Main
Declare Sub MapIt( ByVal filespec As String,
                  ByVal col_name As String,
                  ByVal min_value As Float )

Sub Main
    Call MapIt("C:\MAPINFO\MAPS\WORLD.TAB", "population", 15000000)
End Sub
Sub MapIt(      ByVal filespec As String,
              ByVal col_name As String,
              ByVal min_value As Float )

    Dim a_name As Alias
    a_name = col_name
    Open Table filespec
    Map From TableInfo(0, TAB_INFO_NAME)
    Select * From TableInfo(0, TAB_INFO_NAME)
        Where a_name >= min_value
End Sub

```

In the MapIt procedure, a **Select** statement specifies an Alias variable (a_name) instead of an explicit column name. Note that the col_name parameter is not an Alias parameter; this is because MapBasic does not allow by-value Alias parameters. To work around this limitation, the column name is passed as a by-value String parameter, and the contents of the String parameter are copied to a local Alias variable (a_name).

The example above demonstrates how an Alias variable can contain a string representing a column name ("population"). An Alias variable also can contain a full column reference in the form *tablename.columnname*. The following example demonstrates the appropriate syntax:

```

Dim tab_expr As Alias
Open Table "world"
Fetch First From world
tab_expr = "world.COL1"
Note tab_expr

```

The preceding **Note** statement has the same effect as the following statement:

```
Note world.COL1
```

Scope

The syntax *tablename.columnname* (for example, world.population) is similar to the syntax used to reference an element of a custom **Type**. MapBasic tries to interpret any *name.name* expression as a reference to an element of a **Type** variable. If the expression cannot be interpreted as a type element, MapBasic tries to interpret the expression as a reference to a column in an open table. If this fails, MapBasic generates a runtime error.

Using the “RowID” Column Name To Refer To Row Numbers

RowID is a special column name that represents the row numbers of rows in the table. You can treat RowID as a column, although it isn't actually stored in the table. Think of RowID as a *virtual column*, available for use, but not visible. The first row of a table has a RowID value of one, the second row has a RowID value of two, and so on.

The following example selects the first row from the World table:

```
Select * from world Where RowID = 1
```

The following example uses **RowID** to **Select** all of the states with a 1990 population greater than the median.

```
Dim median_row As Integer
Select * From states Order By pop_1990 Into bypop
median_row = Int(TableInfo(bypop, TAB_INFO_NROWS)/2)
Select * From bypop Where RowID > median_row
```

Since the **TableInfo()** function returns the total number of rows in the virtual table bypop, the variable median_row contains the record number of the state with the median population. The last select statement selects all the states that come after the median in the ordered table bypop.

If you delete a row from a table, the row is not physically deleted until you perform a pack operation. (Rows that have been deleted appear grayed in a Browse window.) Any deleted row still has a RowID value. Thus, deleting a row from a table does not affect the RowID values in the table; however, if you delete a row, save your changes, and then pack the table, the table's RowID values do change. To pack a table, choose MapInfo's Table > Maintenance > Pack Table command, or issue the MapBasic statement **Pack Table**.

Using the “Obj” Column Name To Refer To Graphic Objects

The **Obj** column is a special column name that refers to a table's graphical objects. Any table that has graphical objects has an **Obj** column (although the Obj column does not appear in any Browser window). If a row does not have an associated graphic object, that row has an empty **Obj** value.

The following example selects all rows that do not have a graphic object:

```
Select * From sites Where Not Obj
```

This is useful, for instance, in situations where you have geocoded a table and not all of the records matched, and you want to select all of the records that did not match.

The following example copies a graphical object from a table into an Object variable:

```
Dim o_var As Object
Fetch First From sites
o_var = sites.obj
```

For more information about graphical objects, see [Chapter 10: Graphical Objects](#).

Finding Map Addresses In Tables

MapInfo users can find addresses in maps by choosing Query > Find. MapBasic programs can perform similar queries by issuing **Find** statements and **Find Using** statements. The **Find Using** statement specifies the table to be queried; the **Find** statement tries to determine the geographic coordinates of a location name (for example, “23 Main St”). The **Find** statement also can locate the intersection of two streets, given a string that includes a double-ampersand (for example, “Pawling Ave && Spring Ave”).

After issuing a **Find** statement, call **CommandInfo()** to determine whether the address was located, and call **CommandInfo()** again to determine the location’s geographic coordinates. Unlike MapInfo’s Query > Find command, the MapBasic **Find** statement does not automatically re-center a Map window. If you want to re-center the Map window to show the location, issue a **Set Map** statement with a **Center** clause. Also, the **Find** statement does not automatically add a symbol to the map to mark where the address was found. If you want to add a symbol, use the **CreatePoint()** function or the **Create Point** statement. For a code example, see **Find** in the MapBasic *Reference* or online Help.

Geocoding

To perform automatic geocoding:

1. Use the **Fetch** statement to retrieve an address from a table.
2. Use the **Find Using** statement and the **Find** statement to find the address.
3. Call **CommandInfo()** to determine how successful the **Find** statement was; call **CommandInfo()** again to determine x- and y-coordinates of the found location.
4. Create a point object by calling the **CreatePoint()** function or the **Create Point** statement.
5. Use the **Update** statement to attach the point object to the table.

To perform interactive geocoding, issue the following statement:

```
Run Menu Command M_TABLE_GEOCODE
```

If you need to perform high-volume geocoding, you may want to purchase MapMarker, a dedicated geocoding product that is sold separately. MapMarker geocodes faster than MapInfo and allows single-pass geocoding across the entire United States. MapBasic applications can control MapMarker through its programming interface. For more information on MapMarker, contact MapInfo sales. The phone numbers appear at the start of this and other MapInfo manuals.

Performing SQL Select Queries

MapInfo users can perform sophisticated queries by using MapInfo’s Query > SQL Select dialog. All of the power of the SQL Select dialog is available to MapBasic programmers through MapBasic’s **Select** statement. You can use the **Select** statement to filter, sort, sub-total, or perform relational joins on your tables. For information, see **Select** in the MapBasic *Reference*.

Error Checking for Table and Column References

MapBasic cannot resolve references to tables and columns at compile time. For instance, if your program references a column called states.pop, the MapBasic compiler cannot verify whether the states table actually has a column called pop. This means that typographical errors in column references will not generate errors at compile time. However, if a column reference (such as states.pop) contains a typographical error, an error will occur when you run the program.

Try the following to minimize the possibility of generating runtime errors. Use the **Interactive** clause with the **Open Table** statement, when appropriate. If the table cannot be located, a dialog will prompt the user to locate the table. Don't assume that the table was opened under its default alias. After you issue an **Open Table** statement, call **TableInfo(0, TAB_INFO_NAME)** to determine the alias assigned to the table. For more information on opening tables, see **Open Table** in the *MapBasic Reference*.

Writing Row-And-Column Values to a Table

To add new rows to a table, use the **Insert** statement. To change the values stored in the columns of existing rows, use the **Update** statement. Both statements are described in the *MapBasic Reference* and online Help.

If you add new rows to a table or modify the existing rows in a table, you must save your changes by issuing a **Commit** statement. Alternately, to discard any unsaved edits, issue a **RollBack** statement.

Creating New Tables

Use the **Create Table** statement to create a new, empty table. Use the **Create Index** statement to add indexes to the table, and use **Create Map** to make the table mappable.

The following example creates a mappable table with a name, address, city, amount, order date, and customer ID columns. The name field and the customer ID field are indexed.

```
Create Table CUST
(Name Char(20),
 Address Char(30),
 City Char(30),
 Amount Decimal(5,2),
 OrderDate Date,
 CustID Integer)
File "C:\customer\Cust.tab"
Create Map For CUST CoordSys Earth

Create Index On CUST (CustID)

Create Index On CUST(Name)
```

You can also create a table by saving an existing table (for example, a selection) as a new table using the **Commit** statement, or by importing a table using the **Import** statement.

Modifying a Table's Structure

Every table has a structure. The structure refers to issues such as how many columns are in the table, and which of the columns are indexed. A MapInfo user can alter a table's structure by choosing MapInfo's Table > Maintenance > Table Structure command. A MapBasic program can alter a table's structure by issuing statements such as **Alter Table** and **Create Index**.

As a rule, a table's structure cannot be modified while the table has unsaved edits. If you have added rows to a table, but you have not saved the table, the table has unsaved edits. If a table has unsaved edits, you must save the edits (by issuing a **Commit** statement) or discard the edits (by issuing a **Rollback** statement) before modifying the table's structure.

The **Alter Table** statement modifies a table's structure. The following example renames the Address column to ShipAddress, lengthens the Name column to 25 characters, removes the Amount column, adds new ZIP Code and Discount columns, and re-orders the columns.

```
Alter Table CUST (Rename Address ShipAddress,  
  Modify Name Char(25),  
  Drop Amount  
  Add Zipcode Char(10),  
    Discount Decimal(4,2)  
  Order Name, Address, City, Zipcode,  
    OrderDate, CustID, Discount)
```

You cannot change the structure of tables that are based on spreadsheets or delimited ASCII files, and you cannot change the structure of the Selection table.

Use the **Add Column** statement to add a temporary column to a table. The **Add Column** statement lets you create a dynamic column that is computed from values in another table. **Add Column** can also perform advanced polygon-overlay operations that perform proportional data aggregation, based on the way one table's objects overlap another table's objects. For example, suppose you have one table of town boundaries and another table that represents a region at risk of flooding. Some towns fall partly or entirely within the flood-risk area, while other towns are outside the risk area. The **Add Column** statement can extract demographic information from the town-boundaries table, then use that information to calculate statistics within the flood-risk area. For information about the **Add Column** statement, see the MapBasic *Reference*.

Creating Indexes and Making Tables Mappable

Table indexes help MapInfo to optimize queries. Some operations, like MapInfo's **Find** and **Geocode** menu items, require an index to the field to be matched against. For instance, before you can use the **Find** command to locate a customer in your database by name, you must index the name column. Select statements execute faster for many queries when you use columns with indexes. SQL joins create a temporary index if the fields specified in the **Where** clause are not indexed. There is no limit to the number of columns that can be indexed. The **Obj** column is always indexed.

To create an index in MapBasic, use the **Create Index** statement. To remove an index, use the **Drop Index** statement.

MapBasic cannot use indexes created in other packages and MapBasic cannot index on an expression.

An index does not change the order of rows in a Browser window. To control the order of rows in a Browser, issue a **Select** statement with an **Order By** clause, and browse the selection.

Reading A Table's Structural Information

The functions **TableInfo()**, **ColumnInfo()** and **NumTables()** let you determine information about the tables that are currently open.

- **TableInfo()** returns the number of rows in the table, the number of columns, and whether or not the table is mappable.
- **ColumnInfo()** returns information about a column in a table, such as the column's name, the column's data type, and whether the column is indexed.
- **NumTables()** returns the number of currently open tables (including temporary tables such as Query1).

The following program determines which tables are open and copies the table names into an array.

```
Include "mapbasic.def"
Dim i, table_count As Integer
Dim tablenames() As String

' determine the number of open tables
table_count = NumTables()

' Resize the array so that it can hold
' all of the table names.
ReDim tablenames(table_count)

' Loop through the tables
For i = 1 To table_count

    ' read the name of table # i
    tablenames(i) = TableInfo(i, TAB_INFO_NAME)

    'display the table name in the message window
    Print tablenames(i)

Next
```

Working With The Selection Table

Selection is a special table name that represents the set of rows that are currently selected. A MapBasic program (or an end-user) can treat the Selection table like any other table.

For example, you can browse the set of currently-selected rows by issuing the following statement:

```
Browse * From Selection
```

When you access the Selection table in this way, MapInfo takes a snapshot of the table and names the snapshot **QueryN**, where *N* is a integer value of one or greater. Like Selection, QueryN is a temporary table. The **SelectionInfo()** function lets you determine the table alias MapInfo will assign to the current Selection table (i.e., to learn whether the current Selection table will be known as Query1 or as Query2). **SelectionInfo()** also lets you determine other information about the Selection, such as the number of selected rows.

Cleaning Up “QueryN” Tables

As you use MapInfo, you may find that you have opened a number of “QueryN” tables (Query1, Query2, etc.). For example, if you click on a map object and then browse the selection, the window’s title may read “Query1 Browser.” Each QueryN is a snapshot of a former selection.

MapBasic programs can cause QueryN tables to be opened as well. For example, making a reference to a column expression such as **Selection.Obj** causes MapInfo to open a QueryN table. If you want your MapBasic program to close any QueryN tables that it opens, do the following:

- When you use **Select** statements, include the optional **Into** clause. Then, instead of accessing the table name “Selection” access the table name that you specified in the **Into** clause. If you use the **Into** clause, MapInfo will not open QueryN tables when you access the query results. When you are done working with the query results table, close it by using a **Close Table** statement.
- If the user makes a selection (for example, by clicking on a map object), and then your program works with the selection, MapInfo will open a QueryN table. The following example shows how to close the QueryN table.

```
' Note how many tables are currently open.
i_open = NumTables()

' Access the Selection table as necessary. For example:
Fetch First From Selection
obj_copy = Selection.obj

' If we just generated a QueryN table, close it now.
If NumTables() > i_open Then
    Close Table TableInfo(0, TAB_INFO_NAME)
End If
```

Changing the Selection

Use the **Select** statement to change which rows are selected. The **Select** statement is a very powerful, versatile statement. You can use the **Select** statement to filter, sort, or sub-total your data, or to establish a relational join between two or more tables. All of the power of MapInfo’s Query > SQL Select command is available to MapBasic programmers through the **Select** statement.

If you issue a **Select** statement, and if you do not want the results table to have a name such as Query1, you can assign another name to the results table. The **Select** statement has an optional **Into** clause that lets you specify the name of the results table. For example, the following statement makes a selection and names the results table “Active.”

```
Select * From sites
Where growth > 15
Into Active
```

For an introduction to the capabilities of SQL Select queries, see the MapInfo *User Guide*. For detailed information about the **Select** statement, see **Select** in the MapBasic *Reference*.

Updating the Currently-Selected Rows

You can use the **Update** statement to modify the Selection table. If you modify the Selection table, the changes that you make are applied to the base table on which the selection is based.

For example, the following **Select** statement selects some of the rows from the employees table. After the **Select** statement, an **Update** statement modifies the data values of the selected rows.

```
Select * from employees
  Where department = "marketing" and salary < 20000

Update Selection
  Set salary = salary * 1.15
```

The **Update** statement will alter the values of rows in the employees table, because the selection is based on the employees table.

Using the Selection for User Input

The Selection process is part of the user interface. Some applications are arranged so that the user selects one or more rows, then chooses an appropriate menu item. When the user makes a selection, the user is specifying an object (a noun). When the user chooses a menu item, the user is specifying an action (a verb) to apply to that object.

The sample program, TextBox, is based on this noun/verb model. The user selects one or more text objects, then chooses the Tools > TextBox > Create Text Boxes command. The TextBox application then queries the Selection table, and draws boxes around the text objects that the user selected.

To query the current selection, use the **SelectionInfo()** function. By calling **SelectionInfo()**, you can determine how many rows are selected (if any) at the present time. If rows are currently selected, you can call **SelectionInfo()** to determine the name of the table from which rows were selected. You then can call **TableInfo()** to query additional information about the table.

If your application includes a sub-procedure called **SelChangedHandler**, MapInfo calls that procedure every time the selection changes. For example, you may want some of your application's custom menu items to only be enabled when rows are selected. To perform that type of selection-specific menu maintenance, create a **SelChangedHandler** procedure. Within the procedure, call

SelectionInfo(SEL_INFO_NROWS) to determine if any rows are selected. Based on whether any rows are selected, issue an **Alter Menu Item** statement that enables or disables appropriate menu items. For more information on menu maintenance, see **Chapter 7: Creating the User Interface**.

Accessing the Cosmetic Layer

Each Map window has one Cosmetic layer, a special-purpose layer which is the top layer in the map. If the user performs a Find operation, MapInfo places a symbol at the “found” location. Such symbols are stored in the Cosmetic layer. In earlier versions of MapInfo, labels were also stored in the Cosmetic layer. Version 4.0, however, treats labels as display attributes, not as Cosmetic objects. See **Chapter 10: Graphical Objects** for more information on labeling.

To control the Cosmetic layer through MapBasic, issue table-manipulation statements (such as **Select**, **Insert**, **Update**, or **Delete**) and specify a table name such as **CosmeticN** (where N is an Integer, one or larger). For example, the table name Cosmetic1 corresponds to the Cosmetic layer of the first Map window on the screen. The following statement selects all objects in that Map window’s Cosmetic layer:

```
Select * From Cosmetic1
```

To determine a Cosmetic layer’s exact table name, call **WindowInfo()** with the code WIN_INFO_TABLE. For example, the following statement deletes all objects from the Cosmetic layer of the active map window (assuming that the active window is a Map window):

```
Delete From WindowInfo(FrontWindow(), WIN_INFO_TABLE)
```

Accessing Layout Windows

MapBasic’s object-manipulation statements can be applied to the objects on a Layout window. To manipulate a Layout window, issue statements that use the table name **LayoutN** (where N is an integer, one or larger).

For example, the table name Layout1 corresponds to the first Layout window that you open. The following statement selects all objects from that Layout window:

```
Select * From Layout1
```

You can determine a Layout window’s exact table name by calling the **WindowInfo()** function with the WIN_INFO_TABLE code.

Note: Objects stored on a Layout window use a special coordinate system, which uses “paper” units (units measured from the upper-left corner of the page layout). Any MapBasic program that creates or queries object coordinates from Layout objects must first issue a **Set CoordSys** statement that specifies the **Layout** coordinate system.

For example, the TextBox sample program draws boxes (rectangle objects) around any currently-selected text objects, regardless of whether the selected text objects are on a Map window or a Layout window. If the selected objects are Layout objects, TextBox issues a **Set CoordSys Layout** statement.

When you are using MapInfo interactively, MapInfo’s Statistics Window gives you an easy way of determining the table name that corresponds to a Layout window or to a Map window’s Cosmetic layer. If you select an object in a Map’s Cosmetic layer, and then show the Statistics Window (for example, by choosing Options > Show Statistics Window), the Statistics window displays a message such as, “Table Cosmetic1 has 1 record selected.” Similarly, if you select an object from a Layout window, the Statistics window displays, “Table Layout1 has 1 record selected.”

Multi-User Editing

If your MapBasic program works with tables in a multiple-user environment, you may encounter file-sharing conflicts. Sharing conflicts occur because MapInfo only allows one user to modify a table at a time.

This section spells out the rules that govern MapInfo's multi-user editing behavior. Read this section if you want to write a MapBasic program that allows multiple users to modify the same table at the same time.

The Rules of Multi-User Editing

MapInfo's multi-user table editing has three restrictions, described below.

- Rule 1: A table may only be edited by one user at a time.

Imagine two hypothetical users: User A and User B. Both users are attempting to use the same table, which is stored on a network.

User A begins editing the table. (For example, User A adds new rows to the table.) Moments later, User B attempts to edit the same table. MapInfo prevents User B from editing the table, and displays the message, "Cannot perform edit. Someone else is currently editing this table." If User B is trying to edit the table through a MapBasic application, a runtime error occurs in the application.

As long as User A continues to edit the table, MapInfo prevents User B from editing the same table. This condition remains until User A performs Save, Revert (discarding the edits), or Close Table.

Note: User B is allowed to read from the table that User A is editing. For example, User B can display the table in a Map window. However, User B will not "see" the edits made by User A until User A performs a Save.

- Rule 2: Users cannot read from a table while it is being saved.

After editing the table, User A chooses the File > Save Table command. Then, while the Save operation is still underway, User B attempts to read data from the table. As long as the Save is underway, MapInfo prevents User B from accessing the table at all. MapInfo displays a dialog box (on User B's computer) with the message, "Cannot access file <tablename>.DAT for read." The dialog contains Retry and Cancel buttons, with the following meaning:

Retry

If User B clicks Retry, MapInfo repeats the attempt to read from the file. The Retry attempt will fail if the Save is still underway. The user can click the Retry button repeatedly. After the Save operation finishes, clicking the Retry button succeeds.

Cancel

If User B clicks Cancel, MapInfo cancels the operation, and the Retry/Cancel dialog box disappears. Note: If User B was loading a workspace when the sharing error occurred, clicking Cancel may halt the loading of the rest of the workspace. For example, a workspace contains **Open Table** statements. If the **Open Table** statement was the statement that caused the sharing conflict, and if the user Cancels the Retry/Cancel dialog, MapInfo will not open the table. Subsequent statements in the workspace may fail because the table was not opened.

- Rule 3: A Save cannot be started while the table is being read by other users.
If other users are reading the table at the exact moment that User A chooses File > Save Table, the Save Table command cannot proceed. MapInfo displays the message, "Cannot open file <tablename>.DAT for writing." The dialog contains Retry and Cancel buttons, with the following meaning:

Retry

If User A clicks Retry, MapInfo repeats the attempt to save the table. The user can click the Retry button repeatedly. Clicking the Retry button will only succeed if the other users have finished reading from the table.

Cancel

If User A clicks Cancel, MapInfo cancels the Save operation, and the Retry/Cancel dialog box disappears. At this point, the table has not been saved, and the edits will not be saved unless User A chooses File > Save Table again.

How to Prevent Conflicts When Reading Shared Data

As discussed in the previous section, some sharing conflicts display a Retry/Cancel dialog box. Ordinarily, the Retry/Cancel dialog box appears at the moment a sharing conflict occurs. However, a MapBasic program can suppress the dialog box by using the **Set File Timeout** statement.

In the parts of your program where you open or read from a shared table, use the **Set File Timeout** statement with a value larger than zero. For example, if you have a procedure that opens several tables, you may want to issue this statement at the start of the procedure:

```
Set File Timeout 100
```

The **Set File Timeout** statement sets a time limit; in this example, the time limit is 100 seconds. In other words, MapInfo will automatically retry any table operations that produce a sharing conflict, and MapInfo will continue to retry the operation for up to 100 seconds. Note that MapInfo retries the table operations *instead of* displaying a Retry/Cancel dialog. If the sharing conflict still occurs after 100 seconds of retries, the automatic retry stops, and MapInfo displays the Retry/Cancel dialog box.

Preventing Conflicts When Writing Shared Data

Several MapBasic statements alter the contents of a table. For example, the **Insert** statement adds new rows to a table. If your program attempts to alter the contents of a table, and a sharing conflict occurs, a MapBasic runtime error occurs. To trap this error, use the **OnError** statement. For example, if you have a procedure that inserts new rows into a table (as in the example below), you should create an error-handling routine, and place an **OnError** statement at the top of the procedure to enable error trapping. (Error-handling is discussed in more detail in **Chapter 6: Debugging and Trapping Runtime Errors**.)

CAUTION: Use the **Set File Timeout** statement and the **OnError** statement exclusively. In places where an error handler is enabled, the file-timeout value should be zero. In places where the file-timeout value is non-zero, error handling should be disabled. The following example demonstrates this logic.

```
Function MakeNewRow(ByVal new_name As String) As Logical

'turn off automatic retries
  Set File Timeout 0

'turn off window redraws
  Set Event Processing Off

'enable error-trapping
  OnError Goto trap_the_error

'Add a new row, and save the new row immediately.
  Insert Into Sitelist ("Name") Values ( new_name )
  Commit Table Sitelist

'Set return value to indicate success.
  MakeNewRow = TRUE

exit_ramp:

  Set Event Processing On
  Exit Function

trap_the_error:
  ' The program jumps here if the Insert or Commit
  ' statements cause runtime errors (which will happen
  ' if another user is already editing the table).

  If Ask("Edit failed; try again?", "Yes", "No") Then
    ' ... then the user wants to try again.
    Resume 0
  Else
    ' the user does not want to retry the operation.
    ' If the Insert succeeded, and we're getting an error
    ' during Commit, we should discard our edits.
    Rollback Table Sitelist

    ' set function's return value to indicate failure:
    MakeNewRow = FALSE
    Resume exit_ramp
  End If
End Function
```

Note the following points:

- When you modify a shared table, try to minimize the amount of time that the table has unsaved edits. In the example above, the **Commit** statement follows immediately after the **Insert** statement, so that there is very little time during which the table has unsaved edits.
- The example uses **Set Event Processing Off** to suspend event processing; as a result, MapInfo will not redraw any windows during the edit. If we did not suspend event processing, the **Insert** statement might cause MapInfo to redraw one or more windows, and the window redraw could conceivably trigger a sharing conflict (for example, because other tables in the same Map window may have a sharing conflict).
- This function sets file-timeout to zero. The procedure that calls this function may need to reset file-timeout to its previous value.

Opening a Table for Writing

When you open a table in a multiple-user environment, there is a chance that MapInfo will open the table with read-only access, even if the files that comprise the table are not read-only. If a MapBasic program issues an **Open Table** statement at the exact moment that the table is being accessed by another user, MapInfo may open the table with a read-only status. The read-only status prevents successive statements from modifying the table.

The following example shows how to prevent MapInfo from opening shared tables with a read-only status. Instead of simply issuing an **Open Table** statement, issue the statement within a loop that iterates until the file is opened read/write.

```
Retry_point:

Open Table "G:\MapInfo\World"
If TableInfo("World", TAB_INFO_READONLY) Then
    Close Table World
    Goto Retry_point
End If
```

Files that Make Up a Table

A table consists of several files: one file contains information about the table structure (column names, etc.); another file contains the table's row-and-column values; another file contains the table's graphic objects (if any); and the remaining files contain indexes. The file containing the row-and-column data can be in any format supported by MapInfo: .dbf, Lotus .wks or .wk1 format, delimited ASCII file format, or Excel .xls file format.

- *filename.tab*: Describes the structure of your table.
- *filename.dat* or *filename.dbf* or *filename.wks*: Contains tabular (row-and-column) data.
- *filename.map*: Contains the table's graphic objects.
- *filename.id*: Contains a geographic index.
- *filename.ind*: Contains indexes for columns in the table.

Because each table consists of several component files, you must be very careful when renaming a table. To rename a table, choose MapInfo's Table > Maintenance > Rename Table command, or issue the MapBasic **Rename Table** statement.

Raster Image Tables

Raster image tables (tables that display only raster image data, not vector data) do not have all of the component files listed above, because raster image tables do not contain tabular data. Every raster image table consists of at least two files: a .tab file (which stores the image's control points) and the file or files that store the raster image. For example, if a raster image table is based on the file photo.tif, the table might consist of two files: photo.tif and photo.tab.

In many ways, a raster image table is just like any other table. To open a raster image table, use an **Open Table** statement. To display a raster image table in a Map window, issue a **Map** statement. To add a raster image table to an existing map, issue an **Add Map Layer** statement. However, you cannot perform a **Select** operation on a raster image table. To determine if a table is a raster table, call **TableInfo()** with the TAB_INFO_TYPE code. If the table is a raster table, **TableInfo()** returns the code TAB_TYPE_IMAGE. As a rule, MapInfo does not alter the original image file on which a raster table is based. Therefore:

- If you use the **Drop Table** statement to delete a raster table, MapInfo deletes the table file, but does not delete the image file on which the table is based.
- If you use the **Rename Table** statement on a raster table, MapInfo renames the table file, but does not rename the image file on which the table is based.
- If you use the **Commit** statement to copy a raster table, MapInfo copies the table file but does not copy the image file on which the table is based.

A raster image table's .tab file is created when a user completes MapInfo's Image Registration dialog. If you need to create a .tab file for a raster image through a MapBasic program, create the file using standard file input/output statements: create the file using the **Open File** statement, and write text to the file using the **Print #** statement; see example below.

The following program creates a table file to accompany a raster image file. This program assigns “dummy” coordinates, not true geographic coordinates. Therefore, the final table will not be suitable for overlaying vector map layers. However, if the raster image is a non-map image (as a company logo), the use of non-geographic coordinates is not a problem.

```

Include "mapbasic.def"
Declare Sub Main
Declare Function register_nonmap_image(ByVal filename As String,
                                      ByVal tablename As String) As Logical

Sub Main
  Dim fname, tname As String
  fname = "c:\data\raster\photo.gif" 'name of an existing image
  tname = PathToDirectory$(fname)
        + PathToTableName$(fname) + ".tab" 'name of table to create
  If FileExists(tname) Then
    Note "The image file is already registered; stopping."
  Else
    If register_nonmap_image(fname, tname) Then
      Note "Table file created for the image file: "
        + fname + "."
    Else
      Note "Could not create table file."
    End If
  End If
End Sub

Function register_nonmap_image( ByVal filename As String,
                              ByVal tablename As String) As Logical
  register_nonmap_image = FALSE
  OnError GoTo handler
  Open File  tablename  For Output  As #1  FileType "Mita"
  Print #1, "!Table"
  Print #1, "!Version 300"
  Print #1, "!charset Neutral"
  Print #1
  Print #1, "Definition Table"
  Print #1, "  File "" + filename + """"
  Print #1, "  Type ""RASTER"" "
  Print #1, "  (1,1) (1,1) Label ""Pt 1"", "
  Print #1, "  (5,1) (5,1) Label ""Pt 2"", "
  Print #1, "  (5,5) (5,5) Label ""Pt 3"" "
  Print #1, "  CoordSys NonEarth Units ""mm"" "
  Print #1, "  Units ""mm"" "
  Print #1, "  RasterStyle 1 45" ' Brightness; default is 50
  Print #1, "  RasterStyle 2 60" ' Contrast; default is 50
  Close File #1
  register_nonmap_image = TRUE ' set function return value
last_exit:
  Exit Function
handler:
  Close File #1
  Resume last_exit
End Function

```


Working With Metadata

What is Metadata?

Metadata is data that is stored in a table's .TAB file, instead of being stored as rows and columns. For example, if you want to record summary information about who edited a table or when they performed the edits, you could store that information as metadata.

Metadata is not displayed in the standard MapInfo user interface. Users cannot see a table's metadata (unless they display the .TAB file in a text editor or run the TableMgr sample MBX). However, MapBasic applications can read and write metadata values.

Each table can have zero or more metadata *keys*. Each key represents an information category, such as an author's name, a copyright notice, etc. For example, a key named "\Copyright" might have the value "Copyright 1999 Acme Corp."

What Do Metadata Keys Look Like?

Each metadata key has a name, which always starts with the "\" (backslash) character. The key name never ends with a backslash character. Key names are not case-sensitive.

The key's value is always a string, up to 239 characters long.

The following table provides samples of metadata keys and key values.

Sample Key Name	Sample Key Value
"\Copyright Notice"	Copyright 2001 Bryan Corp."
"Info"	"Tax Parcels Map"
"Info Author"	"Meghan Marie"
"Info\Date\Start"	"12/14/01"
"Info\Date\End"	"12/31/01"
"IsReadOnly"	"FALSE"

Note the following points:

- Spaces are allowed within key names and within key values.
- You can define a hierarchy of keys by using key names that have two or more backslash characters. In the table above, several of the keys belong to a hierarchy that starts with the "Info" key. Arranging keys in hierarchies allows you to work with an entire hierarchy at a time (for example, you can delete an entire hierarchy with a single statement).
- "\IsReadOnly" is a special key, reserved for internal use by MapInfo. When you add metadata to a table, MapInfo automatically creates the \IsReadOnly key. Do not attempt to modify the \IsReadOnly key.
- The table above shows each string within quotation marks to emphasize that they are string values. However, when you retrieve keys from a table, the strings retrieved by MapBasic do not actually include quotation marks.

Examples of Working With Metadata

The **GetMetadata\$()** function allows you to query a table's metadata, but only if you already know the exact name of the metadata key. If you know that a table has a key called "\Copyright" then the following function call returns the value of that key:

```
s_variable = GetMetadata$(table_name, "\Copyright")
```

The **Metadata** statement allows you to create, modify, or query a table's metadata, even if you do not know the names of the keys. The following examples demonstrate the various actions that you can perform using the **Metadata** statement. Note: In the following examples, **table_name** represents a string variable that contains the name of an open table.

The following example stores a key value in a table. If the key already exists, this action changes the key's value; if the key does not already exist, this action adds the key to the table's metadata.

```
Metadata Table table_name
  SetKey "\Info\Author" To "Laura Smith"
```

The following statement deletes the "\Info\Author" key from the table.

```
Metadata Table table_name
  Dropkey "\Info\Author"
```

The following statement deletes an entire hierarchy of keys at one time. All keys whose names start with "\Info\" will be deleted.

```
Metadata Table table_name
  Dropkey "\Info" Hierarchical
```

When you use the **Metadata** statement to write or delete metadata, the changes take effect immediately. You do not need to perform a Save operation.

You also can use the **Metadata** statement to read the metadata from a table, even if you do not know the names of the keys. To read a table's metadata:

1. Issue a **Metadata Table ... SetTraverse** statement to initialize a traversal.
2. Issue a **Metadata Traverse ... Next** statement to retrieve a key. This statement retrieves the key's name into one string variable, and retrieves the key's value into another string variable.
3. Continue to issue **Metadata Traverse ... Next** statements to retrieve additional keys. Typically, this statement is issued from within a loop. Once you have exhausted the keys, **Metadata Traverse ... Next** returns an empty string as the key name.
4. Terminate the traversal by issuing a **Metadata Traverse ... Destroy** statement. This action releases the memory used by the traversal.

The following example shows how to traverse a table's metadata.

```
Sub Print_Metadata(ByVal table_name As String)

    Dim i_traversal As Integer
    Dim s_keyname, s_keyvalue As String

    ' Initialize the traversal. Specify "\" as the
    ' starting key, so that the traversal will start
    ' with the very first key.
    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
```

For a complete listing of the syntax of the **Metadata** statement, see **Metadata** in the MapBasic *Reference* or online Help.

Working With Seamless Tables

What is a Seamless Table?

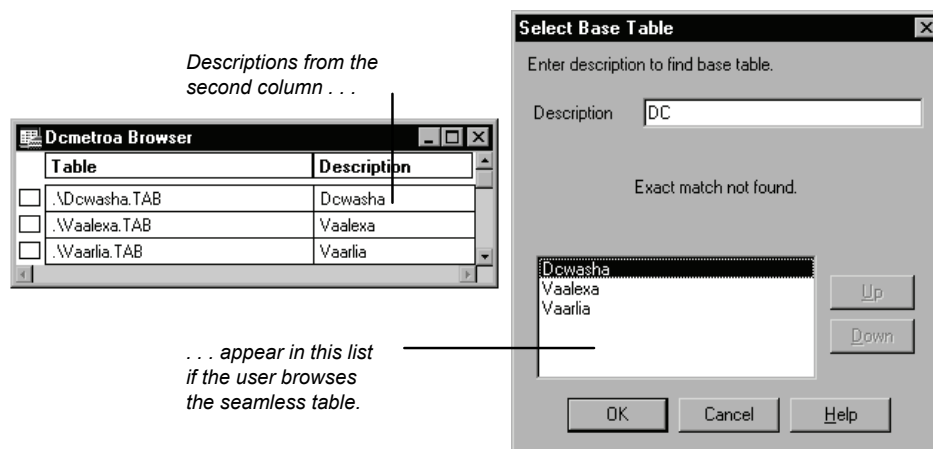
Seamless tables allow you to group multiple tables together and treat them as a single table. Once you have grouped your tables into a seamless table, you can add the entire group of tables to a Map window very easily, simply by adding the seamless table (in the Layer Control dialog). For an introduction to working with seamless tables, see the MapInfo *User Guide*.

How Do Seamless Tables Work?

MapInfo includes a MapBasic program, Seamless Manager (seammgr.mbx), that allows you to create and manipulate seamless tables. To see how a seamless table is composed, you need to turn the table's "seamless behavior" off, as follows:

1. Open a seamless table, such as DCMetroA.
2. Run the Seamless Manager application.
3. Choose Tools > Seamless Manager > Turn Seamless Off to turn off the seamless attribute for the DCMetroA table.
4. Choose Window > New Browser Window to display the table in a Browser window.

Like ordinary tables, a seamless table has rows and columns. Each row corresponds to a base table that is included in the seamless table.



The first column in a seamless table contains table names. The second column contains descriptions, which appear in the user interface. The table names in the first column may contain directory paths. You can omit the directory paths if the base tables are in the same directory as the seamless table, or if the base tables can be located by the Search Directories path (which is specified as a Preference, in the Directory Preferences dialog).

Every row in a seamless table has a map object attached to it, just as objects are attached to rows in conventional tables. However, the objects in a seamless table are not intended for display. Each row in a seamless table has a rectangle object, which defines the minimum bounding rectangle (MBR) for the table named in the first column. When a user displays a seamless table in a Map window, MapInfo compares the Map window's current extents against the MBRs stored in the table. MapInfo only opens the base tables when necessary (i.e., when the area currently visible in the Map window intersects the table's MBR).

MapBasic Syntax for Seamless Tables

Use the **Set Table** statement to turn a seamless table into a conventional table. For example, if you want to edit the descriptions in a seamless table, you could issue the following statement:

```
Set Table DCMetroA Seamless Off
```

and then edit the table's descriptions in a Browser window.

Call **TableInfo(, TAB_INFO_SEAMLESS)** to determine whether a table is a seamless table.

Call **GetSeamlessSheet()** to display a dialog box that prompts the user to choose one base table from a seamless table.

Limitations of Seamless Tables

All of the base tables in a seamless table must have the same structure (i.e., the same number of columns, the same column names, etc.).

Note that some MapInfo operations cannot be used on seamless tables. For example:

- You cannot simultaneously select objects from more than one base table in a seamless table.
- The MapBasic **Find** statement cannot search an entire seamless table; the **Find** statement can only work with one base table at a time.
- You cannot make a seamless table editable in a Map window.
- You cannot create a thematic map for a seamless table.

Accessing DBMS Data

The preceding discussions showed you how to work with local MapInfo tables, tables on your hard disk, or perhaps on a network file-server. This section describes how MapBasic can access DBMS tables, such as Oracle or SQL Server databases.

MapBasic's remote-data statements and functions all begin with the keyword **Server**, with the exception of the **Unlink** statement. For details on the syntax, see the MapBasic *Reference* or online Help.

How Remote Data Commands Communicate with a Database

MapInfo allows a MapBasic application to connect to multiple databases at one time and issue multiple intermixed SQL statements. This is done through connection handles and statement handles.

Connection handles (or numbers) identify information about a particular connection. MapBasic defines connection handles as variables of type integer (i.e., a connection number). An application receives a connection handle upon connecting to a data source. The connection handle is used to associate subsequent statements with a particular connection.

Statement handles (or numbers) identify information about an SQL statement. MapBasic defines statement handles as variables of type integer (i.e., a statement number). An application must receive a statement handle upon calling the **Server_Execute()** function to submit an SQL request. The statement handle is used to associate subsequent SQL requests, like the Fetch and Close operations, to a particular Select statement.

Connecting and Disconnecting

Before a MapBasic application can begin executing SQL statements to remote databases, it must request a connection using the **Server_Connect** function. Once a successful connection is established, the function returns a connection handle (*hdbc*) for use with subsequent SQL DataLink calls.

```
Dim hdbc As Integer  
hdbc = Server_Connect("ODBC", "DLG=1")
```

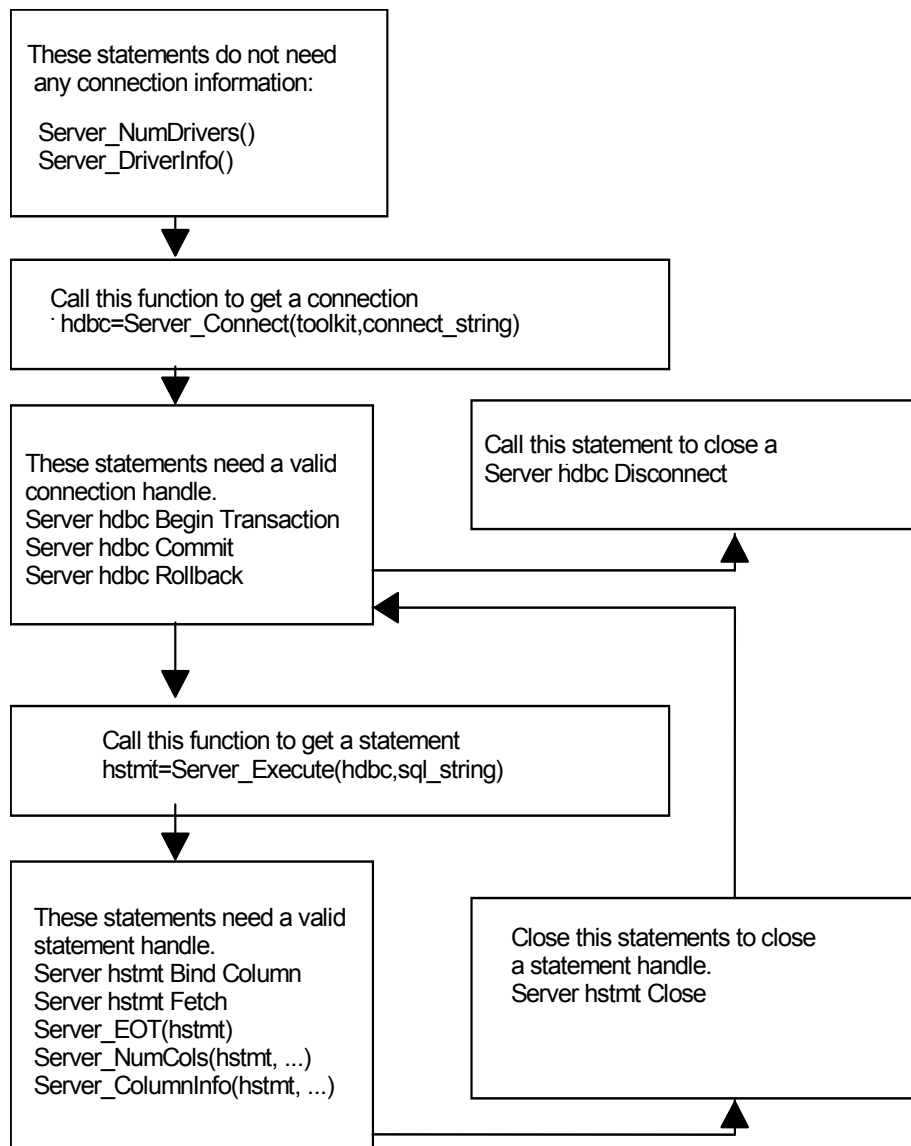
When the driver performs a commit or rollback, it resets all statement requests associated with that connection. The Driver Manager handles the work associated with switching connections while transactions are in progress on the current connection.

Use the following statement to disconnect:

```
Server hdbc Disconnect
```

This statement closes the connection and frees all resources associated with it.

The following chart describes the sequence in which SQL MapBasic Server statements can be issued. There are some statements that require no connection information (for example, **Server_NumDrivers()**), some that require only a connection handle (for example, **Server Commit**), and some that require a statement handle (for example, **Server Fetch**).



You can download an entire table, some rows and columns, or a result set from an ODBC data source using the **Info** feature of the MapBasic statement **Server Fetch**. However, any updates applied to the downloaded table are not applied back to the server database table. Updating remote databases is accomplished by the **Save File** statement.

Accessing/Updating Remote Databases with Linked Tables

A linked table is a special kind of MapInfo table that retains links to a remote database. Edits can be made over multiple MapInfo sessions. Because the linked table updates are occurring outside of an RDBMS transaction, other RDBMS users can update the same rows in the same tables. An optimistic concurrency control mechanism is used to prevent data corruption. Concurrency control is accomplished with the **Automatic/Interactive** clause of the **Commit Table** statement. When the data

is saved, a connection with the remote database is re-established, editing conflicts are resolved, and the changed data is written to the RDBMS. A linked table is created with the MapBasic statement **Server Link Table**.

Linked tables contain information to re-establish connections and identify the remote data to be updated. This information is stored as metadata in the tab file.

An unedited linked table can be refreshed with current data from the remote database without respecifying the connection data, query, and table. A linked table is refreshed with the MapBasic statement **Server Refresh**.

A linked table can be unlinked with the MapBasic statement **Unlink**. Unlinking a table removes the link to the remote database. The end product is a normal MapInfo base table.

Using MapInfo's spatial indexing, users will be able to store and retrieve points in any database; or spatial objects in supported spatial objects. See Appendix E, Making A Remote Table Mappable.

Live Access to Remote Databases

You can access data live from remote databases with the **Register Table** statement. When you specify the **Type** as **ODBC**, the **Register Table** statement tells MapInfo to examine the ODBC table and build a corresponding table file (*filename.TAB*).

Performance Tips for Table Manipulation ---

Minimize Transaction-File Processing

Ordinarily, when a user edits a MapInfo table, MapInfo stores the edits in a temporary file known as a transaction file. As the user performs more and more edits, the transaction file grows larger. A large transaction file can slow down some operations, therefore, if your MapBasic program performs table editing, you may want to take one of the following steps to prevent the transaction file from growing too large:

- Save your edits (i.e., perform a **Commit** statement) regularly. For example, you might set up your program so that it performs a commit after every 100 edits. Saving your edits empties out the transaction file.
- Use a **Set Table ... FastEdit** statement to turn on FastEdit mode. In FastEdit mode, edits are saved immediately to a table, instead of being stored in a transaction file. For details, see the MapBasic *Reference* or online Help. See also: **Set Table ... Undo Off**.

Use Indices Where Appropriate

Some queries are faster if you index one or more columns in your table. For example, **Select** statements can be faster if you index the columns used in **Where**, **Order By**, or **Group By** clauses. However, you may not want to index every single column in your table. Indexing every column can slow down some operations because MapInfo must spend more time maintaining indices. If your application performs intensive table manipulation that does not involve queries, you may be able to improve speed by doing the following:

1. Delete the indices from your table (using the **Drop Index** statement).
2. Perform table edits as necessary.
3. Save your edits.
4. Use the **Create Index** statement to re-create the indices.

This strategy can speed up heavy-duty table manipulation, because MapInfo no longer needs to maintain indices during the editing operations.

Using Sub-Selects

The **Select** statement can include a **Where** clause that performs a sub-select, as described in the MapBasic *Reference*. However, you may find it faster to perform two non-nested **Select** statements, instead of one nested **Select ... Where (Select ...)** statement.

If you perform a sub-select of this type:

```
... Where x = Any( Select ... ) ...
```

then MapInfo does optimize the query performance, but only if column x is indexed.

Optimized Select Statements

Some types of **Select** queries are optimized for fast performance. See **Select** in the MapBasic *Reference* or online Help.

Using Update Statements

MapBasic allows you to update map objects one at a time, by performing an **Alter Object** statement and then an **Update** statement on individual rows, often within a loop. However, this type of table manipulation can be very slow, because you are issuing several statements for every row that you modify.

In some cases, you can obtain much faster performance by issuing a single **Update** statement that affects an entire table, rather than updating one row at a time. For an example, see the topic “Updating Symbols Quickly” in the MapBasic online Help.

File Input/Output

In MapBasic, there is an important distinction between managing *files* and managing MapInfo *tables*. The preceding chapter describes how MapBasic lets you manage tables. This chapter describes how you manage files *that are not tables*.

Sections in this Chapter:

- ♦ Overview of File Input/Output 179
- ♦ Sequential File I/O 180
- ♦ Platform-Specific & International Character Sets 182

Overview of File Input/Output

File input/output (usually abbreviated file i/o) is a process of reading information from files (input) and/or writing information to files (output). The MapBasic language provides a set of standard BASIC input/output statements and functions to let you read and/or write text or binary files. Furthermore, because MapInfo and MapBasic are designed to accommodate different hardware platforms, MapBasic's file i/o statements provide mechanisms that let you ensure seamless sharing of data.

There are three different types of file access: *sequential*, *random*, and *binary*. Which mode you should use depends on the nature of the data in the file(s) you need to access. The three modes are summarized below:

- Use **sequential** file i/o to read text from variable-length text files. For example, if one line of a text file is fifty characters long, and subsequent lines in the text file are longer or shorter than fifty characters, then the file is variable-length. Use sequential file i/o for accessing such files.
- Use **random** file i/o to read from text files that are fixed-length. If every line in a file is exactly 80 characters long, the file is fixed-length, and you can access the file using random file i/o.
- Use **binary** file i/o to access binary (non-text) file data. If you use binary file i/o to store data in a file, MapInfo stores numeric data in an efficient storage format. Binary files containing numerical data cannot be viewed or edited in a text editor, however, they provide a more efficient format for storing numerical data than text files.

Regardless of which type of file i/o you will perform, the first step to performing file i/o is to open the file you want to use. In MapBasic, you open a file using the **Open File** statement. This statement has several optional clauses; which clauses you need to use depends on your specific situation. The following statement opens a text file for sequential input:

```
Open File "settings.txt" For Input As #1
```

When you open a file, you specify a file number; in the example above, the number is one. Later statements in your program refer to the same number that you specified in the **Open File** statement. For example, to read text from the file into a String variable, you could issue a **Line Input** statement, and the **Line Input** statement would refer to the same file number (#1) as the **Open File** statement:

```
Line Input #1, s_nextline
```

If you need to have two or more files open at the same time, make sure that each file is opened under a different number.

In some situations, you may need to create a new file in which to store your data. To create a new file, issue an **Open File** statement that includes the **For Output** clause:

```
Open File "workfile.txt" For Output As #2
```

Alternately, you can specify **For Append** in the **Open File** statement. With Append mode, MapBasic creates the file if it does not already exist, or MapBasic lets you append data to the file if it already does exist. When you are finished reading from or writing to a file, issue a **Close File** statement. For example:

```
Close File #1
```

The number parameter is the same identification number assigned to the file in the **Open File** statement. The pound sign (#) is optional. You do not need to execute a “save” command to save a file that was created or modified through file input/output. You are done modifying the file as soon as you issue the **Close File** statement. (MapBasic does provide a **Save File** statement, but its purpose is to let you copy a file, not save changes to a file.)

There are many ways in which programs can generate runtime errors during file i/o. If the **Open File** statement specifies the wrong file name, or if you attempt to open a file for output, but the file is flagged as read-only, a runtime error will occur. If your program writes data to a file, the program could generate a runtime error if the program runs out of disk space. If you try to open a file for output, but that file is currently being modified by another network user, your program will generate a runtime error. If you are developing an application that performs file input/output, you should build error-handling routines into your program to detect and correct error conditions, and you should test your application under conditions likely to cause problems (for example, out of disk space). For information on how to create an error handler, see Chapter 5.

In some circumstances, you can prevent errors from happening by calling appropriate functions. For example, before you issue an **Open File** statement, you can call the `FileExists()` function to determine whether the file exists. Also, if your program needs to create a temporary, working file, but you do not know what name or directory path to assign to the file (because you do not know the names of your users' directories), call the `TempFileName$()` function. Other statements that are related to file i/o:

- The **Kill** statement deletes a file.
- The **Save File** statement saves a copy of a file.
- The **Rename File** statement changes the name of a file.
- Functions such as **ProgramDirectory\$()**, **HomeDirectory\$()** and **ApplicationDirectory\$()** let you determine different directory paths at runtime. For example, to build a string representing the name of a file that exists in the MapInfo directory (for example, the Startup workspace), when you do not know the name of the directory, call **ProgramDirectory\$()**, to determine where MapInfo is installed.

Sequential File I/O

If you intend to perform sequential file i/o (reading/writing of variable-length text files), there are three different options you can specify in the **Open File** statement's **For** clause: **Input**, **Output**, or **Append**.

Use the **For Input** clause if you intend to read from an existing file. For example, the Named Views sample program (nviews.mb) issues the following statement to open an existing text file for input:

```
Open File view_file For Input As #1
```

The string variable `view_file` contains the name of a text file.

After you open a file for Input, you can read from the file using either the **Input #** statement or the **Line Input #** statement. The **Line Input #** statement reads an entire line from the file into a String variable. With the **Input #** statement, you can treat each line of text as a comma-separated list of values, and read each value into a separate variable. For example, the Named Views application reads data that is formatted in the following manner:

```
"New York", -75.75, 42.83, 557.5  
"Texas", -100.2, 31.29, 1200
```

Each line of the text file contains four values: a name, an x-coordinate, a y-coordinate, and a zoom distance. The Named Views application uses the following **Input #** statement to read each line into four separate variables:

```
Input #1, vlist(tot).descript,  
         vlist(tot).x,  
         vlist(tot).y,  
         vlist(tot).zoom
```

The `vlist` variable is an array of custom type variables.

When you read data sequentially, you need to test to see whether each read was successful. After your program has read the entire contents of the file, if you attempt to read further the read operation will fail. To test whether a read operation was successful, call the **EOF()** function (end-of-file) after each input operation. If the **EOF()** function returns a value of **FALSE**, then you have not yet exhausted the contents of the file (which means that your read was successful). When the **EOF()** function returns **TRUE**, you are at the end of the file.

Note: Reading the last line of the file does not cause the end-of-file condition. The **EOF()** function will only return **TRUE** after you have attempted to read past the end of the file.

To create a file that contains a comma-separated list of expressions, issue an **Open File** statement with the **For Output** clause or the **For Append** clause. After opening the file, use the **Write #** statement to write data to the file. In the **Write #** statement, you can specify a comma-separated list of expressions to be written to each line in the file. For example, the Named Views application issues the following **Write #** statement (within a loop) to create a file with the four values (name, x, y, and zoom) shown above:

```
Write #1, vlist(i).descript, vlist(i).x, vlist(i).y, vlist(i).zoom
```

The **Write #** statement encloses each string expression in double-quotation marks within the file, as shown in the example above ("New York"...). In some situations, using the **Write #** statement may be inappropriate, because you may not want text to be enclosed in quotation marks. To write text to a file without quotation marks, use **Print #** instead of **Write #**.

If you want to read an entire line into one String variable, use the **Line Input #** statement. Use the **Print #** statement to create a file that can later be read using the **Line Input #** statement. For an example of using **Print #** and **Line Input #** to read or write an entire line at once, see the sample program `auto_lib.mb`. The `auto_lib` program reads and writes MapInfo workspace files (specifically, the startup workspace file).

You cannot write to a sequential file that was initially opened for input and you cannot read from a sequential file that was initially opened for output.

Random File I/O

To perform random-access file i/o, specify the **For Random** clause in the **Open** statement:

```
Open File "datafile.dat" For Random As #1 Len = 80
```

When you open a file in **Random** mode, you include a **Len** clause that indicates the number of bytes in each line in the file. Note that any text file contains end-of-line terminators; invisible characters that are embedded in the file to mark the end of each line. The line length specified in the **Len** clause (80 in the example above) specifies the exact number of characters in each record, including any end-of-line terminators (for example, carriage-return/line-feed characters).

After you have opened a file for random access, you can read from or write to the file using the **Get** and **Put** statements; see the *MapBasic Reference*.

Binary File I/O

Binary files are files that contain numeric values stored in binary format. The following statement demonstrates how to open a file for binary access:

```
Open File "settings.dat" For Binary As #1
```

After you have opened a file for binary access, you can read from or write to the file using the **Get** and **Put** statements; see the *MapBasic Reference*.

Numerical data stored in binary format is stored very efficiently. For example, each Integer value is stored using exactly four bytes of the file, regardless of how large the Integer value is. By contrast, if an Integer value is nine digits long (for example, 111,222,333), and you store the value in a text file, the value will occupy nine bytes of the file. Binary storage provides a more efficient format for the storage of non-text data. However, if you need to be able to view your files in a text editor, you should store your data in text files rather than binary files.

The records in a binary file can include character strings, but they must be of fixed length.

Platform-Specific & International Character Sets

If you encounter problems reading text files that originated on another hardware platform or in another country, you may need to use the **Open File** statement's optional **CharSet** clause. 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 degree symbol; however, if Windows is configured to use another country's character set, character code 176 may represent a different character. The fact that different countries use different character sets may cause problems if you need to read a file that originated in a different country.

To correct character set-related misinterpretations, include a **CharSet** clause in your **Open File** statement. The **CharSet** clause lets you explicitly state the character set with which the file was originally created. If you include a **CharSet** clause which correctly identifies the file's origin, MapInfo will correctly interpret data while reading from (or writing to) the file. For a listing of character set names that can be used in a **CharSet** clause, see **CharSet** in the MapBasic *Reference*.

File Information Functions

The following functions return information about an open file:

- **FileAttr()** returns the mode in which the file was opened (INPUT, OUTPUT, APPEND, RANDOM, or BINARY).
- **EOF()** returns a logical TRUE if there has been an attempt to read past the end-of-file, or if the file pointer has been placed past the end-of-file.
- **Seek()** returns the location in the file in offset bytes. On a RANDOM file, this is the number of the last record used times the record length, not the record number alone.
- **LOF()** returns the length of the entire file in bytes.

Each of these functions uses the file number assigned in the **Open File** statement as an argument. For more information, see the MapBasic *Reference* or online Help.

Graphical Objects

Much of MapBasic's power lies in its ability to query and manipulate map objects — arcs, ellipses, frames, lines, points, polylines, rectangles, regions, rounded rectangles, and text objects. This chapter discusses how a MapBasic program can query, create, and modify the objects that make up a map. Note, however, that you need to understand the principles of MapInfo tables before you can understand how MapBasic can store objects in tables. If you have not already done so, you may want to read **Chapter 8: Working With Tables** before reading this chapter.

Sections in this Chapter:

♦ Using Object Variables	185
♦ Using the “Obj” Column	185
♦ Querying An Object's Attributes	187
♦ Creating New Objects	193
♦ Creating Objects Based On Existing Objects	196
♦ Modifying Objects	197
♦ Working With Map Labels	199
♦ Coordinates and Units of Measure	203
♦ Advanced Geographic Queries	205

Using Object Variables

MapBasic's **Object** variable type allows you to work with both simple objects, like lines, and complex objects, like regions. (Visual Basic programmers take note: MapBasic's Object type represents graphical shapes, not OLE objects.)

MapBasic Object variables can be treated much like other variables. You can assign values to object variables, pass object variables as arguments to functions and procedures, and store the values of object variables in a MapInfo table.

Use the **Dim** statement to define an object variable:

```
Dim Myobj, Office As Object
```

You do not have to specify the specific type of object that you want the variable to contain. An object variable can contain any type of map or layout object.

Use the equal sign (=) to assign a value to an object variable, as shown in the next example:

```
Office = CreatePoint(73.45, 42.1)
Myobj = Office
```

You can assign objects from other object variables, functions that return objects, or table expressions of the form *tablename.Obj*. However, there is no syntax for specifying a literal ("hard-coded") object expression.

An object variable holds all of the information that describes a map object. If you store a line object in an object variable, the variable contains both geographic information about the line (for example, the line's starting and ending coordinates) and display information (the line's color, thickness, and style). MapBasic also provides four style variable types (Pen, Brush, Symbol, and Font) that can store styles without storing object coordinates.

Using the "Obj" Column

The column named **Obj** is a special column that refers to a table's graphical objects. Any table that has graphical objects has an Obj column, although the Obj column typically does not appear in any Browser window.

To access the contents of the Object column, use an expression of the form *tablename.obj* (or of the form *tablename.object*). The following example declares an object variable (*current_state*), then copies an object from the *states* table into the variable.

```
Dim current_state As Object
Open Table "states"
Fetch First From states
current_state = states.obj
```

You can perform the same kinds of operations with object columns that you can with regular columns. You can use SQL queries that reference the object column, **Update** the values (objects) in the column, and read its contents into variables.

The following statement creates a query table with state abbreviations and the area of each state; the Obj column is used as one of the parameters to the **Area()** function:

```
Select state, Area(obj, "sq mi")
From states
```

The next example creates a one-row table with the total miles of highway in California:

```
Select Sum(ObjectLen(obj, "mi"))
From highways
Where obj Within (Select obj From states Where state = "CA")
```

Some rows do not contain map objects. For example, if you open a database file as a MapInfo table and geocode the table, the geocoding process attaches point objects to the rows in the table. However, if some of the rows were not geocoded, those rows will not have map objects. To select all the rows that do not have objects, use the condition **Not obj** in the **Select** statement's **Where** clause. The next statement selects all rows that do not have map objects:

```
Select *
From sites
Where Not obj
```

Creating an Object Column

Not all tables are "mappable." For example, if you base a table on a spreadsheet or database file, the file initially cannot be displayed in a Map. To make the table mappable, you must use the Create Map statement, which adds an object column to the table.

To remove the Object column from a table, use the Drop Map statement. Note that Drop Map removes the object column completely. In some cases, you may want to delete individual objects from a table, without deleting the entire Object column; this is sometimes referred to as "un-geocoding" a table. To delete individual object values without removing the Object column, use the Delete Object statement.

To determine whether a table has an Object column, call the TableInfo() function with the TAB_INFO_MAPPABLE code.

Limitations of the Object Column

Object columns have some restrictions that do not apply to other column types. For example, you can only have one object column per table. When you perform a selection that joins two tables, and both tables have object columns, the results table contains only one of the table's objects (the objects from the first table listed in the **Select** statement's **From** clause).

The next example performs a query involving two mappable tables: the states table, and an outlets table, which contains point objects representing retail outlets. The **Select** statement's **From** clause lists both tables. Because the states table is listed first, the results table will contain objects from the the states table.

```
Select *
From states, outlets
Where states.state = outlets.state
Map From selection
```

If you list the outlets table first in the **From** clause, as shown below, the Select statement's results table will contain point objects (outlets), rather than state regions:

```
Select *
  From outlets, states
  Where outlets.state = states.state
Map From selection
```

Each row in a table can contain only one object. Note, however, that an individual object can contain multiple parts. A region object can contain many polygons; thus, a group of islands can be represented as a single region object. Similarly, a polyline object can contain many sections. To determine the number of polygons in a region object or the number of sections in a polyline object, select the object, and choose MapInfo's Edit > Get Info command. To determine the number of sections or polygons from within a program, call the `ObjectInfo()` function with the `OBJ_INFO_NPOLYGONS` code.

Querying An Object's Attributes

A MapInfo table can contain a mixture of different types of objects. For example, a street map might contain a mixture of lines and polylines. You can call the `ObjectInfo()` function with the `OBJ_INFO_TYPE` code to determine the object's type. For details, see `ObjectInfo()` in the *MapBasic Reference* or online Help.

If you are using the MapBasic window interactively, there are various other ways you can display an object's type. For example, you could issue the following statements from the MapBasic window to display a message describing the object's type:

```
Fetch First From world
Note world.obj
```

The following statement selects all Text objects from a Layout window.

```
Select *
  From Layout1
  Where Str$(obj) = "Text"
```

To determine information about an object's geographic coordinates, call the `ObjectGeography()` function. For example, call `ObjectGeography()` if you want to determine the x- and y-coordinates of the end points of a line object. Determining coordinates of nodes in a polyline or region is more complex, because polylines and regions have variable numbers of nodes. To determine coordinates of nodes in a polyline or region, call `ObjectNodeX()` and `ObjectNodeY()`.

To determine an object's centroid, use the `Centroid()` function or the `CentroidX()` and `CentroidY()` functions. To determine an object's minimum bounding rectangle (the smallest rectangle that encompasses all of an object), call the `MBR()` function.

To determine other types of object attributes, call the `ObjectInfo()` function. For example, after you copy an object expression from a table into an Object variable, you can call `ObjectInfo()` to determine the type of object (line, region, etc.), or call `ObjectInfo()` to make a copy of the object's Pen, Brush, Symbol, or Font style. If the object is a text object, you can use `ObjectInfo()` to read the string that comprises the text object.

Many of the standard MapBasic functions take objects as arguments, and return one piece of information about the object as a return value. For example, the **Area()**, **Perimeter()**, and **ObjectLen()** functions take object parameters. The example below calculates the area of a flood zone:

```
Dim floodarea As Float
Open Table "floodmap"
Fetch First From floodmap
floodarea = Area(floodmap.obj, "sq km")
```

Note that labels are not the same as text objects. To query a text object, you call functions such as **ObjectInfo()**. To query a label, you call functions such as **LabelInfo()**. Labels are discussed later in this chapter.

Object Styles (Pen, Brush, Symbol, Font)

Every object has one or more style settings. For example, every line object has a Pen style, which defines the line's color, thickness, and pattern (for example, solid vs. dot-dash), and every Point object has a Symbol style, which defines the point's shape, color, and size. Enclosed objects such as regions have both a Pen style and a Brush (fill) style.

The following table summarizes the four object styles.

Object	Object Style
Pen	Width, pattern, and color of a line
Brush	Pattern, foreground color, and background color of a filled area
Font	Font name, style, size, text color, background color; applies only to text objects
Symbol	<p>For MapInfo 3.0-style symbols: Shape, color, and size attributes.</p> <p>For symbols from TrueType Fonts: Shape, color, size, font name, font style (for example, bold, italic, etc.), and rotation attributes.</p> <p>For custom symbols based on bitmap files: File name, color, size, and style attributes.</p>

For detailed information on the four styles, see **Brush clause**, **Font clause**, **Pen clause**, and **Symbol clause** in the MapBasic *Reference* and online Help.

The MapBasic language provides various statements and functions that allow you to create objects (for example, the **Create Text** statement, the **CreateLine()** function, etc.). Each of the object creation statements has optional clauses to let you specify the style(s) for that object. For example, the **Create Line** statement includes an optional **Pen** clause that lets you specify the line's style. If you issue an object creation statement that does not specify any style settings, MapInfo assigns the current styles to the object.

Note: You cannot use the = operator to compare two style values. For example, the following program, which attempts to compare two Brush variables, will generate a runtime error.

```
Dim b1, b2 As Brush

b1 = MakeBrush(2, 255, 0)
b2 = CurrentBrush()

If b1 = b2 Then
    Note "The two brush styles are equal."
End If
```

If you need to compare two styles, use the **Str\$()** function to convert each style into a string expression. For example, the following statement compares two Brush values:

```
If Str$(b1) = Str$(b2) Then
```

If you need to compare specific elements of a style (for example, to see whether two Symbol styles have the same point size), use the **StyleAttr()** function to extract individual style elements (color, etc.), and then compare the individual elements.

Understanding Font Styles

Every text object has a Font style. A Font style defines the type face (for example, Times Roman vs. Helvetica), text style (for example, bold, italic, etc.), and text color. A Font style also identifies how large the text is, in terms of point size. However, the point size is sometimes ignored. The following list summarizes how a Font's point size affects different types of text.

- When you create a text object in a Layout window, the Font's point size controls the text height. If the Font style specifies 10-point text, the text object is defined with 10-point text. The text might not *display* at 10 points, depending on whether you zoom in or out on the Layout; but when you *print* the Layout, the text height will be 10 points.
- When you use the **Create Text** statement to create a text object in a mappable table, the current font's point size is ignored. In this situation, the text height is controlled by map coordinates, which you specify in the **Create Text** statement. When you issue a **Create Text** statement, you specify two pairs of x- and y-coordinates that define a rectangular area on the map; the text object fills the rectangular area. Because of this design, text objects stored in a mappable table will grow larger as you zoom in, and grow smaller as you zoom out.
- When you use the **CreateText()** function to create a text object in a mappable table, the current font's point size controls the initial size of the text. However, zooming in on the map will cause the text to grow larger.
- When you create a label in a Map window, the Font's point size controls the text height. The text displays and prints at the height specified by the Font style. Note that labels behave differently than text objects stored in a table. Labels are discussed later in this chapter.

A Font style includes a font name, such as “Courier” or “Helvetica.” Font names may be different on each hardware platform; for example, “Geneva” is a common font name on the Macintosh, but not on other platforms. Helv and TmsRmn (or Times New Roman) in the Microsoft Windows environment are called Helvetica and Times on Macintosh and Sun platforms. Helvetica, Times and Courier are recognized in a MapBasic **Font** clause regardless of the platform that is in use at runtime.

Style Variables

MapBasic provides style variable types - Pen, Brush, Symbol, and Font - that correspond to object style attributes. There are several ways you can assign a style to a style variable:

- Build a style expression by calling **MakePen()**, **MakeBrush()**, **MakeFont()**, **MakeSymbol()**, **MakeCustomSymbol()**, or **MakeFontSymbol()**, and assign the value to the style variable. These functions allow you to explicitly specify the desired styles. For example, the ScaleBar sample program calls **MakeBrush()** to build black and white brush expressions, so that the scale bar can have alternating blocks of black and white.
- Call **CurrentPen()**, **CurrentBrush()**, **CurrentFont()**, or **CurrentSymbol()**, and assign the return value to the style variable. These functions read the current styles (i.e., the styles that appear if you choose MapInfo's Options > Line Style, Region Style, Symbol Style, or Text Style command when there are no objects selected).
- Call **ObjectInfo()** to determine the style of an existing object, and assign the return value to a style variable.
- Let the user choose a style through a dialog. If a dialog contains a PenPicker, BrushPicker, SymbolPicker, or FontPicker control, the user can choose a style by clicking on the control. For more information on dialogs, see **Chapter 7: Creating the User Interface**.

The following example demonstrates how to call the **MakePen()** function to construct a Pen style. The Pen style value is assigned to a Pen variable.

```
Dim p_var as Pen
p_var = MakePen(1, 10, RGB(128, 128, 128))
```

The MakePen() function's arguments define the pen style: 1 signifies that the style is one pixel wide, 10 signifies a pattern (dotted), and the RGB() function call specifies a color. For more information about the three parameters that make up a pen style (including a chart of all available line patterns), see Pen clause in the MapBasic *Reference* or online Help. Similarly, for more information about Brush, Font, or Symbol options, see Brush clause, Font clause, or Symbol clause.

The following example demonstrates how to read an existing object's Pen style into a Pen variable:

```
p_var = ObjectInfo(obj_var, OBJ_INFO_PEN)
```

Once you have stored a Pen expression in a Pen variable, you can use the Pen variable within an object creation statement:

```
Create Line Into Variable obj_var
(-73, 42) (-74, 43)
Pen p_var
```

The function StyleAttr() returns one component of a particular style. For example, the TextBox sample program displays a dialog that lets the user choose a pen style; the selected style is stored in the Pen variable, pstyle. TextBox then issues the following statement to read the Pen style's color component into an Integer variable (line_color):

```
line_color = StyleAttr(pstyle, PEN_COLOR)
```

Colors are stored internally as integer numbers. For instance, black is 0 and blue is 255. The **RGB()** function calculates the color value from quantities of red, green, and blue that you specify. For instance, the function call **RGB(0, 255, 0)** returns the color value for green.

Use the **RGB()** function where a color is called for. For example:

```
highway_style = MakePen(2, 2, RGB(0, 0, 255))
```

Alternately, instead of calling **RGB()** you can use one of the standard color definition codes (BLACK, WHITE, RED, GREEN, BLUE, YELLOW, CYAN, and MAGENTA) defined in `mapbasic.def`.

Selecting Objects of a Particular Style

The **ObjectInfo()** function lets you extract a Pen, Brush, Symbol, or Font value from an object. Once you have a Pen, Brush, Symbol or Font, you can call the **StyleAttr()** function to examine individual elements (for example, to determine the color of a Symbol style).

You can use the **Select** statement to select objects based on styles. As the following example shows, the **Select** statement's **Where** clause can call the **ObjectInfo()** and **StyleAttr()** functions, so that **MapInfo** selects only those objects that have certain attributes (for example, objects of a certain color).

The following example adds a custom button to the Tools toolbar. If you select a point object and then click the custom button, this program selects all point objects in the same table that have the same color.

```

Include "mapbasic.def"
Declare Sub Main
Declare Sub SelectPointsByColor()

Sub Main
  ' Add a custom button to the Tools toolbar.
  Alter ButtonPad "Tools" Add
    PushButton
      Calling SelectPointsByColor
      HelpMsg "Select points of same color\nSelect By Color"
End Sub

Sub SelectPointsByColor
  Dim i_color, i_open As Integer
  Dim symbol_style As Symbol
  Dim object_name, table_name As String

  ' Note how many tables are currently open.
  i_open = NumTables()

  ' Determine the name of the table in use.
  table_name = SelectionInfo(SEL_INFO_TABLENAME)
  If table_name = "" Then
    ' ... then nothing is selected; just exit.
    Exit Sub
  End If
  ' Exit if the selection is in a non-mappable table.
  If Not TableInfo(table_name, TAB_INFO_MAPPABLE) Then
    Exit Sub
  End If
  ' See whether the selected object is a Point.
  ' If it is a Point, determine its Symbol and Color.
  Fetch First From Selection
  object_name = Str$(Selection.obj)
  If object_name = "Point" Then
    symbol_style = ObjectInfo(Selection.obj, OBJ_INFO_SYMBOL)
    i_color = StyleAttr(symbol_style, SYMBOL_COLOR)
  End If

  ' Accessing "Selection.obj" may have caused MapInfo to
  ' open a temporary table called Query1 (or Query2...).
  ' Let's close that table, just to be tidy.
  If NumTables() > i_open Then
    Close Table TableInfo(0, TAB_INFO_NAME)
  End If

  If object_name <> "Point" Then
    '...the selected object isn't a point; just exit.
    Exit Sub
  End If
End Sub

```



```
' Select all the rows that contain point objects.
Select * From table_name
  Where Str$(Obj) = "Point"
  Into Color_Query_Prep   NoSelect

' Select those point objects that have the same
' color as the original object selected.
Select * From Color_Query_Prep
  Where
    StyleAttr(ObjectInfo(obj,OBJ_INFO_SYMBOL),SYMBOL_COLOR)
    = i_color
  Into Color_Query

Close Table Color_Query_Prep

End Sub
```

This example works with point objects, but the same techniques could be used with other object types. For example, to work with region objects instead, you would test for the object name "Region" instead of "Point", and call **ObjectInfo()** with OBJ_INFO_BRUSH instead of OBJ_INFO_SYMBOL, etc.

Creating New Objects

MapBasic contains a set of statements and functions through which you can create graphical objects. This section provides an introduction to object-creation statements and functions; for more, see the *MapBasic Reference*.

Object-Creation Statements

The following statements can be used to create new objects. All of the statements may be used to create objects on Layout windows. All of the statements except for **Create Frame** may be used to create objects on Map windows.

- **Create Arc** statement: Creates an arc.
- **Create Ellipse** statement: Creates an ellipse or a circle. (A circle is simply a special case of an arc - an arc with equal width and height.)
- **Create Frame** statement: Creates a frame. Frames are special objects that exist only on Layout windows; each frame can display the contents of an open window. Thus, if you want to place two maps on your page layout, create two frames.
- **Create Line** statement: Creates a line.
- **Create Point** statement: Creates a point.
- **Create Pline** statement: Creates a polyline.
- **Create Rect** statement: Creates a rectangle.
- **Create Region** statement: Creates a region.
- **Create RoundRect** statement: Creates a rounded rectangle.
- **Create Text** statement: Creates a text object.

- **AutoLabel** statement: This statement “labels” a Map window by drawing text objects to the Cosmetic layer. This statement does not create labels, it creates text objects. To create labels, use the **Set Map** statement.

Object-Creation Functions

The following MapBasic functions return object values:

- **CreateCircle()** function: returns a circle object.
- **CreateLine()** function: returns a line object.
- **CreatePoint()** function: returns a point object.
- **CreateText()** function: returns a text object.

In some ways, object-creation functions are more powerful than the corresponding object-creation statements, because a function call can be embedded within a larger statement. For example, the following Update statement uses the CreateCircle() function to create a circle object for every row in the table:

```
Update sites
  Set obj = CreateCircle(lon, lat, 0.1)
```

This example assumes that the sites table has a lon column containing longitude values (x coordinates) and a lat column containing latitude values (y coordinates).

Creating Objects With Variable Numbers of Nodes

Polyline objects and region objects are more complex than other objects in that polylines and regions can have variable numbers of nodes (up to 32,763 nodes per object).

You can create a region object using the **Create Region** statement. In the **Create Region** statement, you can explicitly state the number of nodes that the object will contain. However, there are situations where you may not know in advance how many nodes the object should contain. For example, a program might read object coordinates from a text file, then build a region object that contains one node for each pair of coordinates read from the file. In that situation, the program cannot know in advance how many nodes the object will contain, because the number of nodes depends on the amount of information provided in the file.

If your program will create region or polyline objects, you may want to create those objects in two steps:

1. Issue a **Create Region** statement or a **Create Pline** statement to create an empty object (an object that has no nodes).
2. Issue **Alter Object** statements to add nodes to the empty object. The **Alter Object** statement is usually placed within a loop, so that each iteration of the loop adds one node to the object.

The following example demonstrates this process:

```

Include "mapbasic.def"

Type Point
    x As Float
    y As Float
End Type

Dim objcoord(5) As Point
Dim numnodes, i As Integer, myobj As Object
numnodes = 3
set CoordSys Earth
objcoord(1).x = -89.213  objcoord(1).y = 32.017
objcoord(2).x = -89.204  objcoord(2).y = 32.112
objcoord(3).x = -89.187  objcoord(3).y = 32.096

Create Pline Into Variable myobj 0

For i = 1 to numnodes
    Alter Object myobj Node Add (objcoord(i).x,objcoord(i).y)
Next

Insert Into cables (obj) Values (myobj)

```

Storing Objects In a Table

After you create an object and store it in an Object variable, you usually will want to store the new object in a table. The user will not be able to see the object unless you store the object in a table.

To store an object value in a table, use the **Insert** statement or the **Update** statement. Which statement you should use depends on whether you want to attach the object to an existing row or create a new row to store the object.

Use the **Update** statement to attach an object to an existing row in a table. If that row already has an object, the new object replaces the old object. The **Update** statement can update any column in a table; to update a row's graphical object, refer to the special column name **Obj**.

For example, the following statement stores a point object in the **Obj** column of the first row in the **Sites** table:

```

Update sites
    Set Obj = CreatePoint(x, y)
    Where RowID = 1

```

Use the **Insert** statement to add a new row to a table. **Insert** lets you add one row to a table at a time or insert groups of rows from another table. The following statement inserts one new row into the **Sites** table, and stores a line object in the new row's **Obj** column:

```

Insert Into sites (Obj)
    Values (CreateLine(x1, y1, x2, y2))

```

The **TextBox** sample program demonstrates both the **Insert** statement and the **Update** statement. The **TextBox** application draws a box (a rectangle object) around each selected text object; each box is stored using an **Insert** statement. In addition, if the user checks the **Change Text Color to Match Box Color** check box, the program also changes the color of the selected text object, and then uses an **Update** statement to store the modified text object back in the table.

The **Insert** and **Update** statements are both powerful, flexible table-manipulation statements. In the preceding examples, the statements operated only on one column (the graphical object column, Obj); however, you can manipulate any column of your table using **Insert** and **Update**.

Creating Objects Based On Existing Objects

A MapBasic program can create new objects based on existing objects. This section provides an introduction to various MapBasic statements and functions; for more information about a particular statement or function, see the MapBasic *Reference* or online Help.

Creating a Buffer

A buffer region is a region representing the area within a certain distance of another object or objects. Buffers are useful for locating objects within a certain distance of other objects. For instance, you can create a buffer around a fiber optic cable to find all the dig sites within three hundred meters of the cable. You can use the **Create Object** statement to create buffer regions.

The following example creates a 300-meter buffer region around the selected segment of cable, then searches for dig locations within the buffer:

```
Dim danger_zone As Object

Create Object As Buffer
  From selection
  Into Variable danger_zone
  Width 300 Units "m"

Select * From dig_sites Where dig_site.obj Within danger_zone
```

MapBasic also provides a **Buffer()** function, which returns an object value representing a buffer region.

Using Union, Intersection, and Merge

The **Create Object** statement also can calculate unions and intersections of regions. If you specify **Create Object As Merge**, MapInfo removes common segments from two or more neighboring regions, producing a single, combined region. When two regions with a common border are merged (for example, Nevada and California), the resulting region covers the total area of both regions. The border between the neighboring regions is removed.

The following example demonstrates how to combine two regions from the states table:

```
Select *
  From states
  Where state ="CA" Or state = "NV"

Create Object As Merge
  From selection
  Into Table territory
```

The **Merge** operation is an exclusive-or (XOR) process. If you merge two region objects, and one of the objects is completely contained within the other object, the merge operation removes the smaller object's area from the larger object, leaving a hole.

Merge creates a new object. The two merged regions still exist in the source table. You may want to remove the two original regions, as shown below:

```
Select * From Territory Where TerrName = "Western Territory" or TerrName = "NV"
Delete From selection
```

Create Object As Union and Create Object as Intersection let you create a region that represents logical combinations of two or more regions. These statements are different from Merge because they work with all of the segments of the source regions, not just the common segments. A Union is the total area of all polygons. An Intersection is the overlapping area. The object created by a union or an intersection may contain new nodes that don't appear in the original regions. MapBasic also provides a **Combine()** function, which returns the object produced by combining two other objects.

Creating Offset Copies

A group of Offset functions and statements can be used to produce new objects that are offset from the initial objects by specified units.

The following statements can be used to create offset copies of existing objects.

- **Offset()** function: returns a copy of initial object offset by specified distance and angle.
- **OffsetXY()** function: returns a copy of initial object offset by a specified distance along the X and Y axes.
- **SphericalOffset()** function: returns a copy of initial object by a specified distance and angle. The Distance Type used must be Spherical.
- **SphericalOffsetXY()** function: returns a copy of initial object by a specified distance and angle. The Distance Type used must be Spherical.
- **CartesianOffset()** function: returns a copy of initial object by a specified distance and angle. The Distance Type used must be Cartesian.
- **CartesianOffsetXY()** function: returns a copy of initial object by a specified distance and angle. The Distance Type used must be Cartesian.

Modifying Objects

General Procedure for Modifying an Object

MapBasic provides many statements that you can use to modify an existing map object. Regardless of which statement you use to modify an object, the process of modifying an object is as follows:

1. Make a copy of the original object. (Often, this involves declaring an object variable, issuing a **Fetch** statement to position the row cursor, and issuing an assignment statement of the form *variable_name = tablename.obj*).
2. Issue statements or functions to modify the object. (This often involves issuing one or more **Alter Object** statements.)
3. Issue an **Update** statement to store the modified object back in the table.

The TextBox program demonstrates this process. If the user checks the Change Text Color to Match Box Color check box, the TextBox program uses an **Alter Object** statement to change the color of the selected object, and then uses an **Update** statement to store the altered text object back in the table.

Repositioning An Object

Use the **Objects Move** statement to move objects a specified distance along the positive X axis. You can also specify the Distance Units and Distance Type. Use the **Objects Offset** statement to make a new copy of objects offset a specified distance along the positive X axis. You can also specify the Distance Units and Distance Type and specify whether the copied objects are placed in the same table as the source objects or into a different table.

Moving Objects and Object Nodes

To modify an object's coordinates, issue an **Alter Object** statement that includes a **Geography** clause. You may need to issue more than one **Alter Object** statement (one statement to reset the object's x-coordinate, and another statement to reset the y-coordinate).

Modifying An Object's Pen, Brush, Font, or Symbol Style

The **Alter Object** statement lets you modify an object's style. The example below uses the **Alter Object** command to change a selected object in a table:

```
Include "mapbasic.def"
Dim myobj As Object, mysymbol As Symbol
mysymbol = CurrentSymbol()
Fetch First From selection
myobj = selection.obj
If ObjectInfo(myobj, OBJ_INFO_TYPE) = OBJ_POINT Then
  Alter Object myobj
    Info OBJ_INFO_SYMBOL, mysymbol
  Update selection Set obj = myobj Where RowID = 1
Else
  Note "The selected object is not a point."
End If
```

- To modify the height of a text object that appears on a Layout window, change the object's Font style (by issuing an **Alter Object** statement with an **Info** clause).
- To modify the height of a text object that appears on a Map window, change the object's x- and y-coordinates (by issuing an **Alter Object** statement with a **Geography** clause).
- To modify the height of a map label, issue a **Set Map** statement.

Converting An Object To A Region or Polyline

To convert an object to a region object, call the **ConvertToRegion()** function. To convert an object to a polyline object, call the **ConvertToPline()** function. For more information on these functions, see the MapBasic *Reference* or online Help.

Erasing Part Of An Object

The following statements and functions allow you to erase part of an object:

- The **Overlap()** function takes two object parameters, and returns an object value. The resulting object represents the area where the two objects overlap (the intersection of the two objects).
- The **Erase()** function takes two object parameters, and returns an object value. MapInfo erases the second object's area from the first object, and returns the result.
- The **Objects Intersect** statement erases the parts of the current target objects that are *not* covered by the currently-selected object.
- The **Objects Erase** statement erases part of the currently-designated target object(s), using the currently-selected object as the eraser.

The **Objects Erase** statement corresponds to MapInfo's Objects > Erase command, and the **Objects Intersect** statement corresponds to MapInfo's Objects > Erase Outside command. Both operations operate on the objects that have been designated as the "editing target." The editing target may have been set by the user choosing Objects > Set Target, or it may have been set by the MapBasic **Set Target** statement. For an introduction to the principles of specifying an editing target, see the MapInfo Professional *User Guide*.

Points Of Intersection

As mentioned earlier, you can add nodes to a region or polyline object by issuing an **Alter Object** statement. However, the **Alter Object** statement requires that you explicitly specify any nodes to be added. If you want to add nodes at the locations where two objects intersect, use the **Objects Overlay** statement or the **OverlayNodes()** function.

Call the **IntersectNodes()** function to determine the coordinates of the point(s) at which two objects intersect. **IntersectNodes()** returns a polyline object containing a node at each point of intersection. Call **ObjectInfo()** to determine the number of nodes in the polyline. To determine the coordinates of the points of intersection, call **ObjectNodeX()** and **ObjectNodeY()**.

Working With Map Labels

A map label is treated as a display attribute of a map object. However, MapInfo still supports the **AutoLabel** statement to provide backwards compatibility with older versions of the product in which map labels were text objects in the Cosmetic layer.

Turning Labels On

A MapInfo user can configure labeling options through the Layer Control dialog box. A MapBasic program can accomplish the same results through the **Set Map ... Label** statement. For example, the following statement displays labels for layer 1:

```
Set Map Layer 1 Label Auto On Visibility On
```

Turning Labels Off

In the Layer Control dialog, clearing the Label check box (in the list of layers) turns off the default labels for that layer. This MapBasic statement has the same effect:

```
Set Map Layer 1 Label Auto Off
```

Note: The **Set Map ... Auto Off** statement turns off default (automatic) labels, but it does not affect custom labels (labels that were added or modified by the user). The following statement temporarily hides all labels for a layer — both default labels and custom labels:

```
Set Map Layer 1 Label Visibility Off
```

A MapInfo user can reset a layer's labels to their default state by choosing Map > Clear Custom Labels. This MapBasic statement has the same effect:

```
Set Map Layer 1 Label Default
```

Editing Individual Labels

MapInfo users can edit labels interactively. For example, to hide a label, click on the label to select it, and press Delete. To move a label, click the label and drag.

To modify individual labels through MapBasic, use a **Set Map ... Label** statement that includes one or more **Object** clauses. For example, the following statement hides two of the labels in a Map window:

```
Set Map Layer 1 Label
    Object 1 Visibility Off
    Object 3 Visibility Off
```

For each label you want to customize, include an **Object** clause. In this example, **Object 1** refers to the label for the table's first row, and **Object 3** refers to the label for the table's third row. To save custom labels, save a workspace file; see the MapBasic **Save Workspace** statement.

CAUTION: **Packing a table can invalidate custom (edited) labels previously stored in workspaces. When you store edited labels by saving a workspace, the labels are represented as Set Map ... Object ... statements. Each Object clause refers to a row number in the table. If the table contains rows that have been marked deleted (i.e., rows that appear grayed out in a Browser window), packing the table eliminates the deleted rows, which can change the row numbers of the remaining rows.**

In other words, if you pack a table and then load a previously-saved workspace, any edited labels contained in the workspace may be incorrect. Therefore, if you intend to pack a table, you should do so *before* creating custom labels.

If the only deleted rows in the table appear at the very end of the table (i.e., at the bottom of a Browser window), then packing the table will not invalidate labels in workspaces.

Querying Labels

Querying a Map window's labels is a two-step process:

1. Initialize MapBasic's internal label pointer by calling **LabelFindFirst()**, **LabelFindByID()**, or **LabelFindNext()**.
2. Call **Labelinfo()** to query the "current" label.

For an example, see **Labelinfo()** in the MapBasic Help, or see the sample program, LABELER.MB.

Other Examples of the Set Map Statement

To see the MapBasic syntax that corresponds to the Layer Control dialog, do the following:

1. Open the MapBasic window.
2. Make a Map window the active window.
3. Choose Map > Layer Control to display the Layer Control dialog box.
4. Select the desired options, and click OK.

MapInfo applies your changes, and displays a **Set Map** statement in the MapBasic window. You can copy the text out of the MapBasic window and paste it into your program.

To see the MapBasic syntax that corresponds to editing an individual label, do the following:

1. Modify the labels in your Map window. (Move a label, delete a label, change a label's font, etc.)
2. Save a workspace file.
3. View the workspace file in a text editor, such as the MapBasic editor. Edits to individual labels are represented as **Set Map ... Layer ... Label ... Object** statements in the workspace.

Differences Between Labels and Text Objects

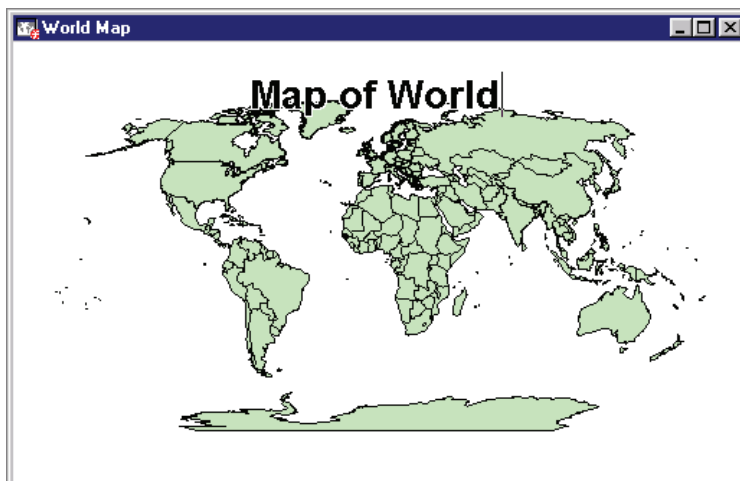
The following table summarizes the differences between text objects and labels.

	Text objects	Labels
MapBasic statements used to create the text:	AutoLabel , Create Text , CreateText()	Set Map
MapBasic statements used to modify the text:	Alter Object	Set Map
MapBasic functions used to query the text (for example, to determine its color):	ObjectInfo(), ObjectGeography()	LabelFindByID(), LabelFindFirst(), LabelFindNext(), Labelinfo()
MapBasic statement used to select the text:	Select	MapBasic programs cannot select labels.
Saving text in a Map:	Text objects can be stored in mappable tables.	Labels are only stored in workspaces.
Saving text in a Layout:	Text objects created in a Layout can be saved in a workspace.	Not applicable. Labels cannot appear in layouts (except when a map is in a layout).
Controlling the text height:	Text height is affected by the current map scale. Text grows larger as you zoom in, and smaller as you zoom out.	A label's text height is controlled by its font. Zooming in or out does not affect a label's text height.
Converting between text and labels:	Not applicable. Given a text object, there is no MapBasic function that returns a Label.	Given a label, the Labelinfo() function can return a text object that approximates the label. See LABELER.MBX for an example.

When you create a label, you specify the label's anchor point (in x- and y-coordinates). For example, if you are viewing a map of the World table, this statement creates a label that acts as a title:

```
Set Map Layer 1 Label Object 1
  Visibility On          'show this record's label
  Anchor (0, 85)         'anchor the label at this (x,y)
  Text "Map of World"    'set label's text
  Position Center        'set position relative to anchor
  Font("Arial",289,20,0) 'set font style (20-point, etc.)
```

The resulting label can act as a map title.



If you need to place text on your map, you may find it easier to create labels, rather than text objects. You could create a table whose sole purpose is to be labeled, using this procedure:

1. Create a table (using the **Create Table** statement) that contains a character column. Make the character column wide enough to store the text that you want to appear on the map. Make the table mappable (using the **Create Map** statement).
2. Add the table to your Map window (using the **Add Map** statement). Use the **Set Map** statement to set the table's labeling options (font, **Auto On**, etc.).
3. When you want to add text to the map, insert a point or line object into the table, using an invisible symbol style (shape 31) or invisible pen style (pattern 1). The object will not be visible, but its label will appear. (Use line objects if you want the text to be rotated. Note: The sample program COGOLine.mb demonstrates how to create a line object at a specific angle.)

Note: With this strategy, you do not need to use **Set Map ... Object** statements to customize each label's position. You can display labels at their default positions. Then, if you want to move a label, move the object that corresponds to the label.

Coordinates and Units of Measure

A MapBasic application can work in only one coordinate system at a time. MapBasic uses Earth coordinates, non-Earth coordinates, or Layout coordinates. The fact that MapBasic has a current coordinate system gives rise to the following programming guidelines:

- Before you create, modify, or query objects from an Earth map, make sure that MapBasic is working in an Earth coordinate system. This is the default. With many MapBasic applications you do not need to worry about coordinate systems.
- Before creating, modifying, or querying objects from a non-Earth map, make sure that MapBasic is working in a non-Earth coordinate system. To do this, issue a **Set CoordSys Nonearth** statement.
- Before creating, modifying, or querying objects from a Layout window, make sure that MapBasic is working in a Layout coordinate system. To do this, issue a **Set CoordSys Layout** statement.

Each MapBasic application has a **CoordSys** system setting that represents the coordinate system currently in use by that application. The default coordinate system setting is the Earth (longitude, latitude) system. By default, every MapBasic application can work with objects from Earth maps, and most MapInfo tables fall into this category. If a MapBasic application needs to work with objects on a Layout window, you must first issue a **Set CoordSys Layout** statement, as follows:

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

The **Set CoordSys Layout** statement lets you specify a paper unit name, such as "in" (inches). This dictates how MapBasic will interpret Layout window coordinate information. To work in centimeters or millimeters, specify the unit name as cm or mm respectively. The following program opens a Layout Window, then places a title on the layout by creating a text object. Since the object is created on a Layout window, the **Create Text** statement is preceded by a **Set CoordSys Layout** statement.

```
Include "mapbasic.def"

Dim win_num As Integer
Layout
win_num = FrontWindow()
Set CoordSys Layout Units "in"

Create Text
  Into Window win_num
  "Title Goes Here"
  (3.0, 0.5) (5.4, 1.0)
  Font MakeFont("Helvetica", 1, 24, BLUE, WHITE)
```

In the example above, the Layout coordinate system uses inches as the unit of measure. All of the coordinates specified in the **Create Text** statement represent inches. After you change the coordinate system through the **Set CoordSys** statement, the new coordinate system remains in effect until you explicitly change it back. Every MapBasic application has its own coordinate system setting. This allows one application to issue a **Set CoordSys** statement without interfering with any other applications that are running.

The MapBasic coordinate system is independent of the coordinate system used by any MapInfo Map window. The default coordinate system is latitude/longitude (NAD 1927) (represented by decimal degrees, not degrees, minutes and seconds.)

All coordinates specified in MapBasic statements or functions should be in latitude and longitude unless you change the MapBasic coordinate system with the **Set CoordSys** statement. For example, the function **Centroidx()** returns the longitude of an object's centroid in decimal degrees, by default, even if the object is stored in a table or a window that has been assigned a different coordinate system. For example, the selection resulting from the statement below has the values: WY -107.554 43, the longitude and latitude of the centroid of Wyoming:

```
Select state, CentroidX(obj), CentroidY(obj)
From states
Where state = "WY"
```

After the following statements are executed, the selection contains: WY -934612.97 2279518.38; the coordinates reflect an Albers projection.

```
Set CoordSys Earth Projection 9, 62, "m", -96, 23, 29.5, 45.5, 0, 0
Select state, CentroidX(obj), CentroidY(obj)
From states
Where state = "WY"
```

To reset the MapBasic coordinate system to its default, issue the following statement:

```
Set CoordSys Earth
```

Units of Measure

MapBasic programs deal with the following units of measure:

- Area units, such as square miles and acres, represent measurements of geographic areas. For a complete list of the area units supported by MapBasic, see **Set Area Units** in the *MapBasic Reference*. Because different area units are supported, functions such as **Area()** can return results in whatever units are appropriate to your application.
- Distance units, such as kilometers and miles, represent measurements of geographic distance. For a list of distance units supported by MapBasic, see **Set Distance Units** in the *MapBasic Reference*.
- Paper units, such as inches or centimeters, represent non-geographic distances. For example, if you issue a **Set Window** statement to reset the width or height of a Map window, you specify the window's new size in paper units, such as inches (on the screen).

At any point during a MapInfo session, there is a current distance unit, a current area unit, and a current paper unit. The default units are miles, square miles, and inches, respectively. The effect of default units is best illustrated by example. The following statement creates a circle object:

```
obj_var = CreateCircle(x, y, 5)
```

Because MapBasic's default distance unit is miles, the circle object will have a radius of five miles. However, if you reset the distance unit by issuing a **Set Distance Units** statement, the meaning of the radius parameter (5) changes. Thus, the following example creates a circle object with a radius of 5 kilometers:

```
Set Distance Units "km"
obj_var = CreateCircle(x, y, 5)
```

To reset the current area unit or the current paper unit, use the **Set Area Units** statement or the **Set Paper Units** statement, respectively.

Advanced Geographic Queries

MapBasic programs can perform complex data queries that take both tabular and graphical data into account. For example, your program can use the **Add Column** statement to calculate totals and averages of data values within a region, based on how the region object overlaps and intersects objects in other map layers.

To understand how MapBasic and MapInfo can perform data-driven geographic analysis, you must understand how MapBasic programs can manage and query tables. If you have not already done so, you may want to read **Chapter 8: Working With Tables** before reading this section.

Using Geographic Comparison Operators

MapBasic does not allow you to use the equal operator (=) to perform logical comparisons of objects (If object_a = object_b). However, MapBasic does provide several geographic operators that let you compare objects to see how they relate spatially. The MapBasic comparison operators **Contains**, **Within**, and **Intersects** and the optional modifiers **Part** and **Entire** allow you to compare objects in much the same way as the relational operator can be used with numbers.

Below is an example of a geographic comparison in an **If...Then** statement:

```
If Parcel_Object Within Residential_Zone_Obj Then
  Note "Your Property is zoned residential."
End If
```

The example below illustrates a geographic comparison in a **Select** statement:

```
Select * From wetlands
Where obj Contains Part myproject
```

At least one of the objects used in a **Within** and **Contains** condition should be an object that represents an enclosed area: regions, ellipses, rectangles, or rounded rectangles.

Whether you use **Within** or **Contains** depends on the order of the objects in the expression. The rule is as follows:

- Use **Within** to test whether the first object is inside the second object.
- Use **Contains** to test whether the first object has the second object inside of it.

For example, when comparing points with regions:

Points are Within regions.
Regions Contain points.

The following statement selects the state(s) containing a distribution center object:

```
Select * From states
Where obj Contains distribution_ctr
```

The next statement selects all of the landfills within a county:

```
Select * From landfill
Where obj Within county_obj
```

The **Within** operator and the **Contains** operator test whether the *centroid* of an object is inside the other object. Use **Entire(ly)** to test whether the whole object is inside another object. Use **Part(ly)** to test whether any part of an object is within the other object.

The next statement selects all sections of a highway with any part going through a county:

```
Select * From highway
Where obj Partly Within countyobj
```

The **Partly Within** operator tests whether any portion of the first object is within the other object or touching it at any point. You also can use the **Entirely Within** operator to test if all of an object is within the area of another object. Since checking all of the segments of an object involves more calculations than checking only the centroid, conditions that involve the **Partly** modifier or the **Entirely** modifier evaluate more slowly.

The **Intersects** operator can be used with all types of objects. If any part of an object crosses, touches, or is within the other object, the objects intersect. Regions that touch at one corner intersect. A point on a node of a polyline intersects the polyline, lines that cross intersect, and a point inside a region intersects that region.

The table below summarizes MapBasic's geographic operators:

Operator	Usage	Evaluates TRUE if:
Contains	objectA Contains objectB	first object contains the centroid of the second object
Contains Part	objectA Contains Part objectB	first object contains part of the second object
Contains Entire	objectA Contains Entire objectB	first object contains all of the second object
Within	objectA Within objectB	first object's centroid is within the second object
Partly Within	objectA Partly Within objectB	part of first object is within the second object
Entirely Within	objectA Entirely Within objectB	the first object is entirely inside of the second object
Intersects	objectA Intersects objectB	the two objects intersect at some point

Querying Objects in Tables

You can use MapBasic functions or geographic comparison operators to build queries using the object column of your table. Building these queries is much like building queries for regular columns, except that there are no object literals. Instead, queries using objects typically use functions or comparison operators (for example, Entirely Within) to analyze objects.

The statement below uses the **ObjectLen()** function to find all the sections of cable greater than 300 meters in length:

```
Select *
From cable
Where ObjectLen(obj, "m") > 300
```

The next example calculates the total area of wetlands in Indiana:

```
Select Sum(Area(obj,"sq mi"))
  From wetlands
  Where obj Within (Select obj From states Where state = "IN")
```

The next statement selects all the storage tanks within one kilometer of a well at longitude lon, and latitude lat:

```
Set Distance Units "km"
Select * From tanks Where obj Within
  CreateCircle(lon,lat, 1)
```

The statement below creates a selection with employees and the distance they live from an office (in order of farthest to nearest):

```
Select
  Name, Distance(Centroidx(obj), Centroidy(obj),
                 office_lon, office_lat, "km")
  From employee
  Order By 2 Desc
```

Using Geographic SQL Queries With Subselects

MapBasic allows you to query objects from one table in relation to objects in another table. For instance, you might want to query a table of doctors to see which ones are in Marion County, Indiana. Doctors are in one table, counties in another.

One approach is to select a county from the county table, copy the object into a variable, and query the table of doctors against the object variable. This is how it looks:

```
Dim mycounty As Object
Select *
  From counties
  Where name="Marion" and state="IN"
Fetch First From selection
mycounty = selection.obj
Select *
  From doctors
  Where obj Within mycounty
```

If you use a subselect in the **Where** clause instead of the variable mycounty, you can produce the same results with fewer statements:

```
Select *
  From doctors
  Where obj Within
    (Select obj From counties Where name="Marion" And state="IN")
```

Notice that the subselect (the latter select, which appears in parentheses) returns a table with only one column and one row - the object representing Marion County, Indiana. MapInfo examines each row in the doctors table to determine whether that row is inside Marion County. The subselect performs the same function as the variable in the previous example (mycounty), because it returns the appropriate object to the expression.

To ensure that the subselect returns only the object column, the **Select** clause of the subselect lists only one column, **obj**. The statement will not evaluate properly if there are many columns in the subselect or if the column isn't an **object** column.

Use the Any() operator when the subselect returns multiple rows. The next example shows a subselect that uses Any() to process a group of rows. It finds all the doctors in counties that have a per-capita income of less than \$15,000. Compare the locations with each county in the subselect.

```
Select *
  From doctors
  Where obj Within
    Any (Select obj From counties Where inc_pcap < 15000)
```

Switch the order in the **Select** statement to select counties instead of doctors. The statement below finds all the counties that have a doctor specializing in neurology:

```
Select *
  From counties
  Where obj Contains
    (Select obj From doctors Where specialty = "Neurology")
```

The following example finds all the states bordering Nebraska:

```
Select *
  From states
  Where obj Intersects (Select obj From states Where state = "NE")
```

Using Geographic Joins

Joins *link* two tables together by matching, row-for-row, entries in specified columns from two tables. The result is one table with a combination of columns for both tables with as many rows as there are matches. MapBasic extends the relational concept of a join with geographic join criteria. For instance, if you join demographic data with the states map, the resulting table can have all of the information from the states map as well as the demographic data for each state.

MapInfo supports geographic conditions in the join. For instance, instead of matching two tables by a numeric ID, you can join tables by matching objects from one table that contain an object in the second table. This is particularly useful when there is no matching field. You can join all of the housing projects in a table with their congressional districts without having the congressional district information in the projects table to begin with. Determining the district may be the reason to perform the join in the first place - to see which projects are in which congressional districts. The SQL **Select** statement for that operation is:

```
Select *
  From projects, congdist
  Where projects.obj Within congdist.obj
```

After you have joined the tables geographically, you can use the **Update** statement to enter the congressional district names (from the name column) into the projects table (the column cd) as follows:

```
Update Selection Set cd = name
```

The resulting projects table now contains the name of the congressional district for every project. The following example calculates the total dollars spent on projects in each congressional district:

```
Select congdist.name, sum(project.amt)
  From congdist, project
  Where congdist.obj Contains project.obj
  Group By 1
```

Since the table order in the **Where** clause has changed, use the condition **Contains** instead of **Within**.

Proportional Data Aggregation

The **Add Column** statement can perform advanced polygon-overlay operations that perform proportional data aggregation, based on the way one table's objects overlap another table's objects. For example, suppose you have one table of town boundaries and another table that represents a region at risk of flooding. Some towns fall partly or entirely within the flood-risk area, while other towns are outside the risk area. The **Add Column** statement can extract demographic information from the town-boundaries table, then use that information to calculate statistics within the flood-risk area. For information about the **Add Column** statement, see the *MapBasic Reference*.

Advanced Features of Microsoft Windows

This chapter discusses how a MapBasic application can take advantage of Windows-specific technology.

Sections in this Chapter:

- ♦ Declaring and Calling Dynamic Link Libraries (DLLs) . . . 211
- ♦ Creating Custom Button Icons and Draw Cursors 216
- ♦ Inter-Application Communication Using DDE 218
- ♦ Incorporating Windows Help Into Your Application 223

Declaring and Calling Dynamic Link Libraries (DLLs)

Dynamic Link Libraries, or DLLs, are files that contain executable routines and other resources (such as custom icons for toolbar buttons). You can use DLLs as libraries of external routines, and call those routines from your MapBasic program. You can issue a **Call** statement to a DLL routine, just as you would use a **Call** statement to call a MapBasic procedure. There are many DLLs available from commercial sources. The documentation for a particular DLL should describe the routines that it contains, its specific name, and any required parameters.

Note: If your MapBasic program calls DLLs, the DLLs must be present at run time. In other words, if you provide your users with your compiled application (MBX file), you must also provide your users with any DLLs called by your MBX.

The Windows DLLs are documented in the *Windows Software Developer's Kit* (SDK). Third-party books that describe the standard Windows files are also available.

Specifying the Library

Before your MapBasic program can call a DLL routine, you must declare the DLL through a **Declare** statement (just as you use the **Declare** statement to declare the sub-procedures in your MapBasic source code). In the **Declare** statement, you specify the name of the DLL file and the name of a routine in the library.

```
Declare Sub my_routine Lib "C:\lib\mylib.dll"  
    (ByVal x As Integer, ByVal y As Integer)
```

If you specify an explicit path in your **Declare** statement (for example, "C:\lib\mylib.dll"), MapInfo tries to load the DLL from that location. If the DLL file is not in that location, MapInfo does not load the DLL (possibly causing runtime errors). If your **Declare** statement specifies a DLL name without a path (for example, "mylib.dll"), MapInfo tries to locate the DLL from various likely locations, in the following order:

1. If the DLL is in the same directory as the .MBX file, MapInfo loads the DLL; otherwise, go to step 2.
2. If the DLL is in the directory where MapInfo is installed, MapInfo loads the DLL; otherwise, go to step 3.
3. If the DLL is in the Windows\System directory, MapInfo loads the DLL; otherwise, go to step 4.
4. If the DLL is in the Windows directory, MapInfo loads the DLL; otherwise, go to step 5.
5. MapInfo searches for the DLL along the user's system search path.

MapInfo follows the same search algorithm when loading bitmap icon and cursor resources from DLLs.

Passing Parameters

Many DLLs take parameters; for example, the example above shows a **Declare** statement for a DLL routine that takes two parameters.

MapBasic can pass parameters two ways: By value (in which case MapInfo copies the arguments onto the stack), or by reference (in which case MapInfo puts the address of your MapBasic variable on the stack; the DLL then can modify your MapBasic variables). For an introduction to the conceptual differences between passing parameters by reference vs. by value, see [Chapter 5: MapBasic Fundamentals](#).

To pass a parameter by value, include the **ByVal** keyword in the **Declare** statement (as shown in the example above). If you omit the **ByVal** keyword, the argument is passed by reference.

The following MapBasic data types may not be passed by value: Arrays, custom data types (i.e., structures), and aliases. Fixed-length string variables may be passed by value, but only if the DLL treats the parameter as a structure. See *String Arguments*, below.

Calling Standard Libraries

The next example shows how a MapBasic program can reference the MessageBeep routine in the standard Windows library known as User.

```
Declare Sub MessageBeep Lib "user"  
  (ByVal x As SmallInt)
```

Note that this **Declare** statement refers to the library name "user" not "user.dll". User is the name of a standard library that is included as part of Windows; other standard Windows library names include GDI and Kernel.

After you declare a DLL routine using a **Declare Sub** statement, you can use the **Call** statement to call the routine the way you would call any sub-procedure:

```
Call MessageBeep(1)
```

Calling a DLL Routine by an Alias

Some DLL routines have names that cannot be used as legal MapBasic identifiers. For example, a DLL routine's name might conflict with the name of a standard MapBasic keyword. In this situation, you can use the **Alias** keyword to refer to the DLL routine by another name.

The following example shows how you could assign the alias Beeper to the MessageBeep routine in the User library:

```
Declare Sub Beeper Lib "user" Alias "MessageBeep"  
  (ByVal x As SmallInt)  
  
Call Beeper(1)
```

Note: The name by which you will call the routine - "Beeper" in this example - appears after the **Sub** keyword; the routine's original name appears after the **Alias** keyword.

String Arguments

When calling a DLL routine, a MapBasic program can pass variable-length string variables by reference. If you are writing your own DLL routine in C, and you want MapBasic to pass a string by reference, define the argument as **char *** from your C program.

CAUTION: When MapBasic passes a by-reference string argument, the DLL routine can modify the contents of the string variable. However, DLL routines should not increase the size of a MapBasic string, even if the string is declared as variable-length in MapBasic.

A MapBasic program can pass fixed-length string variables by reference or by value. However, if you pass the argument by value, the DLL routine must interpret the argument as a C structure. For example, if your MapBasic program passes a 20-character string by value, the DLL could receive the argument as a structure consisting of five four-byte Integer values.

When a MapBasic program passes a string argument to a DLL, MapInfo automatically includes a null character (ANSI zero) to terminate the string. MapInfo appends the null character regardless of whether the MapBasic string variable is fixed-length or variable-length.

If your DLL routine will modify the string argument, make sure that the string is long enough. In other words, take steps within your MapBasic program, so that the string variable that you pass contains a sufficiently long string.

For example, if you need a string that is 100 characters long, your MapBasic program could assign a 100-character string to the variable before you call the DLL routine. The MapBasic function **String\$()** makes it easy to create a string of a specified length. Or you could declare the MapBasic string variable to be a fixed-length string (for example, **Dim stringvar As String * 100** will define a string 100 bytes long). MapBasic automatically pads fixed-length string variables with spaces, if necessary, so that the string length is constant.

Array Arguments

MapBasic allows you to pass entire arrays to DLL routines in the same way that you can pass them to MapBasic sub-procedures. Assuming that a DLL accepts an array as an argument, you can pass a MapBasic array by specifying the array name with empty parentheses.

User-Defined Types

Some DLLs accept custom data types as parameters. (Use the **Type** statement to create custom variable types.) MapBasic passes the address of the first element, and the rest of the elements of the user-defined type are packed in memory following the first element.

CAUTION: For a DLL to work with custom variable types, the DLL must be compiled with “structure packing” set to tightest packing (one-byte boundaries). For example, using the Microsoft C compiler, you can use the **/Zp1** option to specify tightest packing.

Logical Arguments

You cannot pass a MapBasic Logical value to a DLL.

Handles

A handle is a unique integer value defined by the operating environment and used to reference objects such as forms and controls. Operating-environment DLLs use handles to Windows (HWND), Device Contexts (hDC), and so on. Handles are simply ID numbers and you should never perform mathematical functions with them.

If a DLL routine takes a handle as an argument, your MapBasic program should declare the argument as **ByVal Integer**.

If a DLL function returns a handle as its return value, your MapBasic program must declare the function's return value type as **Integer**.

Example: Calling a Routine in KERNEL

The following example illustrates calling a DLL. The DLL in this example, "kernel", is a standard Windows library. This program uses a routine in the kernel library to read a setting from the Windows configuration file, WIN.INI.

```

Declare Sub Main
' Use a Declare Function statement to reference the Windows
' "kernel" library.
Declare Function GetProfileString Lib "kernel" (
    lpszSection As String,
    lpszEntry As String,
    lpszDefault As String,
    lpszReturnBuffer As String,
    ByVal cbReturnBuffer As Smallint)
    As Smallint

Sub Main
    Dim sSection, sEntry, sDefault, sReturn As String
    Dim iReturn As Smallint

    ' read the "sCountry" setting
    ' from the "[intl]" section of WIN.INI.

    sReturn = String$(256, " ")
    sSection = "intl"
    sEntry = "sCountry"
    sDefault = "Not Found"
    iReturn = GetProfileString(sSection, sEntry,
                             sDefault, sReturn, 256)

    ' at this point, sReturn contains a country setting
    ' (for example, "United States")
    Note "[" + sSection + "]" + chr$(10) + sEntry + "=" + sReturn
End Sub

```

The **Declare Function** statement establishes a reference to the kernel library. Note that the library is referred to as "kernel" although the actual name of the file is `krnl386.exe`. Windows uses the correct library if your program refers to "kernel". However, if you create your own library, your **Declare Function** statements should reference the actual name of your DLL file. The kernel library receives special handling because it is a standard part of the Windows API.

If you use DLLs to store custom ButtonPad icons and/or custom draw cursors, you can use the same basic technique - calling **SystemInfo(SYS_INFO_MIPLATFORM)** to determine which DLL to use. However, the MapBasic syntax is somewhat different: Instead of using a **Declare** statement, you reference DLL resources (bitmap icons and cursors) by including a **File** clause in the **Create ButtonPad** statement, as shown in the following example.

```

Declare Sub Main
Declare Function getDLLname() As String
Declare Sub DoIt

Sub Main
    Dim s_dllname As String

    s_dllname = getDLLname()

    Create ButtonPad "Custom" As
        ToolButton Calling doit
            Icon 134 File s_dllname
            Cursor 136 File s_dllname
    End Sub
Function getDLLname() As String
    If SystemInfo(SYS_INFO_MIPLATFORM)
        = MIPLATFORM_WIN32 Then
        getDLLname = "..\icons\Test32.DLL"
    Else
        getDLLname = "..\icons\Test16.DLL"
    End If
End Function

Sub DoIt
    'this procedure called if the user
    'uses the custom button...
End Sub

```

A discussion of creating custom ButtonPad icons appears later in this chapter.

Troubleshooting Tips for DLLs

The following tips may help if you are having trouble creating your own DLLs.

- If you are using C++ to create your own DLLs, note that C++ compilers sometimes append extra characters to the end of your function names. You may want to instruct your C++ compiler to compile your functions as "straight C" to prevent your function names from being changed.
- The Microsoft 32-bit C compiler provides three calling conventions: Standard (keyword `__stdcall`), C (keyword `__cdecl`) and fast call (keyword `__fastcall`). If you are creating DLLs to call from MapBasic, do not use the fast call convention.
- If you are having trouble passing custom MapBasic data types (structures) to your DLL, make sure that your C data structures are "packed" to one-byte boundaries, as discussed above.
- MapBasic can pass arguments by reference (the default) or by value. Note, however, that passing arguments by value is not standardized among compilers; for example, different compilers behave differently in the way that they process C-language doubles by value. Therefore, you may find it more predictable to pass arguments by reference. When you pass an argument by reference, you are passing an address; the major compilers on the market are consistent in their handling of addresses.

- It is good programming to make your DLLs “self-contained.” In other words, each DLL routine should allocate whatever memory it uses, and it should free whatever memory it allocated.
- It is important to set up your MapBasic **Declare** statement correctly, so that it declares the arguments just as the DLL expects the arguments. If a DLL routine expects arguments to be passed by value, but your program attempts to pass the arguments by reference, the routine may fail or return bad data.

Creating Custom Button Icons and Draw Cursors

The MapBasic language lets you control and customize MapInfo's ButtonPads, which are an important part of MapInfo's user interface. For an introduction to how MapBasic can control ButtonPads, see [Chapter 7: Creating the User Interface](#).

A small picture (an icon) appears on each button. You may want to create your own custom icons to go with the custom buttons that you create. The process of creating custom icons varies from platform to platform. On Windows, custom ButtonPad icons are stored as BMP resources in DLL files.

A MapBasic program also can use custom cursors (the shapes that moves with the mouse as you click and drag in a Map or Layout window). This section discusses the process for creating custom cursors for Windows.

Reusing Standard Icons

Before you go about creating your own custom button icons, take a moment to familiarize yourself with the icons that are built into MapInfo. Starting with version 4.0, MapInfo includes a wide assortment of custom icons. These icons are provided to make it easier for MapBasic developers to create custom buttons.

To see a demonstration of the built-in icons, run the sample program Icon Sampler (ICONDEMO.MBX). The following picture shows one of the ButtonPads created by the Icon Sampler.



Each of the icons built into MapInfo has a numeric code. For a listing of the codes, see ICONS.DEF. To see an individual button's code, run ICONDEMO.MBX, and place the mouse cursor over a button; the button's ToolTip shows you the button's code.

If none of MapInfo's built-in icons are appropriate for your application, you will want to create custom icons, as described in the following pages.

Custom Icons

To create custom icons for MapInfo, you need a resource editor. The MapBasic development environment does not include its own resource editor; however, MapBasic programs can use the resources that you create using third-party resource editors. For example, you could create custom icons using AppStudio (the resource editor that is provided with Microsoft Visual C).

On Windows, custom icons are stored in a DLL file. Before you begin creating custom icons, you should develop or acquire a DLL file where you intend to store the icons. This DLL file can be a “stub” file (i.e., a file that does not yet contain any useful routines).

You must create two bitmap resources for each custom icon. The first bitmap resource must be 18 pixels wide by 16 pixels high; this is the icon that will appear if the user does not check the Large Buttons check box in MapInfo’s Toolbar Options dialog box. The second bitmap resource must be 26 pixels wide by 24 pixels tall; this is the icon that will appear if the user does check the Large Buttons check box. You must create both resources.

The process of creating custom bitmaps involves the following steps:

- Acquire or develop the DLL file where you will store your custom icons.
- Edit the DLL using a resource editor, such as AppStudio.
- For each icon you wish to create, add two **bitmap (BMP) resources**: one bitmap that is 18 wide by 16 high, and another bitmap that is 26 wide by 24 high (in pixels).

Note: You must create **bitmap** resources, not **icon** resources.

- Assign sequential ID numbers to the two bitmap resources. For example, if you assign an ID of 100 to the 18 x 16 bitmap, assign an ID of 101 to the 26 x 24 bitmap.

Once you have created the pair of bitmap resources, you can incorporate your custom bitmaps into your MapBasic application using either the **Create ButtonPad** or the **Alter ButtonPad** statement. In your program, refer to the ID of the smaller (18 x 16) bitmap resource. For example, if you assigned the IDs 100 and 101 to your bitmap resources, your program should refer to ID 100, as shown in the following statement:

```
Alter ButtonPad "Tools"
  Add PushButton
    Icon 100 File "MBICONS1.DLL"
    HelpMsg "Add new record"
    Calling new_route
  Show
```

The DLL file where you store your custom icons (in this example, MBICONS1.DLL) must be installed on your user’s system, along with the .MBX file. The DLL file can be installed in any of the following locations: The directory where the .MBX file is located; the directory where the MapInfo software is installed; the user’s Windows directory; the system directory within the Windows directory; or anywhere along the user’s search path. If you place the DLL in any other location, your MapBasic program must specify the directory path explicitly (for example, **Icon 100 File “C:\GIS\MBICONS1.DLL”**). Note that the **ProgramDirectory\$()** and **ApplicationDirectory\$()** functions can help you build directory paths relative to the MapInfo directory or relative to the directory path where your MBX is installed.

Custom Draw Cursors for Windows

The process of creating custom draw cursors is similar to the process of creating custom icons. However, draw cursors have some attributes that do not apply to icons (for example, each draw cursor has a “hot spot”).

To create custom draw cursors, use a resource editor to store CURSOR resources in a DLL. You can store CURSOR resources and BMP resources in the same DLL file.

Inter-Application Communication Using DDE

Inter-Process Communication, or IPC, is the generic term for the exchange of information between separate software packages. Windows supports IPC through the Dynamic Data Exchange protocol, commonly known as DDE.

If two Windows applications both support DDE, the applications can exchange instructions and data. For instance, a DDE-capable Windows package, such as Microsoft Excel, can instruct MapInfo to carry out tasks (for example, Map From World).

Overview of DDE Conversations

A DDE conversation is a process that can take place between two Windows applications. Both applications must be running, and both must support DDE conversations. A single DDE conversation can involve no more than two applications; however, MapInfo can be involved in multiple conversations simultaneously.

In a conversation, one application is active; it begins the conversation. This application is called the **client**. The other, passive application is called the **server**. The client application takes all initiative; for instance, it sends instructions and queries to the server application. The server reacts to the instructions of the client.

How MapBasic Acts as a DDE Client

The MapBasic language supports the following statements and functions that allow a MapBasic application to act as the client in a DDE conversation.

DDEInitiate()	Opens a conversation
DDERequest\$()	Requests information from the server application
DDEPoke	Sends information to the server application
DDEExecute	Instructs the server application to perform an action
DDETerminate	DDETerminateAll

Refer to the MapBasic *Reference* or online Help for detailed information on these statements and functions.

To initiate a DDE conversation, call the **DDEInitiate()** function. **DDEInitiate()** takes two parameters: an **application** name, and a **topic** name.

Typically, the **application** parameter is the name of a potential server application (for example, Excel is the DDE application name of Microsoft Excel). The list of valid **topic** parameters varies depending of the application. Often, the **topic** parameter can be the name of a file or document currently in use by the server application.

For instance, if Excel is currently editing a worksheet file called TRIAL.XLS, then a MapBasic application can initiate a conversation through the following statements:

```
Dim channelnum As Integer
channelnum = DDEInitiate("Excel", "TRIAL.XLS")
```

In this example, Excel is the application name, and TRIAL.XLS is the topic name.

Many DDE applications, including MapInfo, support the special topic name System. You can use the topic name System to initiate a conversation, then use that conversation to obtain a list of the available topics.

Each DDE conversation is said to take place on a unique *channel*. The **DDEInitiate()** function returns an integer channel number. This channel number is used in subsequent DDE-related statements.

Once a conversation has been initiated, the MapBasic application can send commands to the server application by issuing the **DDEExecute** statement. For instance, a MapBasic application could instruct the server application to open or close a file.

A MapBasic application can request information from the server application by calling the **DDERequest\$()** function. When calling **DDERequest\$()**, you must specify an item name. A DDE item name tells the server application exactly what piece of information to return. If the server application is a spreadsheet, the item name might be a cell name.

Use the **DDEPoke** statement to send information to the server. Generally, when a MapBasic application pokes a value to the server application, the value is stored in the appropriate document, as if it had been entered by the user. The following example shows how a MapBasic program can store the text "NorthEast Territory" in a cell in the DDE server's worksheet.

```
DDEPoke channelnum, "R1C2", "NorthEast Territory"
```

Once a DDE conversation has completed its task, the MapBasic (client) application should terminate the conversation by issuing a **DDETerminate** or **DDETerminateAll** statement. **DDETerminate** closes one specific DDE conversation; **DDETerminateAll** closes all open DDE conversations that were opened by that same application. Multiple MapBasic applications can be in use at one time, with each application conducting its own set of DDE conversations.

When a MapBasic application acts as a DDE client, the application may generate runtime errors if the server application "times-out" (does not respond to the client's actions within a certain amount of time). On Windows 3.1, this time-out behavior is controlled by a setting in the file MAPINFO.INI. MapInfo creates a [MapInfo Common] section of MAPINFO.INI. The [MapInfo Common] section can contain a DDeTimeout setting, such as the following:

```
DDeTimeout=10000
```

The number represents a length of time, in milliseconds; the default value is ten thousand (ten seconds). If your application encounters time-out errors while acting as a DDE client, you may want to edit MAPINFO.INI to specify a larger setting than the default.

If you are running a 32-bit version of MapInfo, the time-out setting is stored in the Windows registry instead of in a .INI file. For more details about how MapInfo stores settings in the registry, search for “registry” in the MapBasic online Help index.

How MapInfo Acts as a DDE Server

MapInfo acts as the server when another Windows application initiates the DDE conversation. This allows the client application to read from MapBasic global variables and even poke values into MapBasic global variables. The DDE client can also perform execute operations to run MapBasic statements; for example, the client could use DDE execute functionality to issue a MapBasic **Map** statement. (However, the client cannot issue MapBasic flow-control statements.)

Other software packages do not necessarily provide the same set of DDE statements that MapBasic provides. While MapBasic provides a **DDEPoke** statement, other packages may provide the same functionality under a different name. To learn what DDE statements are provided by a particular Windows application, refer to the documentation for that application.

Any application that acts as a DDE client must address the three basic DDE parameters application, topic, and item that were described above.

Application name: Specify MapInfo as the application name to initiate a DDE conversation with MapInfo as the server.

Topic name: Specify System or specify the name of a MapBasic application that is currently running (for example, SCALEBAR.MBX).

Item name: The item name that you specify depends on the topic you use. If you use MapInfo as the application name and System as the topic name, you can use any item name from the table below.

Application:“MapInfo”

Topic:“System”

Actions and Items Supported by the DDE Conversation:

DDE action	DDE item name	Effect
Peek request	"SysItems"	MapInfo returns a TAB-separated list of item names accepted under the System topic: Topics SysItems Formats Version
Peek request	"Topics"	MapInfo returns a TAB-separated list of currently available topics (System, and the names of all running MapBasic applications).
Peek request	"Formats"	MapInfo returns a list of all Clipboard formats supported by MapInfo (TEXT).
Peek request	"Version"	MapInfo returns a text string representing the MapInfo version number, multiplied by 100. For example, MapInfo 4.0.0 returns "400". See example below.
Peek request	A MapBasic expression	MapInfo interprets the string as a MapBasic expression and returns the value as a string. If expression is invalid, MapInfo returns an error. This functionality applies to MapInfo 4.0 and higher.
Execute	A text message	MapInfo tries to execute the message as a MapBasic statement, as if the user had typed the statement into the MapBasic window. The statement cannot contain calls to user-defined functions, although it can contain calls to standard functions. The statement cannot reference variables that are defined in compiled applications (.MBX files). However, the statement can reference variables that were defined by executing Dim statements into the MapBasic window.

For example, the following MapBasic program—which you can type directly into the MapBasic window—conducts a simple DDE conversation using "MapInfo" as the application and "System" as the topic.

```
Dim i_channel As Integer
i_channel = DDEInitiate("MapInfo", "System")
Print DDERequest$(i_channel, "Version")
DDETerminate i_channel
```

The **DDEInitiate()** function call initiates the DDE conversation. Then the **DDERequest\$()** function performs a peek request, using "Version" as the item name.

If you use the name of a running MapBasic application (for example, "C:\MB\SCALEBAR.MBX", or "SCALEBAR.MBX", or "SCALEBAR") as the DDE topic name, you can use any item name from the table below.

Application: "MapInfo"

Topic: *The name of a running MapBasic application*

Actions and Items Supported by the DDE Conversation:

DDE action	DDE item name	Effect
Peek request	-{items}"	MapInfo returns a TAB-separated list of the global variables defined by the running application. See example below.
Peek request	The name of a global variable	MapInfo returns a string representing the value of the variable.
Peek request	A string that is not the name of a global variable	If the MapBasic application has a function called RemoteQueryHandler() , MapInfo calls the function. The function can determine the item name by calling: CommandInfo(CMD_INFO_MSG) This functionality is new in MapInfo 4.0.
Poke	The name of a global variable	MapInfo stores the new value in the variable.
Execute	A text message	If the MapBasic application has a procedure called RemoteMsgHandler , MapInfo calls the procedure. The procedure can determine the text message by calling: CommandInfo(CMD_INFO_MSG)

For example, the following MapBasic program — which you can type directly into the MapBasic window — conducts a simple DDE conversation using "SCALEBAR.MBX" as the topic. This conversation prints a list of the global variables used by SCALEBAR.MBX.

Note: This conversation will only work if the application SCALEBAR.MBX is already running.

```
Dim i_channel As Integer
i_channel = DDEInitiate("MapInfo", "SCALEBAR.MBX")
Print DDERequest$(i_channel, "{items}" )
DDETerminate i_channel
```

How MapInfo Handles DDE Execute Messages

There are two ways that the client application can send MapInfo an execute message:

- When a conversation uses "System" as the topic, and the client application sends an execute message, MapInfo tries to execute the specified message as a MapBasic statement.
- When a conversation uses the name of a MapBasic application as the topic, and the client sends an execute message, MapInfo calls the application's **RemoteMsgHandler** procedure, which can then call **CommandInfo()** to determine the text of the execute message.

A MapBasic application can act as the client in one DDE conversation, while acting as the server in another conversation. A MapBasic application can initiate a conversation with another MapBasic application, or with MapInfo itself.

Communicating With Visual Basic Using DDE

Many MapBasic programmers use Microsoft's Visual Basic language to enhance their MapBasic applications. You might use Visual Basic to create elaborate dialog boxes that would be difficult to create using the MapBasic **Dialog** statement. For example, a Visual Basic program can create custom controls that are not available through MapBasic's **Dialog** statement.

MapBasic applications can communicate with Visual Basic applications using DDE (or using OLE Automation). For more information about communicating with Visual Basic applications, see [Chapter 12: Integrated Mapping](#).

Examples of DDE Conversations

For an example of using DDE to read and write values of cells in a Microsoft Excel worksheet, see **DDEInitiate()** in the MapBasic *Reference* or online Help.

The sample program, AppInfo (APPINFO.MBX), provides a more complex DDE example. The AppInfo program is a debugging tool. If you run your MapBasic application, and then you run AppInfo, you can use AppInfo to monitor the global variables in your MapBasic program. The WhatApps() procedure queries the DDE item name "Topics" to retrieve the list of running MBX files. The WhatGlobals() procedure conducts another DDE conversation, using the "{Items}" item name to retrieve the list of global variable names.

DDE Advise Links

When MapInfo acts as a server in a DDE conversation, the conversation can support both warm and hot advise links. In other words, when a Windows application initiates a DDE conversation that monitors the values of MapBasic global variables, Windows is able to notify the DDE client when and if the values of the MapBasic global variables change.

When a MapBasic application acts as a client in a DDE conversation, there is no mechanism for creating an advise link.

Incorporating Windows Help Into Your Application

If you are developing a complex application, you may want to develop an online help file that explains the application. To create a help file, you need a help compiler. The MapBasic development environment does not include a help compiler. However, if you already own a Windows help compiler, and you use it to create a Windows help file, you can control the help file through a MapBasic application. Note: MapInfo's Technical Support staff cannot assist you with the creation of on-line help files.

Within your program, you can control the Help window by using the **Open Window**, **Close Window** and **Set Window** statements. The following statement opens the Help window, showing the Contents screen of the MapInfo help file:

```
Set Window Help Contents
```

The **Set Window** statement has many uses; see the MapBasic *Reference* for details. Most forms of the **Set Window** statement require an Integer window identifier, but if you specify the **Help** keyword, you should omit the Integer identifier - there is only one Help window.

If you create a custom help file, and call the file Dispatch.hlp, the following statement displays your help file in the Help window:

```
Set Window Help File "C:\MAPINFO\DISPATCH.HLP"
```

The following statement sets the Help window so that it displays the help screen that has 500 as its context ID number:

```
Set Window Help ID 500
```

Context ID numbers (such as 500 in the preceding example) are defined in the [MAP] section of a help file's Project file (for example, *filename.hpj*). For more information about the architecture of a Windows help file, see the documentation for the Windows Software Developers Kit (SDK).

If you want to provide a help screen for a specific dialog in your application, place a Button control in the dialog, and assign the Button a title called "Help."

```
Control Button  
Title "Help"  
Calling show_help_sub
```

Assign the Help Button control a handler procedure, and have the handler procedure issue a **Set Window** statement. The user will be able to obtain help for the dialog by clicking the Help button. For more information about assigning handler procedures to dialog controls, see **Chapter 7: Creating the User Interface**.

Integrated Mapping

You can control MapInfo Professional for using programming languages other than MapBasic. For example, if you know how to program in Visual Basic, you can integrate a MapInfo Map window into your Visual Basic application, while doing most — maybe even all — of your programming in Visual Basic. This type of application development is known as Integrated Mapping, because you are integrating elements of MapInfo into another application.

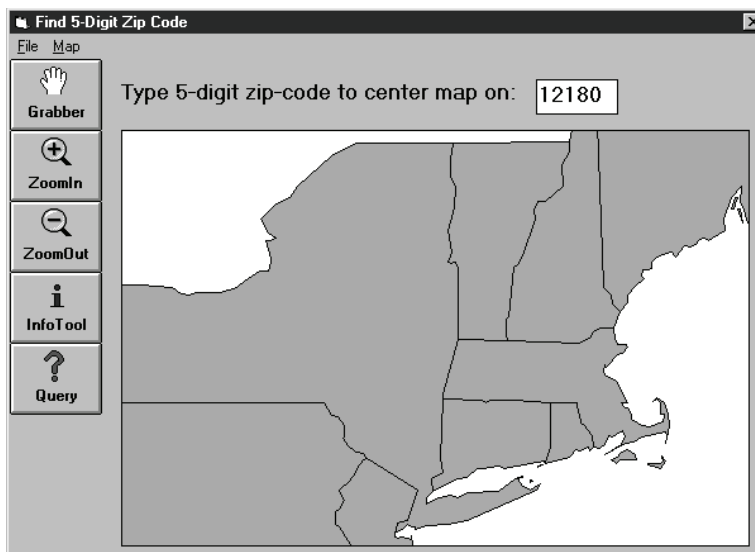
If you already know how to program in other programming languages, such as C or in Visual Basic, you will find that Integrated Mapping provides the easiest way to integrate MapInfo windows into non-MapBasic applications.

Sections in this Chapter:

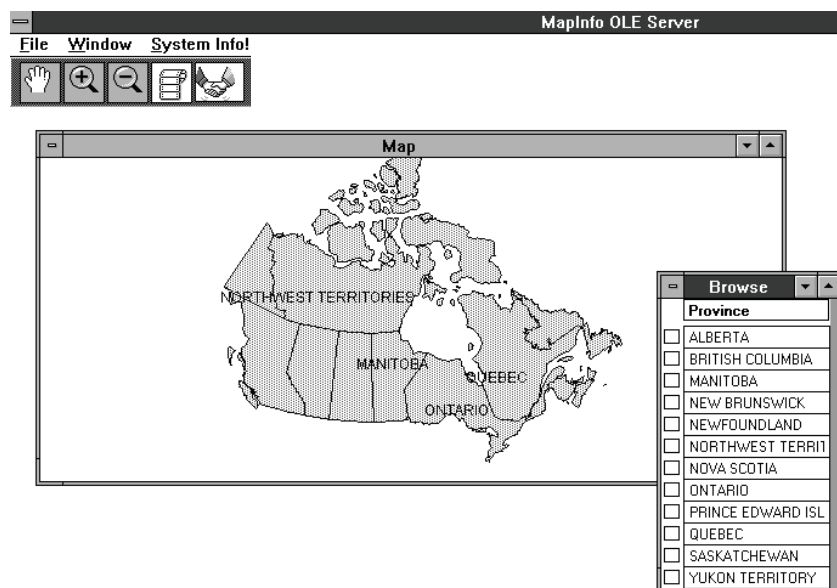
- ♦ **What Does Integrated Mapping Look Like? 226**
- ♦ **Conceptual Overview of Integrated Mapping 227**
- ♦ **Technical Overview of Integrated Mapping 228**
- ♦ **A Short Sample Program: “Hello, (Map of) World” 229**
- ♦ **A Closer Look at Integrated Mapping. 229**
- ♦ **Using Callbacks to Retrieve Info from MapInfo. 237**
- ♦ **Alternatives to Using OLE Callbacks 241**
- ♦ **Related MapBasic Statements and Functions 243**
- ♦ **MapInfo Command-Line Arguments. 253**
- ♦ **Adding Toolbar Buttons and Handlers 258**
- ♦ **Learning More 261**

What Does Integrated Mapping Look Like?

You control the appearance of the Integrated Mapping application. If you want, you can create a user interface that is radically different from the MapInfo user interface. For example, the following picture shows the FindZip application (a sample Visual Basic application that integrates a MapInfo Map window into a Visual Basic form).



The following picture shows a multiple-document interface (MDI) application, also written in Visual Basic, that includes MapInfo Map and Browser windows.



When you integrate a map into your program, the user sees a genuine MapInfo Map window — not a bitmap, metafile, or any other type of snapshot. You can allow the user to interact with the map (for example, using the Zoom tools to magnify the map). An integrated Map window has all of the capabilities of a Map window within MapInfo.

Note: When the user runs an Integrated Mapping application, the MapInfo “splash screen” (the image that ordinarily displays while MapInfo is loading) does not appear.

Conceptual Overview of Integrated Mapping

To create an Integrated Mapping application, you write a program — but not a MapBasic program. Integrated Mapping applications can be written in several languages. The most often-used languages are C and Visual Basic. The code examples in this chapter use Visual Basic.

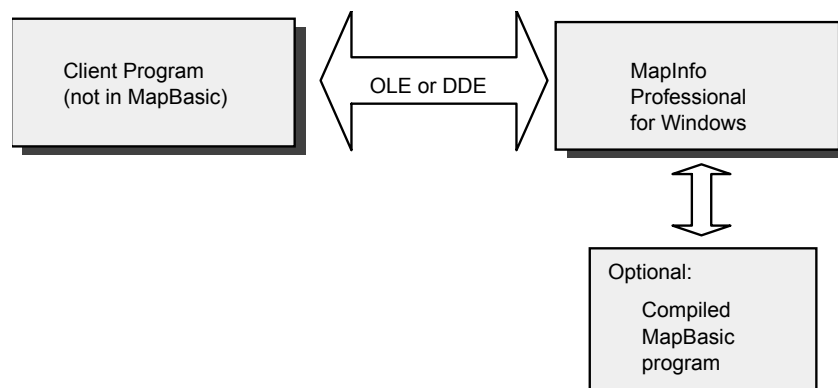
Within your program, you issue a statement to launch MapInfo in the background. For example, if you are using Visual Basic, you could launch MapInfo by calling Visual Basic’s **CreateObject()** function. MapInfo launches silently in the background, without displaying a splash screen.

Your program manipulates MapInfo by constructing strings that represent MapBasic statements, using OLE Automation (or DDE, if you prefer) to send the strings to MapInfo. MapInfo executes the statements as if you had typed the statements into the MapBasic window.

If you want to open a Map window, use MapBasic’s **Map From** statement, just as you would in a conventional MapBasic program. But in an Integrated Mapping application, you also issue additional statements (for example, **Set Next Document Parent**) to make the Map window become a child window of your application. This process is known as “reparenting” the window. You can reparent Map, Browse, Graph, Layout, and Legend windows.

Note: Reparenting MapInfo’s windows into another application does not give MapInfo access to the other application’s data. Before you can display data in a MapInfo window, you must store the data in a MapInfo table.

This illustration shows the major elements of an Integrated Mapping application:



Note that the compiled MapBasic program (.MBX) element is optional. For some applications, you might not need to create a compiled MapBasic program. However, if you have already written MapBasic programs, you can continue to use your existing MapBasic code as part of an Integrated Mapping solution.

Technical Overview of Integrated Mapping

System Requirements

- Integrated Mapping requires MapInfo 4.0 or later. You may use a full copy of MapInfo or a MapInfo runtime (a special “stripped-down” version of MapInfo, sold only as the base for custom applications).
- Your user’s computer must have enough free memory and system resources to run both your client program and MapInfo simultaneously.
- Your client program (for example, your Visual Basic program) must be able to act as an OLE Automation controller or as a DDE client. OLE Automation is strongly recommended, because it is faster and more reliable than DDE. Automation also provides better error reporting than DDE. MapInfo uses OLE properties to report runtime error codes; if you use DDE instead of OLE, you cannot retrieve runtime error codes.

Your client program must be able to create a user-interface element (for example, a window, form, or control) as a place-holder for where the map will go. Your client program must also be able to determine the Windows HWND value of the user-interface element.

For example, in Visual Basic you can place a PictureBox control on a form. When you send a command to MapInfo, telling MapInfo to create a map inside the PictureBox, you must specify the PictureBox’s HWND.

Other Technical Notes

- To develop an Integrated Mapping application, you must write a program in a language other than MapBasic. (We refer to this program as the **client program**.) You can write the client program using various popular development products, such as C/C++, Visual Basic (3.0 or later), PowerBuilder, or Delphi.
- Integrated Mapping uses OLE Automation, but does not use OLE Embedding. When you want to place a MapInfo Map window into your application, you do not embed it; instead, you “reparent” the window by sending MapInfo a series of command strings. The end result is that MapInfo windows appear to the user as child windows of your application.
- Integrated Mapping does not involve VBX controls or OCX controls. The MapInfo software does include some DLLs, but you do not call those DLLs directly; those DLLs are used internally by MapInfo.

A Short Sample Program: “Hello, (Map of) World”

The following Visual Basic example will give you a sense of how easy it is to integrate MapInfo windows into another application.

Create a new Visual Basic project. In the project's General Declarations procedure, declare an Object variable. (In this example, we will name the variable **mi**, but you can use any variable name you like.)

```
Dim mi As Object
```

Next, add statements to the Form_Load procedure, so that the procedure looks like this:

```
Sub Form_Load()  
  
    Set mi = CreateObject("MapInfo.application")  
    mi.do "Set Application Window " & Form1.hWnd  
    mi.do "Set Next Document Parent " & Form1.hWnd & " Style 1"  
    mi.do "Open Table ""World"" Interactive Map From World"  
    mi.RunMenuCommand 1702  
    mi.do "Create Menu ""MapperShortcut"" ID 17 As ""(-"" "  
  
End Sub
```

When you run this Visual Basic program, it launches MapInfo in the background, and creates a Map window. The Map window behaves as a child window of the Visual Basic program. The following sections provide detailed explanations of each step in the Integrated Mapping process.

A Closer Look at Integrated Mapping

The following section explains how to integrate elements of MapInfo into a Visual Basic application. This discussion is written with two assumptions:

- You should already understand the basic terms and concepts of Windows programming. For example, you should know what a “child window” is. For background information on the concepts of Windows programming, see the documentation for your programming language.
- You should already know how to program in Visual Basic, because the code examples in this discussion use Visual Basic syntax. However, even if you are not a Visual Basic developer, you should read this section. The basic concepts and procedures discussed in this section also apply to other programming languages.

Starting MapInfo Professional

To start a unique instance of MapInfo Professional, call Visual Basic's **CreateObject()** function, and assign the return value to a Visual Basic Object variable. (Tip: You may want to make the Object variable global, otherwise, the MapInfo object is released when you exit the local procedure.) For example, if you named your Object variable “mapinfo” then the following statement launches MapInfo:

```
Set mapinfo = CreateObject("MapInfo.Application")
```

To attach to a previously-running instance of MapInfo which was *not* launched by a **CreateObject()** call, use Visual Basic's **GetObject()** function.

```
Set mapinfo = GetObject( , "MapInfo.Application")
```

Note: If you are working with a MapInfo runtime instead of a full copy of MapInfo, specify "MapInfo.Runtime" instead of "MapInfo.Application". Note that a MapInfo runtime and a full copy of MapInfo can run simultaneously.

The **CreateObject()** and **GetObject()** functions use OLE Automation to connect to MapInfo. If you need to use DDE rather than OLE, use Visual Basic's **Shell()** function to start MapInfo, and then use the **LinkMode** property to establish the DDE connection.

Under 32-bit Windows (Windows 95 or Windows NT), multiple instances of MapInfo can be running simultaneously. If you launch MapInfo, and then launch an Integrated Mapping application that calls **CreateObject()**, two separate instances of MapInfo will be running. However, under 16-bit versions of Windows, only one instance of MapInfo can run at a time. Under Windows 3.1, if you are already running MapInfo, and then you launch an Integrated Mapping application that calls **CreateObject()**, the Integrated Mapping application will not be able to launch another instance of MapInfo.

Sending Commands to MapInfo

After launching MapInfo, construct text strings that represent MapBasic statements. For example, if you want MapInfo to execute a MapBasic **Open Table** statement, you might construct the following string (within Visual Basic):

```
msg = "Open Table ""STATES.TAB"" Interactive "
```

If you connected to MapInfo using OLE Automation, send the command string to MapInfo by using the **Do** method. For example:

```
mapinfo.Do msg
```

When you use the **Do** method, MapInfo executes the command string as if you had typed the command into the MapBasic window.

If you connected to MapInfo using DDE, send the command string to MapInfo by using the DDE **LinkExecute** method.

Querying Data from MapInfo

To query the value of a MapBasic expression, construct a string that represents the expression. For example, if you want to determine the value returned by the MapBasic function call **WindowID(0)**, construct the following string (within Visual Basic):

```
msg = "WindowID(0)"
```

If you connected to MapInfo using OLE Automation, send the expression string to MapInfo by using the **Eval** OLE method. For example:

```
Dim result As String  
result = mapinfo.Eval "WindowID(0)"
```

When you use the **Eval** method, MapInfo interprets the string as a MapBasic expression, determines the value of the expression, and returns the value, as a string. Note: If the expression has a Logical value, MapInfo returns a one-character string, "T" or "F".

If you connected to MapInfo using DDE, query the value by using the DDE **LinkRequest** method.

Reparenting MapInfo Windows

After you launch MapInfo, use the MapBasic statement **Set Application Window** so that MapInfo dialog boxes and error messages are owned by your client program. (In the following statement, "FormName" is the name of a form in Visual Basic.)

```
msg = "Set Application Window " & FormName.hWnd  
mapinfo.Do msg
```

Then, whenever you want to integrate a MapInfo window into the Visual Basic application, send MapInfo a **Set Next Document** statement, followed by the MapBasic statement that creates the window. For example, the following commands create a MapInfo Map window as a child window of the Visual Basic program. ("MapFrame" is the name of a PictureBox control in Visual Basic.)

```
msg = "Set Next Document Parent " & MapFrame.hWnd & " Style 1"  
mapinfo.Do msg  
  
msg = "Map From States"  
mapinfo.Do msg
```

The **Set Next Document** statement lets you "reparent" document windows. Within the **Set Next Document** statement, you specify the HWND (handle) of a control in your Visual Basic program. The next time you create a MapInfo window (using the **Map**, **Graph**, **Browse**, **Layout**, or **Create Legend** statements), the newly-created window is reparented, so that it has your client program as its parent.

The **Set Next Document** statement includes a **Style** clause which controls the type of window you will create. The example above specifies **Style 1** which produces a child window with no border. You could specify **Style 2** to produce a popup window with a half-height title bar (like MapInfo's Legend window), or **Style 3** to produce a popup window with a full-height title bar.

For each window that you reparent, issue a pair of statements — a **Set Next Document Parent** statement, followed by the statement that creates the window. After you create the window, you may want to query the value "WindowID(0)" to obtain MapInfo's Integer Window ID for the new window. (Many MapBasic statements require that you know the window's ID.)

```
mapid = Val(mapinfo.eval("WindowID(0)"))
```

Note that even after you have reparented a Map window, MapInfo maintains that window. If part of the window needs to be repainted, MapInfo automatically repaints it. Therefore, your client program can simply ignore any erase or repaint messages pertaining to the reparented window.

If you are working in C, you might not be able to ignore erase messages. In this case you should set your parent window's style to include the WS_CLIPCHILDREN window style.

Reparenting Legends, Raster Dialogs and Other Special Windows

MapInfo has several modeless windows, including the Info window, Ruler window, Message window, the Raster related dialogs, and Statistics window. To reparent one of these special “floating” windows, use MapBasic’s **Set Window ... Parent** statement. For example, the FindZip sample program uses the following statement to reparent the Info window:

```
mapinfo.do "Set Window Info Parent " & FindZipForm.hWnd
```

Note that the process for reparenting the Info window is different than the process for reparenting Map windows. When reparenting the Info window, you do not use the **Set Next Document** statement. The process is different because there is only one Info window, whereas you can have numerous Map windows.

Legend windows are a special case. Ordinarily, the MapInfo user interface has only one Legend window, just as it has only one Info window. However, the MapBasic language includes a **Create Legend** statement, so that you can create additional Legend windows.

To reparent MapInfo’s standard “one and only” Legend window, use MapBasic’s **Set Window Legend Parent** statement.

To create a custom Legend window and reparent it, use MapBasic’s **Set Next Document** statement, and then use MapBasic’s **Create Legend** statement. Note that in this case, you are creating a Legend that is tied to one specific Map or Graph window. Unlike MapInfo’s default Legend window, such custom Legend windows do not change when the active window changes.

You can make a legend float inside a Map window. In the **Set Next Document** statement, specify the Map window’s HWND as the parent. The legend becomes a frame “trapped” within the Map window. For an example of this technique, see the sample program FindZip.

Allowing the User to Resize a Map Window

Whether the user is able to resize the Map window depends on how you set up your application. The sample program, FindZip, places a Map window in a Visual Basic PictureBox control, so that it cannot be resized. However, you could reparent a Map window using an MDI interface, which allows the user to resize the window.

Note: When the user resizes the Map window, MapInfo does not automatically reset the map’s contents to fill the new window size. Therefore, if your application allows the user to resize the Map window, you must call the Windows API function MoveWindow to make the Map window conform to the new size.

For example, if your Visual Basic program will run under 32-bit Windows, you can use the following Visual Basic **Declare** statement to access the MoveWindow API function:

```
Declare Function MoveWindow Lib "user32" _
    (ByVal hWnd As Long, _
    ByVal x As Long, ByVal y As Long, _
    ByVal nWidth As Long, ByVal nHeight As Long, _
    ByVal bRepaint As Long) As Long
```


When the user resizes the Map window, call `MoveWindow`. In Visual Basic, a resize event triggers the `Form_Resize()` procedure; you could call `MoveWindow` from within that procedure, as shown in the following example.

```
Dim mHwnd As Long
mHwnd = Val(mapinfo.Eval("WindowInfo(FrontWindow(),12)"))
MoveWindow mHwnd, 0, 0, ScaleWidth, ScaleHeight, 0
```

The number 12 corresponds to the MapBasic identifier `WIN_INFO_WND`.

`ScaleWidth` and `ScaleHeight` are properties of a Visual Basic form, representing the form's current width and height.

Note: The `ScaleMode` property must be set to `Pixels`, so that `ScaleWidth` and `ScaleHeight` represent pixel measurements.

Integrating MapInfo Toolbar Buttons

You cannot re-parent MapInfo's `ButtonPads` (toolbars). If you want your client program to have toolbar buttons, you must create the buttons in the language you are using. For example, if you are using Visual Basic, you must create your toolbar buttons using Visual Basic.

If you want a Visual Basic toolbar button to emulate a standard MapInfo button, use MapInfo's `RunMenuCommand` method. (This method has the same effect as the MapBasic **Run Menu Command** statement.) For example, the `FindZip` sample program has an `InfoTool_Click` procedure, which issues the following statement:

```
mapinfo.RunMenuCommand 1707
```

When the user clicks the Visual Basic control, the `FindZip` program calls MapInfo's `RunMenuCommand` method, which activates tool number 1707 (MapInfo's Info tool). As a result of the method call, MapInfo's Info tool becomes the active tool.

The "magic number" 1707 refers to MapInfo's Info tool. Instead of using magic numbers, you can use identifiers that are more self-explanatory. MapBasic defines a standard identifier, `M_TOOLS_PNT_QUERY`, which has a value of 1707. Thus, the following `RunMenuCommand` example has the same effect as the preceding example:

```
mapinfo.RunMenuCommand M_TOOLS_PNT_QUERY
```

Using identifiers (such as `M_TOOLS_PNT_QUERY`) can make your program easier to read. However, if you plan to use identifiers in your code, you must set up your program so that it includes an appropriate MapBasic header file. If you are using Visual Basic, use the header file `MAPBASIC.BAS`. If you are using C, use the header file `MAPBASIC.H`.

The following table lists the ID numbers for each of MapInfo's standard tool buttons. The codes in the third column appear in MAPBASIC.BAS (for Visual Basic), MAPBASIC.H (for C), and MENUS.DEF (for MapBasic).

Main Toolbar Buttons	Number	Identifier Code
Select	1701	M_TOOLS_SELECTOR
Marquee Select	1722	M_TOOLS_SEARCH_RECT
Radius Select	1703	M_TOOLS_SEARCH_RADIUS
Boundary Select	1704	M_TOOLS_SEARCH_BOUNDARY
Zoom In	1705	M_TOOLS_EXPAND
Zoom Out	1706	M_TOOLS_SHRINK
Grabber	1702	M_TOOLS_RECENTER
Info	1707	M_TOOLS_PNT_QUERY
Label	1708	M_TOOLS_LABELER
Ruler	1710	M_TOOLS_RULER
Drag Window	1734	M_TOOLS_DRAGWINDOW
Symbol	1711	M_TOOLS_POINT
Line	1712	M_TOOLS_LINE
Polyline	1713	M_TOOLS_POLYLINE
Arc	1716	M_TOOLS_ARC
Polygon	1714	M_TOOLS_POLYGON
Ellipse	1715	M_TOOLS_ELLIPSE
Rectangle	1717	M_TOOLS_RECTANGLE
RoundedRect	1718	M_TOOLS_ROUNDEDRECT
Text	1709	M_TOOLS_TEXT
Frame	1719	M_TOOLS_FRAME

You also can create custom drawing-tool buttons, which call your program after being used. For a general introduction to the capabilities of custom toolbuttons, see [Chapter 7: Creating the User Interface](#). For details on using custom toolbuttons within an Integrated Mapping application, see the “callbacks” discussion, later in this chapter.

Customizing MapInfo's Shortcut Menus

MapInfo displays a shortcut menu if the user right-clicks on a MapInfo window. These shortcut menus appear even in Integrated Mapping applications. Depending on the nature of your application, you may want to modify or even eliminate MapInfo's shortcut menus. For example, you probably will want to remove the Clone View menu command from the Map window shortcut menu, because cloning a Map window may not work in an Integrated Mapping application.

To remove one or more items from a MapInfo shortcut menu, use MapBasic's **Alter Menu ... Remove** statement, or redefine the menu entirely by using a **Create Menu** statement. For details, see the MapBasic *Reference* or online Help.

To add custom items to a MapInfo shortcut menu, use MapBasic's **Alter Menu ... Add** statement, and specify the **Calling OLE** or **Calling DDE** syntax; see the "callbacks" discussion later in this chapter.

To eliminate a shortcut menu entirely, use the MapBasic statement **Create Menu** to redefine the menu, and use the control code "(" as the new menu definition. For example, the following statement destroys MapInfo's shortcut menu for Map windows:

```
mapinfo.do "Create Menu ""MapperShortcut"" ID 17 As "(" " "
```

Printing an Integrated MapInfo Window

You can use MapBasic's **PrintWin** statement to print a MapInfo window, even a reparented window. For example, see the FindZip sample program. The FindZip program's File menu contains a Print Map command. If the user chooses Print Map, the program executes the following procedure:

```
Private Sub Menu_PrintMap_Click()  
    mapinfo.do "PrintWin"  
End Sub
```

MapBasic's **PrintWin** statement prints the map on a single page, with nothing else on the page.

You also can use MapBasic's **Save Window** statement to output a Windows metafile (WMF file) representation of the Map window. For example, see the FindZip sample program: If the user chooses Print Form, the program creates a metafile of the map, attaches the metafile to the form, and then uses Visual Basic's **PrintForm** method. The end result is a printout of the form which includes the metafile of the map.

Detecting runtime Errors

When your client program sends MapInfo a command string, the command might fail. For example, the command "Map From World" fails if the World table is not open. MapInfo generates an error code if the command fails.

To trap a MapInfo error, set up error trapping just as you would for any other OLE Automation process. In Visual Basic, for example, use the **On Error** statement to enable error-trapping.

To determine which error occurred in MapInfo, read MapInfo's OLE Automation properties `LastErrorCode` and `LastErrorMessage`. For details on these properties, see *OLE Automation Object Model*, later in this chapter. For a listing of MapBasic's error codes, see the text file ERRORS.DOC.

Note: The `LastErrorCode` property returns values that are 1000 greater than the error numbers listed in `ERRORS.DOC`. In other words, if an error condition would cause a compiled MapBasic application to produce a runtime error 311, the same error condition would cause an Integrated Mapping application to set the `LastErrorCode` property to 1311.

When you run a MapBasic application (MBX file) via Automation, the MBX will not trap its own runtime errors. You can run an MBX by using the `Do` method to issue a MapBasic **Run Application** statement. However, if a MapBasic runtime error occurs within the MBX, the MBX will halt, even if the MBX uses the MapBasic **OnError** statement. If you are building an MBX which you will call via Automation, try to keep the MBX simple. Within the MBX, avoid using MapBasic's **OnError** statement; instead, do as much error checking and prevention as possible in the controlling application before running the MBX.

Terminating MapInfo

If you create a new instance of MapInfo by calling Visual Basic's **CreateObject()** function, that instance of MapInfo terminates automatically when you release its Object variable. If the Object variable is local, it is released automatically when you exit the local procedure. To release a global Object variable, assign it a value of `Nothing`:

```
Set mapinfo = Nothing
```

If you use DDE to communicate with MapInfo, you can shut MapInfo down by using the **LinkExecute** method to send an "End MapInfo" command string.

Terminating Your Visual Basic Program

If you are creating a 16-bit Visual Basic program that uses DDE to communicate with MapInfo, make sure you terminate your DDE links before you exit your Visual Basic program. If you exit your Visual Basic program while DDE links are still active, you may experience undesirable behavior, including runtime error messages. This problem occurs when you run 16-bit Visual Basic programs under a 32-bit version of Windows (Windows 95 or Windows NT). To avoid this problem, set up your Visual Basic program so that it terminates its DDE links before it exits.

A Note About MapBasic Command Strings

As shown in the preceding pages, you can create strings that represent MapBasic statements, and then send the strings to MapInfo by using the `Do OLE Method`. Note that you can combine two or more statements into a single command string, as the following Visual Basic example illustrates. (In Visual Basic, the `&` character performs string concatenation.)

```
Dim msg As String

msg="Open Table ""States"" Interactive "
msg=msg & "Set Next Document Parent " & Frm.hWnd & " Style 1 "
msg=msg & "Map From States "

mapinfo.do msg
```

When parsing the command string at run time, MapInfo automatically detects that the string contains three distinct MapBasic statements: An **Open Table** statement, a **Set Next Document** statement, and a **Map From** statement. MapInfo is able to detect the distinct statements because **Open**, **Set**, and **Map** are reserved keywords in the MapBasic language.

Note the space after the keyword **Interactive**. That space is necessary; without the space, the command string would include the substring “InteractiveSet” which is not valid MapBasic syntax. Because each command string ends with a space, MapInfo can detect that **Interactive** and **Set** are separate keywords.

If you combine multiple MapBasic statements into a single command string, make sure you include a space after each statement, so that MapInfo can detect that the string contains separate statements.

A Note About Dialog Boxes

In an Integrated Mapping application, the control OKButton will be ineffective in dismissing the dialog. Use a regular push-button control and set a variable to determine if the user has clicked that button.

A Note About Accelerator Keys

In an Integrated Mapping application, MapInfo’s accelerator keys (for example, Ctrl-C to copy) are ignored. If you want your application to provide accelerator keys, you must define those accelerators within your client program (for example, your Visual Basic application).

However, Integrated Mapping applications do support pressing the S key to toggle Snap To Node on or off.

Using Callbacks to Retrieve Info from MapInfo

You can set up your Integrated Mapping application so that MapInfo automatically sends information to your client program. For example, you can set up your program so that whenever a Map window changes, MapInfo calls your client program to communicate the Integer window ID of the window that changed. This type of notification, where an event causes MapInfo to call your client program, is known as a *callback*.

- Callbacks allow MapInfo to send information to your client program under the following circumstances:
- **The user interacts with a MapInfo window while using a custom tool.** For example, if the user clicks and drags on a Map window to draw a line, MapInfo can call your client program to communicate the x- and y-coordinates chosen by the user.
- **The user chooses a menu command.** For example, suppose your application customizes MapInfo’s shortcut menus (the menus that appear if the user right-clicks). When the user chooses a custom command from a shortcut menu, MapInfo can call your client program to notify your program of the menu event.
- **A Map window changes.** If the user changes the contents of a Map window (for example, by adding or removing map layers, or by panning the map), MapInfo can send your client program the Integer window ID of the window that changed. (This is analogous to MapBasic’s special handler procedure, WinChangedHandler.)
- **The status bar text changes in MapInfo.** MapInfo’s status bar does not appear automatically in Integrated Mapping applications. If you want your client program to emulate MapInfo’s status bar, you must set up your application so that MapInfo notifies your client program whenever the status bar text changes.

Technical Requirements for Callbacks

If you plan to use callbacks, your client program must be able to act as a DDE server or as an OLE Automation server. Visual Basic 4.0 Professional Edition and C++ can create applications that are Automation servers. However, applications written using Visual Basic 3.0 cannot act as Automation servers, so they must use DDE to handle callbacks.

General Procedure for Using OLE Callbacks

The following steps provide an overview of the process of using callbacks through OLE:

1. Using Visual Basic 4.0, C++, or any other language that can act as an OLE server, create a class definition that defines one or more OLE methods. For details on how to create a class definition, see the documentation for your programming language.
2. If you want to emulate MapInfo's status bar, create a method called `SetStatusText`. Define this method so that it takes one argument: a string.
3. If you want MapInfo to notify your program each time a Map window changes, create a method called `WindowContentsChanged`. Define this method so that it takes one argument: a four-byte integer.
4. If you want MapInfo to notify your client program whenever custom menu commands or custom buttons are used, create one or more additional method(s), using whatever method names you choose. Each of these methods should take one argument: a string.
5. Create an object using your custom class. For example, if you called the class "CMyClass", the following Visual Basic statement creates an object of that class:

```
Public myObject As New CMyClass
```

6. After your program launches MapInfo, call MapInfo's `SetCallback` method, and specify the name of the object:

```
mapinfo.SetCallback myObject
```

If you want MapInfo to notify your client program when the user uses a custom toolbar button, define a custom button (for example, send MapInfo an **Alter ButtonPad ... Add** statement). Define the custom button so that it uses the syntax **Calling OLE *methodname*** (using the method name you created in step 4).

MapInfo's toolbars are hidden, like the rest of MapInfo's user interface. The user will not see the new custom button. Therefore, you may want to add an icon, button, or other visible control to your client program's user interface. When the user clicks on your Visual Basic icon or button, send MapInfo a **Run Menu Command ID** statement so that your custom toolbar button becomes the "active" MapInfo tool.

7. If you want MapInfo to notify your client program whenever the user uses a custom menu command, define a custom menu command (for example, using the **Alter Menu ... Add** statement to add an item to one of MapInfo's shortcut menus). Define the custom menu command so that it uses the syntax **Calling OLE *methodname*** (using the method name you specified in step 4).
8. Within the method(s) that you defined, issue whatever statements are needed to process the arguments sent by MapInfo.
9. If you created a `SetStatusText` method, MapInfo sends a simple text string to the method, representing the text that MapInfo would display on the status bar. If you want to emulate MapInfo's status bar, add code to this method to display the text somewhere in your user interface.

If you created a `WindowContentsChanged` method, MapInfo sends a four-byte integer (representing a MapInfo window ID number) to indicate which Map window has changed. Add code to this method to do whatever processing is necessary in response to the window's changing. For example, if you are keeping track of the Map window's current zoom level, you may want to call MapInfo's **MapperInfo()** function to determine the Map window's latest zoom level.

If you are using methods to handle custom buttons or menu commands, MapInfo sends a comma-delimited string to your custom method. Within your method, parse the string. The exact format of the string varies, depending on whether the user used a menu command, a point-mode drawing tool, a line-mode drawing tool, etc. The following section explains the syntax of the comma-separated string.

Processing the Data Sent to a Callback

Your Integrated Mapping application can create custom MapInfo menu commands and custom MapInfo toolbar buttons. When the user uses the custom commands or buttons, MapInfo sends your OLE method a string containing eight elements, separated by commas. For example, the string sent by MapInfo might look like this:

```
MI:-73.5548,42.122,F,F,-72.867702,43.025,202,
```

The contents of the comma-separated string are easier to understand if you are already familiar with MapBasic's **CommandInfo()** function. When you write MBX applications (i.e., programs written in the MapBasic language and compiled with the MapBasic compiler), you can have your custom menu commands and custom buttons call MapBasic handler procedures instead of calling OLE methods. Within a handler procedure, you can call **CommandInfo()** to determine various information about recent events. For example, if a MapBasic procedure acts as the handler for a custom drawing-tool button, the following function call determines whether the user held down the Shift key while using the drawing tool:

```
log_variable = CommandInfo(CMD_INFO_SHIFT)
```

The code `CMD_INFO_SHIFT` is defined in the MapBasic header file, `MAPBASIC.DEF`. The following table lists `CommandInfo`-related defines, sorted in order of their numeric values.

Value	Codes That Have Meaning After a Menu Event	Codes That Have Meaning After a Button Event
1		CMD_INFO_X
2		CMD_INFO_Y
3		CMD_INFO_SHIFT
4		CMD_INFO_CTRL
5		CMD_INFO_X2
6		CMD_INFO_Y2
7		CMD_INFO_TOOLBTN
8	CMD_INFO_MENUITEM	

For an explanation of each code, see **CommandInfo()** in the *MapBasic Reference* or online Help.

When you create a custom menu command or button that uses the **Calling OLE *methodname*** syntax, MapInfo constructs a string with all eight **CommandInfo()** return values, separated by commas. The string begins with the prefix MI: so that your OLE server can determine that the method call was made by MapInfo.

The string that MapInfo sends to your method is constructed in the following manner:

```
"MI:" +
CommandInfo(1) + "," + CommandInfo(2) + "," +
CommandInfo(3) + "," + CommandInfo(4) + "," +
CommandInfo(5) + "," + CommandInfo(6) + "," +
CommandInfo(7) + "," + CommandInfo(8)
```

If you assign a unique ID number to each of your custom buttons, you can have all of your buttons call the same method. Your method can determine which button called it by examining the seventh argument in the comma-separated string.

Once MapInfo sends the comma-separated string to your method, it is up to you to add code to your method to parse the string.

Suppose your Integrated Mapping application adds a custom menu command to the MapInfo shortcut menu. Every time the user chooses that custom menu command, MapInfo sends your OLE method a comma-separated string. If the custom menu command has an ID number of 101, the string might look like this:

```
"MI:,,,,,,101"
```

In this case, most of the elements of the comma-separated string are empty, because the **CommandInfo()** function can only return one piece of information about menu events (as is indicated in the table above). Of the eight "slots" in the string, only slot number eight pertains to menu events.

Now suppose you create a custom MapInfo toolbar button that allows the user to click and drag to draw lines on a map. Every time the user uses that custom drawing tool, MapInfo sends your OLE method a comma-separated string, which might look like this:

```
"MI:-73.5548,42.122,F,F,-72.867702,43.025,202,"
```

In this case, the comma-separated string contains several values, because **CommandInfo()** is able to return several pieces of relevant information about toolbar events. The first two elements indicate the x- and y-coordinates of the location where the user clicked; the next two elements indicate whether the user held the Shift and Ctrl keys while clicking; the next two elements indicate the coordinates of the location where the user released the mouse button; and the last element indicates the button's ID number. The final "slot" in the string is empty, because slot number eight pertains to menu events, not button events.

C/C++ Syntax for Standard Notification Callbacks

The preceding section discussed callbacks in the context of Visual Basic. This section identifies the specific C-language syntax for MapInfo's standard callbacks, `SetStatusText` and `WindowContentsChanged`.

If you use MapInfo's SetCallback method, MapInfo can automatically generate notification callbacks to your IDispatch object. MapInfo's standard callbacks have the following C syntax:

```
SCODE SetStatusText(LPCTSTR lpszMessage)
```

MapInfo calls the SetStatusText method whenever the status bar text changes in MapInfo. The single argument is the string value of the new status bar text.

```
SCODE WindowContentsChanged(Unsigned Long windowID)
```

MapInfo calls the WindowContentsChanged method whenever the contents of a reparented Map window change. The single argument represents MapInfo's Integer window ID that identifies which window changed. This callback is analogous to MapBasic's WinChangedHandler procedure.

Alternatives to Using OLE Callbacks

As discussed earlier, MapInfo callbacks can use OLE to send information to your client program. In some cases, however, you may need to set up callbacks that do not use OLE. For example, if you are developing programs in Visual Basic 3.0, you cannot use OLE for your callbacks, because Visual Basic 3.0 does not allow you to create your own OLE Automation servers.

MapInfo supports two types of callbacks that are not OLE-dependent: Callbacks using DDE, and callbacks using compiled MapBasic applications (MBX files).

DDE Callbacks

When you create custom toolbar buttons or menu commands, you specify a **Calling** clause. To handle the callback through DDE, use the syntax **Calling DDE server, topic**. Whenever the user uses the custom button or menu command, MapInfo opens a DDE connection to the DDE server that you designate, and then sends a string to the DDE topic that you designate. The string uses the format discussed in the previous section (for example, "MI:, , , , , 101").

For an example of a DDE callback, see the sample program FindZip. The Form Load procedure sends MapInfo an **Alter ButtonPad ... Add** statement to create a custom toolbar button.

The new toolbar definition includes the following calling clause:

```
Calling DDE "FindZip", "MainForm"
```

Whenever the user clicks on the map using the custom tool, MapInfo opens a DDE connection to the FindZip application, and then sends a string to the "MainForm" topic. ("MainForm" is the value of the form's LinkTopic property.) For an introduction to DDE, see [Chapter 4: Using the Development Environment](#).

MBX Callbacks

If you create a compiled MapBasic application (MBX file), then you can set up your custom buttons and menu commands so that they call MapBasic procedures in the MBX. In the calling clause, use the syntax **Calling procedure** (where *procedure* is the name of a procedure in the MapBasic program). After your Visual Basic application launches MapInfo, run your MBX by sending MapInfo a **Run Application** statement. For example:

```
mapinfo.do "Run Application " "C:\MB\MYAPP.MBX" " "
```

For an introduction to creating custom buttons and menu commands, see [Chapter 7: Creating the User Interface](#).

Online Help

An Integrated Mapping application can invoke MapInfo dialog boxes by using MapInfo's **RunMenuCommand** OLE Method. If your application invokes a MapInfo dialog box, you can control whether online help is available for the dialog box.

Displaying Standard MapInfo Help

You can allow your users to see the standard MapInfo help on the dialog box. This is the default behavior. If the user presses F1 while a MapInfo dialog box is displayed, Windows help displays an appropriate topic from MAPINFOW.HLP (the standard MapInfo help file).

Note: Once the MapInfo help window appears, the user can click various jumps or navigation buttons to browse the rest of the help file. Users may find this arrangement confusing, because the MapInfo help file describes the MapInfo user interface, not the user interface of your Integrated Mapping application.

Disabling Online Help

You can disable all online help for MapInfo dialog boxes by issuing the following MapBasic statement:

```
Set Window Help Off
```

After you issue a **Set Window Help Off** statement, pressing F1 while on a MapInfo dialog box has no effect.

Displaying a Custom Help File

You can set MapInfo to use a custom help file. For example, the following MapBasic statement instructs MapInfo to use the help file CUSTOM.HLP instead of MAPINFOW.HLP:

```
Set Window Help File "CUSTOM.HLP" Permanent
```

After you issue a **Set Window Help File...Permanent** statement, pressing the **F1** key causes MapInfo to display online help; however, MapInfo displays the help file that you specify instead of MAPINFOW.HLP. Use this arrangement if you want to provide online Help for one or more MapInfo dialogs, but you do not want the user to have access to all of the standard MapInfo help file.

If you want to provide custom help for MapInfo dialog boxes, you must set up your custom help file so that its Context ID numbers match MapInfo's dialog box IDs.

To determine the ID number of a MapInfo dialog box:

1. Run MapInfo with the **-helpdiag** command-line argument.
2. Display the MapInfo dialog for which you want to create help.
3. Press F1. Because you used the **-helpdiag** option, MapInfo displays the dialog's ID number instead of displaying help. Make note of the dialog's ID number.
4. Using your Windows help-authoring software, edit your custom help file, so that your custom help topic is assigned the same ID number as the MapInfo dialog box.

For example, MapInfo's Find dialog box has the ID number 2202. If you want to provide your own online help for the Find dialog box, set up your help file so that your custom help topic has the Context ID number 2202.

Note the following points:

- MapBasic does not include a Windows help compiler.
- MapInfo's dialog box ID numbers are likely to change in future versions.

Related MapBasic Statements and Functions

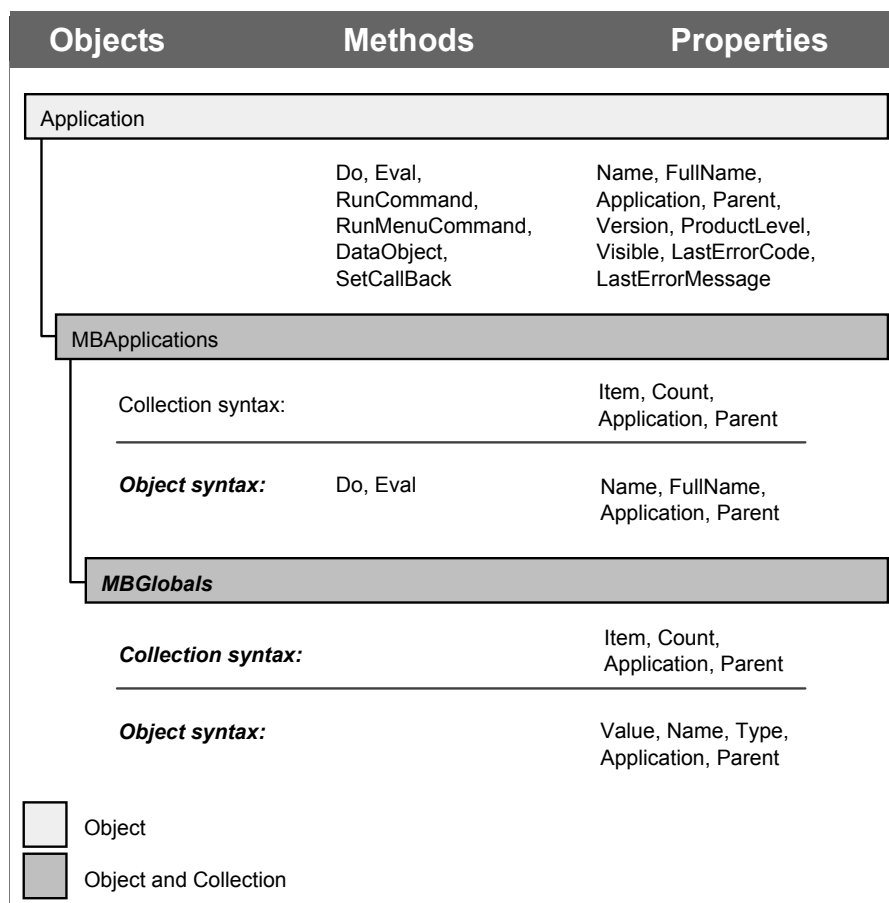
This section lists some of the MapBasic statements and functions that are particularly useful in Integrated Mapping applications. For details on these statements and functions, see the *MapBasic Reference* or online Help.

Statement/Function Name	Description
Create Legend	Creates a new Legend window.
Map	Creates a new Map window.
MenuItemInfoByID() MenuItemInfoByHandler(),	Determines the status of a MapInfo menu command (for example, checked or not checked).
Open Table	Opens MapInfo tables.
RemoteQueryHandler()	Allows MapBasic programs to handle <i>peek</i> requests from DDE clients.
Run Menu Command	Simulates user selecting a MapInfo menu command or ButtonPad button.
SearchInfo()	Returns information about the results obtained by SearchPoint() and SearchRect() .
SearchPoint(), SearchRect()	Searches the selectable layers of a Map window for objects at a specific x,y location or objects within a rectangular area. Allows you to emulate MapInfo's Info tool or Label tool.
Set Application Window	Reparents dialog box windows. Issue this statement once in your client program, after you have connected to or launched MapInfo.
Set Map	Controls many aspects of Map windows.

Set Next Document	Reparents a document window, such as a Map window, to be a child window of your client program.
Set Window	Controls various aspects of MapInfo windows.
Shade, Set Shade	Creates or modifies thematic map layers.
SystemInfo()	Some values returned by SystemInfo() are specific to Integrated Mapping. Example: Specify SYS_INFO_APPLICATIONWND to retrieve the application's HWND.
WindowID(), WindowInfo()	Return info about MapInfo windows, even reparented windows.

OLE Automation Object Models

The following chart provides an overview of MapInfo's 6.5 OLE Automation Type Library. Methods and Properties are described in detail on the following pages.

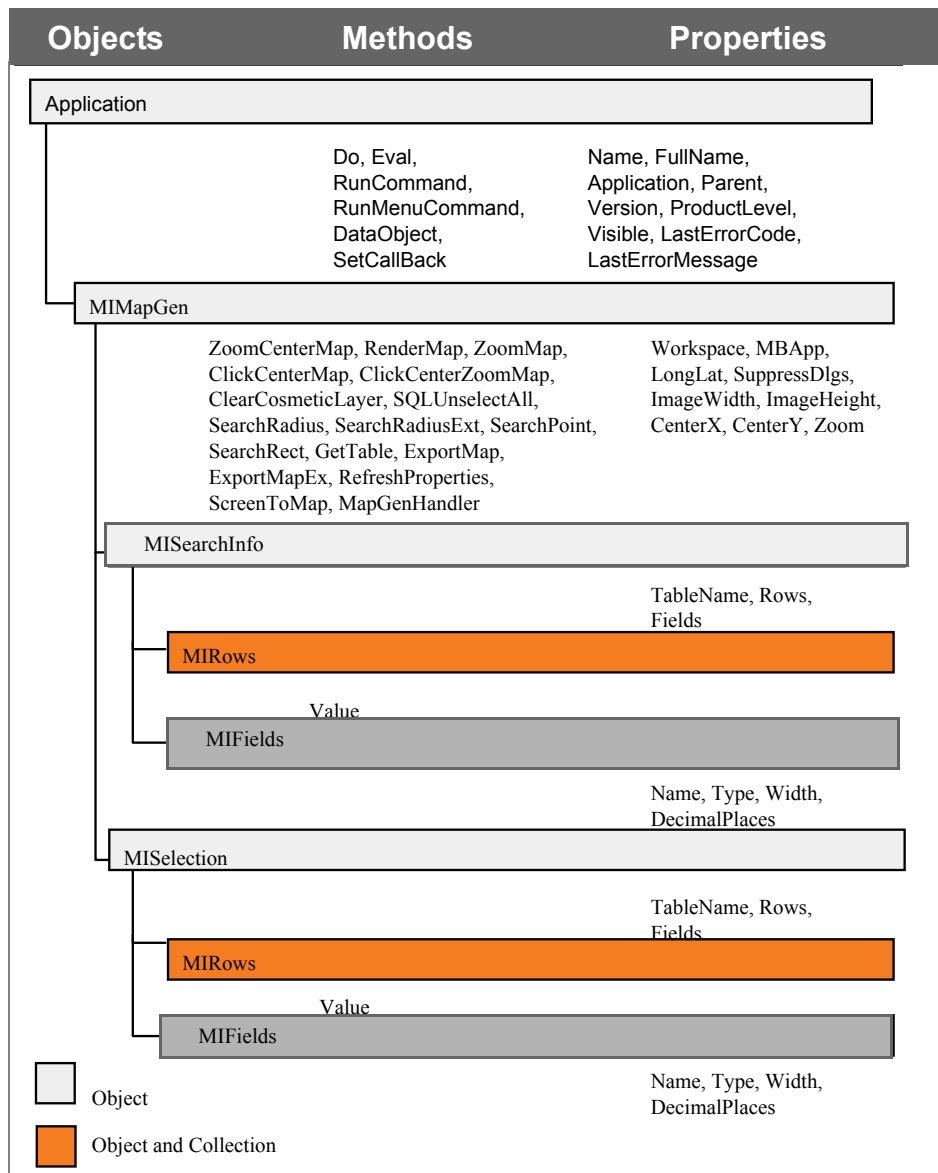


The **Application** object represents the instance of MapInfo.

Each object in the **MBAplications** collection represents a MapBasic application that is currently running.

Each object in the **MBGlobals** collection represents a global variable defined by one of the running MapBasic applications.

The following chart provides additional objects available in MapInfo's 6.5 OLE Automation Type Library. Methods and Properties are described in detail on the following pages.



Properties of the *Application* Object

The following table lists all of the properties that apply to the Application object. All properties in this table are read-only, except for Visible and LastErrorCode.

Property Name	Functionality
Name	Returns application name ("MapInfo Professional"). OLE standard property. This is the default property for the Application object.
FullName	Returns full path to application executable. OLE standard property.
Application	Returns the Application object. OLE standard property.
Parent	Returns the Application object of its parent object; for an Application object, returns itself. OLE standard property.
Version	Returns text of current version number, multiplied by 100 (for example, MapInfo 4.0.0 returns "400").
ProductLevel	Returns integer, indicating which MapInfo product is running. For MapInfo Professional, returns 200.
Visible	Read/write: A boolean value, indicating whether application window is visible. Read the property to determine the window's visibility; write the property to set the window's visibility.
LastErrorCode	Read/write: A small integer value giving the code number of the last MapBasic error that occurred during a Do, Eval, or RunCommand method call. Note: The code numbers returned here are 1000 higher than the corresponding MapBasic error code numbers. Error codes are never automatically cleared to zero; once an error occurs, the error remains until another error occurs (or until you write a new value to the property).
LastErrorMessage	Returns a string: The text of the error message that corresponds to LastErrorCode.

Methods of the *Application* Object

Method Name	Functionality
Do(string)	Interprets a string as a MapBasic statement, and executes the statement. This method is asynchronous.
Eval(string)	Interprets a string as a MapBasic expression, and returns the value of the expression; returns a string. If the expression has a Logical value, MapInfo Professional returns a one-character string, "T" or "F". This method is synchronous.
RunCommand(string)	Interprets a string as a MapBasic statement; this is a synonym for "Do."
RunMenuCommand(menuid)	Executes the menu command indicated by the Integer menuid argument. See example below. This method activates a standard menu command or button; to activate a custom menu command or button, use the Do method to issue a Run Menu Command ID statement.
DataObject(windowID)	Given an integer windowID, returns an IUnknown interface representing that window. To get a metafile representation of the window, use QueryInterface for an IDataObject interface. IDataObject and IUnknown are the only two interfaces defined for this object. Note: This is an advanced feature, intended for C programmers.
SetCallBack(IDispatch)	Registers the OLE Automation object as a "sink" for MapInfo-generated notifications. Only one callback function can be registered at a time. See Notification Callbacks, earlier in this chapter.

For example, the following statement uses the Do method to send MapInfo a **Map** statement:

```
mapinfo.Do "Map From World"
```

The following statement uses the RunMenuCommand method to execute MapInfo's menu command code 1702, which selects MapInfo's Grabber tool. (To determine a specific numeric value for a menu code, look in MENU.DEF, or see the table earlier in this chapter.)

```
mapinfo.RunMenuCommand 1702
```

Properties of the *MBApplications* Collection

MBApplications is a collection of all the MapBasic applications that MapInfo is currently running. The properties in the following table are all read-only.

Property Name	Functionality
Item	Returns IDispatch of a particular <i>programobject</i> object. Argument is a VARIANT type which can evaluate to an integer index (1..Count) or a string value (name of the program). This is the default property for the MBApplications collection.
Count	Returns the long integer number of objects in the collection (i.e., the number of running applications).
Application	Returns IDispatch of the MapInfo application object. OLE standard property.
Parent	Returns IDispatch of its parent object; for this collection, that's the MapInfo application object. OLE standard property.

Properties of an Object in *MBApplications*

Each object in the MBApplications collection is a running MapBasic application. The properties in the following table are all read-only.

Property Name	Functionality
Name	Returns name of application (for example, "FOO.MBX"). OLE standard property. This is the default property for an MBApplication object.
FullName	Returns full path to MapBasic application .MBX file. OLE standard property.
Application	Returns IDispatch of the application. OLE standard property.
Property Name	Functionality
Parent	Returns IDispatch of its parent object; for a <i>programobject</i> , that's the MapInfo application object. OLE standard property.

For example, the following statements determine the name of a running MapBasic application:

```
Dim appsList As Object
Dim firstname As String

Set appsList = mapinfo.MBApplications
If appsList.Count > 0 Then
    firstname = appsList(1).Name
End If
```


Methods of an Object in *MBApplications*

Method Name	Functionality
Do(<i>string</i>)	The specified string is sent to the MapBasic application's RemoteMsgHandler procedure.
Eval(<i>string</i>)	The specified string is sent to the MapBasic application's RemoteQueryHandler () function; the value returned by RemoteQueryHandler is returned. RemoteQueryHandler () must be defined as a function that returns a string. If the expression has a Logical value, MapInfo returns a one-character string, "T" or "F".

Properties of the *MBGlobals* Collection

MBGlobals is a collection of all the MapBasic global variables declared by a specific MapBasic application that is running. The properties in the following table are all read-only.

Property Name	Functionality
Item	Returns IDispatch of a particular <i>mbglobal</i> object. Argument is a VARIANT type which can evaluate to an integer index (1..Count) or a string value (name of the global variable). This is the default property for the MBGlobals collection.
Count	Returns the long integer number of objects in the collection (the number of global variables).
Application	Returns IDispatch of MapInfo application object. OLE standard property.
Parent	Returns IDispatch of its parent object; for this collection, that's the <i>programobject</i> object. OLE standard property.

Properties of an Object in *MBGlobals*

Each object in the MBGlobals collection is a MapBasic global variable. The properties in the following table are all read-only, except for the Value property.

Property Name	Functionality
Value	Read/write. Read the property to retrieve a string representing the value of the MapBasic global variable; write the property to change the value of the variable. This is the default property for an MBGlobal object.
Name	Returns name of the variable. OLE standard property.
Type	Returns a text string giving the type of the variable as one of MapInfo Professional's standard types ("Integer", "Date", etc.).
Application	Returns IDispatch of the application. OLE standard property.
Parent	Returns IDispatch of its parent object; for an <i>MBglobal</i> object, that's the <i>programobject</i> which declared the global variable. OLE standard property.

The following Visual Basic example examines and then alters the value of a global variable (g_status) in a MapBasic application.

```
Dim globinfo As Object
Dim old_value As Integer

' Look at the globals used by the first
' running MapBasic app:
Set globinfo = mapinfo.MBApplications(1).MBGlobals

' Look at a global's current value by reading
' its "Value" property:
old_value = globinfo("g_status").Value

' Assign a new value to the global:
globinfo("g_status") = old_value + 1
```

The expression **globinfo("g_status")** is equivalent to **globinfo("g_status").Value** because Value is the default property.

Properties of the *MIMapGen* Object

The following table lists the properties that apply to the MIMapGen object. The MIMapGen object is used primarily by MapInfo ProServer applications; however, MapInfo Professional applications can use the MIMapGen object as well. For examples of using the MIMapGen object model, see the ProServer documentation.

Property Name	Functionality
Workspace	Path to a MapInfo workspace file. When you set the property, MapInfo loads the workspace.
MBAApp	Path that points to a MapBasic application (MBX file). When you set the property, MapInfo runs the MBX.
LongLat	BOOLEAN: Defines interface coordinate system. When TRUE, all values that you get and put (using CenterX and CenterY) represent longitude and latitude. When FALSE, the map window's coordinate system will be used.
SuppressDlg	BOOLEAN: If TRUE, an action that invokes a dialog will generate an error. This includes dialogs invoked as a result of a Run Menu Command statement.
ImageWidth	Width of the image area, in pixels.
ImageHeight	Height of the image area, in pixels.
CenterX	X-coordinate (for example, Longitude) of the map center.
CenterY	Y-coordinate (for example, Latitude) of the map center.
Zoom	The width of the map (for example, number of miles across). This number reflects the units used by the Map window (for example, miles, kilometers).

Setting the Workspace property is the first step to using the MIMapGen object. MIMapGen is designed to work in situations where there is a single map window (for example, when a web page shows a single map). To begin using MIMapGen, set the Workspace property, so that MapInfo loads a workspace -- typically, a workspace that contains a single Map window. Then you will be able to use the other methods and properties to manipulate the Map window.

Methods of the *MIMapGen* Object

The following methods apply to the MIMapGen object.

Method	Functionality
ZoomCenterMap()	Renders the map based on the current CenterX, CenterY, and Zoom properties. The map is only regenerated if the center or the zoom have changed since the map was last rendered.
RenderMap()	Same effect as ZoomCenterMap, except that the map is always regenerated.
ZoomMap (double ZoomFactor)	Zooms the map in or out, to the extent indicated by the zoom factor. Positive numbers zoom in; negative numbers zoom out.
ClickCenterMap (long MouseX, long MouseY)	Recenters the map based on the mouse click position. The x/y arguments represent locations on the map, in pixels.
ClickCenterZoomMap (long MouseX, long MouseY, double ZoomFactor)	Recenters the map based on the mouse click position, and zooms the map based on the zoom factor; negative number zooms out.
ClearCosmeticLayer()	Same effect as the Map menu command: Deletes all objects from the Cosmetic layer.
SQLUnselectAll()	Same effect as the Query menu command: De-selects all rows.
SearchRadius (double CenterPointX, double CenterPointY, double Radius)	Performs a radius search.
SearchRadiusExt (double CenterPointX, double CenterPointY, double OuterPointX, double OuterPointY)	Performs a radius search, To define the search circle, specify the center point and a point that is somewhere along the circle's radius.
SearchPoint (double CenterPointX, double CenterPointY)	Searches a small area around the specified location.
SearchRect (double x1, double y1, double x2, double y2)	Searches within a rectangular area.
GetTable (string Tablename)	Returns an MISelection object (IDispatch); to access the contents of the table, use the MISelection object.

Method	Functionality
ExportMap (string ImageType, string FileSpec)	Generates an image file (for example, a JPEG, TIFF, PNG, PSD, BMP, WMF, or GIF file) of the Map window. See the MapBasic Save Window statement.
ExportMapEx (string ImageType, string FileSpec, string CopyrightInfo)	Generates an image file (for example, a JPEG, TIFF, PNG, PSD, BMP, WMF, or GIF file) of the Map window. See the MapBasic Save Window statement.
RefreshProperties()	Updates CenterX, CenterY, Zoom, ImageHeight, and ImageWidth.
ScreenToMap (long ScreenX, long ScreenY, double MapX, double MapY)	Converts screen coordinates (pixels) into map coordinates (for example, longitude / latitude).
MapGenHandler (string Message)	Calls the MapBasic sub procedure RemoteMapGenHandler in the MBX application that was executed through the MApp property. Use this method to run MapBasic statements in an MBX file.

Tip: The searching methods search only the topmost selectable layer. To access the search results, see the MISearchInfo object.

Properties of the *MISearchInfo* Object

The following properties apply to the MISearchInfo object.

Property	Functionality
Rows	This property returns an MIRows collection (a collection of MIRow objects). The collection represents the search results.
Fields	This property returns an MIFields collection (a collection of MIField objects). The collection represents a set of field definitions (field names, etc.) describing the search results.
TableName	String: The name of the table that contains the search results.

To obtain an MISearchInfo object, use one of the MIMapGen object's search methods: SearchRadius, SearchRadiusExt, SearchPoint, or SearchRect.

Methods of the *MIRow* Object

The following method applies to the MIRow object. Each MIRow object represents one record returned by a search method, or one row in the table specified in the GetTable method call.

Method	Functionality
Value	Returns a pointer to the data value for the given column specified by using a variant arg. The allowed variant types are VT_12, VT_14, and VT_BSTR (where the VT_BSTR is the column name).

Tip: To obtain a collection of MIRow objects, reference the Rows property of the MISearchInfo object or the MISelection object.

Properties of the *MIField* Object

The following properties apply to the MIField object. Each MIField object describes one of the data columns in the latest search results, or one of the data columns in the table specified in the GetTable method call.

Property	Functionality
Name	String: The name of the column.
Type	Short: The data type of the field. The following values are valid: (1) DT_CHAR, (2) DT_DECIMAL, (3) DT_INTEGER, (4) DT_SMALLINT, (5) DT_TIME, (6) DT_LOGICAL, (8) DT_FLOAT.
Width	Short: The width of the field; applies to DT_CHAR and DT_DECIMAL fields only.
DecimalPlaces	Short: The number of decimal places in a DT_DECIMAL field.

Tip: To obtain a collection of MIField objects, reference the Fields property of the MISearchInfo object or the MISelection object.

Properties of the *MISelection* Object

The following properties apply to the MISelection object.

Property	Functionality
Rows	This property returns an MIRows collection (a collection of MIRow objects). The collection represents all of the rows in a table.
Fields	This property returns an MIFields collection (a collection of MIField objects). The collection represents the field definitions (field names, etc.) for the table that was specified in the GetTable method.
TableName	String: The name of the table that was specified in the GetTable method.

To access the MISelection object, use the GetTable method from the MIMapGen object.

MapInfo Command-Line Arguments

If you use DDE to communicate with MapInfo, you will need to launch MapInfo manually (for example, by calling Visual Basic's **Shell()** function) before you establish the DDE connection. When you launch MapInfo 4.0 for Windows, you can use any of the command-line arguments listed below. If you want the user to remain unaware that MapInfo is running, you will want to specify one of the following arguments.

Command-Line Argument	Effect
-nosplash	MapInfo runs without showing its splash screen, although the main MapInfo window still shows.
-server	MapInfo runs without showing a splash screen or main window. Use this argument when you want MapInfo to act as a behind-the-scenes server to another application (using DDE).
-automation or -embedding	MapInfo runs without displaying a splash screen or main window. Additionally, MapInfo registers its OLE Class Factory with the OLE subsystem, which allows MapInfo to act as a behind-the-scenes OLE server to another application.
-regserver	MapInfo registers its OLE capabilities in the registration database, then exits. Run MapInfo with this argument once, when you install MapInfo. Note that MapInfo automatically registers itself when it is run normally. Note very well that this registers everything about the MapInfo product ' OLE Automation, OLE Embedding, etc.
-unregserver	MapInfo removes all references to itself from the registration database and exits. Use this option at uninstall time to remove MapInfo from the system registry. Using this argument unregisters everything that the -regserver option registered.
-helpdiag	This argument sets a flag in MapInfo, so that MapInfo displays a diagnostic dialog every time you press F1 for online Help. For more information on Help issues, see the discussion earlier in this chapter.

Note: The forward slash ("/") can be used instead of the minus sign.

Getting Started with Integrated Mapping and Visual C++ with MFC

The remainder of this chapter will walk you through the creation of an Integrated Mapping application using Microsoft Visual C++ with MFC. These instructions are written primarily for users of the 32-bit Visual C++ (version 2.0 or higher), but they have also been tested with the 16-bit version of Visual C++ (version 1.52). Differences are noted where appropriate.

Create a New Project

1. Run Visual C++ 2.x (32-bit) or 1.5x (16-bit).
2. Choose File > New to create a new project (Project > AppWizard... in v1.5).
3. Make the project an MFC AppWizard application, and choose the options that you want. For your first demonstration, it's easiest to make the application a single document application (SDI), rather than supporting multiple documents (MDI). Note well that you're not required to enable any of the standard OLE support. If you want to use callbacks to your application from MapInfo, you should enable support for OLE Automation in Step 3 of 6 of the MFC AppWizard.
4. Build the application and run it to verify that everything starts out ok.

Add OLE Automation Client Support

If you did not choose any OLE support during the AppWizard phase, you must add OLE Automation client support now.

1. Open STDAFX.H and add these lines:

```
#include <afxole.h>
#include <afxdisp.h>
```

2. Open your main program source file (i.e., *projectname.CPP*) and add the following lines to the beginning of *CprojectnameApp::InitInstance*:

```
if (!AfxOleInit()) {
    AfxMessageBox(IDP_OLE_INIT_FAILED);
    return FALSE;
}
```

3. Add the message string by opening your resource file (i.e., *projectname.RC*), open the “String Table” resource, and pick Resource > New String (in v1.5, you need to use AppStudio to add the string). In the properties dialog box that appears, set ID: to “IDP_OLE_INIT_FAILED”, and Caption: to “OLE initialization failed. Make sure that the OLE libraries are the correct version.” Close the properties box by clicking in the close box. Then close the resource windows and save the changes when prompted.

Create the MapInfo Support class, and create an instance of it

In Project > ClassWizard (Browse > ClassWizard in v1.5), choose the OLE Automation tab, and click the “Read Type Library” button. Navigate to your MapInfo program directory and select the MAPINFOW.TLB file. Click OK to confirm the classes to be created. This creates the classes that allow you to access MapInfo through the OLE Automation interface.

Open your main program source file (i.e., *projectname.CPP*) and add the following lines of code.

- After all of the other `#includes` add:
`#include "MapInfow.h"`
- Just below the declaration “*CprojectnameApp theApp*”, add the following variable declaration:
`DMapInfo mapinfo;`
- Near the end of *CprojectnameApp::InitInstance*, but before the `OnFileNew()` call, add:
`mapinfo.CreateDispatch("MapInfo.Application");`

Open the file MAPINFOW.H and add the following lines at the bottom of the file:

```
extern DMapInfo mapinfo;
#include "path-to-mapbasic-directory\mapbasic.h"
```

If you’re using Visual C++ v1.5, you must additionally add the OLE libraries to the link command line (Visual C++ v2.x does this automatically). Do this by choosing Options > Project... and clicking the Linker... button. Choose the “Common to both” radio button, and add the following libraries to the Libraries: text box:

```
comobj, storage, ole2, ole2disp, ole2nls, mfcoleui
```

Test your work

Add one more line of code at the end of the *CprojectnameApp::InitInstance* function, immediately following the *CreateDispatch* call added above:

```
::MessageBox(0, mapinfo.GetFullName(), mapinfo.GetName(), MB_OK);
```

Rebuild your program. When you run, you will get a message box on startup with the title “MapInfo Professional” and the full path to the MapInfo executable in the message area. This demonstrates that you are successfully launching MapInfo and accessing through OLE Automation. You probably want to comment out or remove the *::MessageBox* call as you work through the rest of this exercise.

Redefine the Shortcut Menus

When we incorporate a Map into our application, we’ll get all of the functionality that MapInfo provides for that Map automatically. Sometimes, this functionality is not appropriate. The place where this occurs most often is in the default shortcut menu (accessed by right-clicking on the Map), which includes at least one inappropriate command: Clone Map. To eliminate the inappropriate command, redefine the shortcut menu.

Near the end of *CprojectnameApp::InitInstance*, just after the *CreateDispatch* call we added, we’ll do our additional initialization:

```
// disable the help subsystem: not used in this application
mapinfo.Do("Set Window Help Off");
// Reprogram the mapper shortcut menu
mapinfo.Do("Create Menu \"MapperShortcut\" ID 17 as \"(-)\"");
```

This is also a good time to do other initialization, such as opening tables that you know you’ll need.

Reparenting MapInfo’s Dialogs

It’s important to reparent MapInfo’s dialogs to your application window in case MapInfo needs to interact with the user. By doing this, you ensure that the dialog appears over your application and that your application window is disabled while the user interacts with the MapInfo dialog. This one statement reparents both dialogs that you ask MapInfo to show (for example, by using *RunMenuCommand* with predefined item numbers) and error and warning messages that MapInfo shows in response to unusual events.

In *MainFrm.CPP*, function *CMainFrame::OnCreate*, we need to do the following:

- After all of the other *#includes* add:

```
#include "MapInfow.h"
```

- At the end of *CMainFrame::OnCreate*, add:

```
char str[256];
sprintf(str, "Set Application Window %lu", (long) (UINT)m_hWnd);
mapinfo.Do(str);
```

Demonstrate that this works by adding the following statement to the *CprojectnameApp::InitInstance* function, just *after* the *OnFileNew()* call. This will cause MapInfo to display one of its standard dialogs within the context of your application:

```
mapinfo.Do("Note \"Hello from MapInfo\"");
```

Please test your application at this point to ensure that it is working properly.

Adding a Map to your View

Now that you have a functioning MFC application that attaches to MapInfo through OLE Automation, you can start taking advantage of MapInfo's capabilities. In particular, we'll now add a Map to this application.

Go to the Project > ClassWizard (Browse > ClassWizard in v1.5) dialog box. Select the view class (*CprojectnameView*), and the "Message Maps" tab. Select the "*CprojectnameView*" object in the leftmost listbox.

In the Messages listbox, select "WM_CREATE", then press **Add Function**; select "WM_DESTROY", then press **Add Function**; and select "WM_SIZE", then press **Add Function**.

In the view header file (*projectnameVW.H*), add the following member variables to the view class:

```
unsigned long m_windowid;
HWND         m_windowhwnd;
```

In the view source file (*projectnameVW.CPP*), add the following:

- After all of the other #includes add:

```
#include "MapInfow.h"
```
- In the constructor (*CprojectnameView::CprojectnameView*), initialize the variables:

```
m_windowid = 0;
m_windowhwnd = 0;
```
- In the OnCreate method, add the following code after the call to *CView::OnCreate*:

```
//must have ClipChildren style for integratable maps to work
SetWindowLong(m_hWnd, GWL_STYLE,
               GetWindowLong(m_hWnd, GWL_STYLE)
               |WS_CLIPCHILDREN);

char str[256];
mapinfo.Do("Open Table \"States\" Interactive");
sprintf(str,
        "Set Next Document Parent %lu Style 1 Map From States",
        (long) (UINT)m_hWnd);
mapinfo.Do(str);
m_windowid = atol(mapinfo.Eval("WindowID(0)"));
sprintf(str, "WindowInfo(0, %u)", WIN_INFO_WND);
m_windowhwnd = (HWND)atol(mapinfo.Eval(str));
```
- In the OnDestroy method, add the following code *before* the call to *CView::OnDestroy*:

```
if (m_windowhwnd) {
    ::DestroyWindow(m_windowhwnd);
    m_windowhwnd = NULL;
    m_windowid = 0L;
}
```
- In the OnSize method, add the following code after the call to *CView::OnSize*:

```
if (m_windowhwnd && cx > 0 && cy > 0) {
    ::MoveWindow(m_windowhwnd, 0, 0, cx, cy, TRUE);
}
```

Adding a Map Menu Command

All menu items can be added using the example procedure described below. The example shows how to add a “Map>Layer Control” menu item.

1. Open your resource file (i.e., *projectname.RC*), open the “Menu” resource, and select IDR_MAINFRAME. (In Visual C++ 1.5, you’ll need to use the AppStudio to edit the resources.)
2. Add a new main menu item titled “Map”. Under “Map” add a “Layer Control” item and save the changes to the RC file.
3. In Project > ClassWizard (Browse > ClassWizard... in v1.5), chose the Message Map tab, and select *CprojectnameView* from the Class Name list. In the Object ID’s list select the ID that maps to the menu item you just created - this will be ID_MAP_LAYERCONTROL by default. Once you select this, the COMMAND and UPDATE_COMMAND_UI messages will appear in the Messages window. Add function prototypes for each message by selecting each and pressing Add Function, accepting the default names generated.
4. In your *CprojectnameView* class you’ll see both functions added. Add the following lines of code to the function bodies.

```
void CprojectnameView::OnMapLayercontrol()
{
    mapinfo.RunMenuCommand(M_MAP_LAYER_CONTROL);
}
void CprojectnameView::OnUpdateMapLayercontrol(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(m_windowid);
}
```

Adding Toolbar Buttons and Handlers

All toolbar buttons can be added using the example procedure described below. The example will show how to add the MapInfo selector, grabber, zoom-in, and zoom-out tools to the toolbar. For convenience, we’ll also add them to a new menu named Tools; this makes adding them to the toolbar a little easier using the ClassWizard.

1. First, follow the instructions listed above (Adding a Map Menu Command) and create a new menu named Tools, with four new items (Selector, Grabber, Zoom-In, Zoom-Out). Define the UPDATE_COMMAND_UI and COMMAND functions as before, using the appropriate codes from the MAPBASIC.H file for each tool (M_TOOLS_SELECTOR, M_TOOLS_RECENTER, M_TOOLS_EXPAND, and M_TOOLS_SHRINK, respectively). Compile and test your application when you’re done.
2. Open the project RC file, select the bitmap resource IDR_MAINFRAME, and make the bitmap 64 pixels wider (room for 4 more 16-pixel buttons). Move the images of the last several buttons to the right, making room just after the “paste” button. Draw appropriate images for the four new tools, for example, an arrow (selector), a hand (grabber), a magnifying glass (zoom-in), and a magnifying glass with a minus sign (zoom-out).
3. Open the String resource, add new strings for each of the new tools. Use the same IDs as you used when creating the menu items earlier; the strings should be a descriptive string followed by “\n” and the tooltip text. For example, ID_TOOLS_SELECTOR as “Select map objects\nSelector”; ID_TOOLS_GRABBER as “Recenter the map\nGrabber”; ID_TOOLS_ZOOMIN as “Zoom-In to show less area, more detail\nZoom-In”; and ID_TOOLS_ZOOMOUT as “Zoom-Out to show more area, less detail\nZoom-Out”.

4. In MAINFRM.CPP locate the static UINT BASED_CODE buttons[] array and insert the ID constants into the array in the same position that they appear in the bitmap resource.
5. In order to get the user interface right, we need to keep track of which tool is currently selected. In the *CprojectnameView* header file, add an integer variable to keep track of this:

```
int m_eMouseMode;
```

6. Initialize this variable in the class constructor, to represent the initial state of the map. Note that we'll use the MapInfo constants for the various tools to keep track of which one is selected.

```
m_eMouseMode = M_TOOLS_SELECTOR;
```

7. If you created the menu items first, you already have COMMAND and UPDATE_COMMAND_UI entries in the message map; if not, you should add them now.
8. Update the user interface by calling CCmdUI::SetRadio in each OnUpdate routine, and set the m_eMouseMode variable accordingly in each OnTools*Toolname* handler. That is, your routines should now read as follows:

```
void CprojectnameView::OnToolsSelector()
{
    m_eMouseMode = M_TOOLS_SELECTOR;
    mapinfo.RunMenuCommand(M_TOOLS_SELECTOR);
}
void CprojectnameView::OnToolsGrabber()
{
    m_eMouseMode = M_TOOLS_RECENTER;
    mapinfo.RunMenuCommand(M_TOOLS_RECENTER);
}
void CprojectnameView::OnToolsZoomin()
{
    m_eMouseMode = M_TOOLS_EXPAND;
    mapinfo.RunMenuCommand(M_TOOLS_EXPAND);
}
void CprojectnameView::OnToolsZoomout()
{
    m_eMouseMode = M_TOOLS_SHRINK;
    mapinfo.RunMenuCommand(M_TOOLS_SHRINK);
}
void CprojectnameView::OnUpdateToolsSelector(CCmdUI* pCmdUI)
{
    pCmdUI->SetRadio(m_eMouseMode == M_TOOLS_SELECTOR);
    pCmdUI->Enable(m_windowid);
}
void CprojectnameView::OnUpdateToolsGrabber(CCmdUI* pCmdUI)
{
    pCmdUI->SetRadio(m_eMouseMode == M_TOOLS_RECENTER);
    pCmdUI->Enable(m_windowid);
}
void CprojectnameView::OnUpdateToolsZoomin(CCmdUI* pCmdUI)
{
    pCmdUI->SetRadio(m_eMouseMode == M_TOOLS_EXPAND);
    pCmdUI->Enable(m_windowid);
}
void CprojectnameView::OnUpdateToolsZoomout(CCmdUI* pCmdUI)
{
    pCmdUI->SetRadio(m_eMouseMode == M_TOOLS_SHRINK);
    pCmdUI->Enable(m_windowid);
}
```

Using Exception Handling to Catch MapInfo Errors

MapInfo communicates error conditions to the Integrated Mapping application using the MFC `COleDispatchException` class. MapInfo returns the error code in the `COleDispatchException` member variable `m_wCode`, and a description string in the `COleDispatchException` member variable `m_strDescription`. Other general OLE exceptions are passed via the `COleException` class. You must handle these exceptions somewhere in your application; if not, the top-level MFC exception handler will be invoked and will get the message "Command failed". You can add handlers for each type of exception in each of the `DMapInfo` methods. The following illustrates this in the `DMapInfo::Do` method.

The original `DMapInfo::Do` method, as generated by the ClassWizard, looks like this:

```
void DMapInfo::Do(LPCTSTR command)
{
    static BYTE BASED_CODE parms[] = VTS_BSTR;
    InvokeHelper(0x6001000b, DISPATCH_METHOD, VT_EMPTY,
        NULL, parms, command);
}
```

The improved `DMapInfo::Do` method, with exception handling built-in, looks like this:

```
void DMapInfo::Do(LPCTSTR command)
{
    static BYTE BASED_CODE parms[] = VTS_BSTR;
    try {
        InvokeHelper(0x6001000b, DISPATCH_METHOD, VT_EMPTY,
            NULL, parms, command);
    }
    catch(COleDispatchException *e) {
        // Handle the exception in a manner appropriate to your
        // application. The error code is in e->m_wCode.
        AfxMessageBox(e->m_strDescription);
        e->Delete();
    }
    catch(COleException *e) {
        AfxMessageBox("Fatal OLE Exception!");
        e->Delete();
    }
}
```

Add OLE Automation Server Support

In your `CprojectnameDoc.cpp` file, add the Dispatch map after the Message map.

```
BEGIN_DISPATCH_MAP(CprojectnameDoc, CDocument)
//{{AFX_DISPATCH_MAP(CprojectnameDoc)
//NOTE:The ClassWizard will add and remove mapping macros here
//DO NOT EDIT what you see in these blocks of generated code!
//}}AFX_DISPATCH_MAP
END_DISPATCH_MAP()
```

In your `CprojectnameDoc.cpp` file, add to the `CprojectnameDoc` constructor:

```
EnableAutomation();
AfxOleLockApp();
```

In your `CprojectnameDoc.cpp` file, add to the `CprojectnameDoc` destructor:

```
AfxOleUnlockApp();
```

In your *CprojectnameDoc.h* header file, add the Dispatch section after the message map:

```
// Generated OLE dispatch map functions
//{{AFX_DISPATCH(CprojectnameDoc)
//NOTE:The ClassWizard will add and remove member functions here.
//DO NOT EDIT what you see in these blocks of generated code !
//}}AFX_DISPATCH
DECLARE_DISPATCH_MAP()
```

Note: The above code fragments illustrate adding automation support to your CDocument derived class. When using MFC, you can add automation support just as easily to any class derived from CCmdTarget. Thus, for an MDI application, you will want to attach the automation interface to either your CWinApp derived class or your CMDIFrameWnd derived class, both of which are derived from CCmdTarget because you only want to set the IDispatch pointer for MapInfo callbacks once. In an MDI application, documents and their views are destroyed when closed. If you set the IDispatch pointer to a document, it will no longer be valid when the document is closed.

Adding the WindowContentsChanged Callback

If you're writing an SDI application and you added the automation DISPATCH message map to your *CprojectnameDoc* class, then you can set the callback pointer in your *CprojectnameDoc* constructor, or any where else where it will only be called once.

```
mapinfo.SetCallback(this->GetIDispatch(FALSE));
```

In Project > Class Wizard, choose the OLE Automation tab, and select from the Class Name list the class that has OLE Automation enabled (for this example it is your *CprojectnameDoc* class). Choose "Add Method" and fill in the method name as "WindowContentsChanged", return type as "SCODE", and argument list as "long IWindowID". When you Choose OK and exit the dialog, the Class Wizard automatically updates your *CprojectnameDoc* cpp and header file. In the cpp file, fill in the function body of WindowContentsChanged to do any post processing necessary. For example, this is a good place to do legend maintenance.

Learning More

To learn more about Integrated Mapping, look at the sample programs provided with the MapBasic development environment. The following samples are provided:

- Samples\VB\FindZip: Visual Basic program, used as an example throughout this chapter.
- Samples\VB\VMapTool: Visual Basic program that demonstrates advanced tasks, such as callbacks; requires Visual Basic 4.0 Professional Edition.
- Samples\MFC\FindZip: A sample MFC application.
- Samples\PwrBldr\Capitals: A sample 16-bit PowerBuilder application. You must have the PowerBuilder runtime environment on your system to run it.
- Samples\Delphi\TabEdMap: A sample Delphi application.

Check the Samples directory (within the MapBasic directory) for additional samples that may have been added after this manual was printed.

Sample Programs

The MapBasic software includes the following sample program files.

Note: Additional examples may have been added after the printing of this manual.

Sections in this Appendix:

♦ Samples\Delphi Folder	263
♦ Samples DLLEXAMP Folder	263
♦ Samples\MapBasic Folder	263
♦ Samples\MFC Folder	268
♦ Samples\PwrBldr Folder	268
♦ Samples\VB4 Folder	268
♦ Samples\VB6 Folder	269

Samples\Delphi Folder

tabmap: run MapInfo as an OLE server using Delphi.

Samples DLLEXAMP Folder

Samples\DLLEXAMP\Loadlib Folder

loadlib: The files in this directory are the source code to a C language DLL that can be compiled for either Win16 or Win32, and a test program written in MapBasic that exercises the function in the DLL.

Samples\DLLEXAMP\ResDLL Folder

Contains sample programs to demonstrate techniques for Win16 & Win32 compatibility.

Samples\MapBasic Folder

The Samples\MapBasic\ folder contains subfolders that include sample program files. The contents of each subfolder is described in the following sections.

Samples\MapBasic\Animator Folder

Animator.mb: demonstrates how Animation Layers can speed up the redrawing of Map windows.

Samples\MapBasic\Appinfo Folder

AppInfo.mb: retrieves information about the MapBasic applications that are currently running.

Samples\MapBasic\Autolbl Folder

AutoLbl.mb: “labels” a map by placing text objects in the Cosmetic layer (emulating the way earlier versions of MapInfo created labels).

Samples\MapBasic\Cogoline Folder

COGOLine.mb: draws a line at a specified length and angle.

Samples\MapBasic\Coordinateextractor Folder

Coordinateextractor.mb: updates two columns with the x and y coordinates in the table’s native projection or a user selected projection for each object in the table.

Samples\MapBasic\Csb Folder

CoordSysBounds.mb: enables you to check and set the coordinate system bounds of any mappable MapInfo base table.

Samples\MapBasic\Database Folder

Autoref.mb: refreshes linked tables every (Interval) seconds

BuildSQL.mb: allows you to connect to DBMS databases, build, save and load queries. run queries and previews or download the results.

Connect.mb: provides the MapInfo DBMS Connection Connection Manager dialog and related functions. The connection manager allows you to select an existing connection to use, disconnect existing connections, and get new connections.

DescTab.mb: provides a dataLink utility function that given a table opens a dialog box that describes it.

DLSUtil.mb: returns the list value at the selection index for Dialog List control processing.

GetMITab.mb: MapInfo table picker dialog.

MIODbCat.mb: This is the DBMS Catalog tool that is loaded from the MapInfo Professional Tool Manager. This allows the database administrator to create a MapInfo User with the with a MAPINFO_MAPCATALOG table. It also allows the DBA to delete a table from the catalog.

MIRowCnt.mb: This is the DBMS Count Rows in Table tool that is loaded from the MapInfo Professional Tool Manager. This tool lets you connect to DBMS databases and run a count(*) against tables, updating the mapcatalog with the results.

MISetMBR.mb: This is the CoordSysBounds tool that is loaded from the MapInfo Professional Tool Manager. This tool allows the DBA to change the bounds of a table in the MapInfo_MAPCATALOG table.

MIUpldDB.mb: provides the ability to generate the Database specific SQL statements allowing you to upload a MapInfo table.

MIUpload.mb: This is the Spatialize SQL Server Table tool that is loaded from the MapInfo Professional Tool Manager. This tool provides the ability to upload a MapInfo table to a remote database with spatial column information. The Spatial columns are used with DBMS linked tables, which allows a remote database table to be mappable in MapInfo.

PickCol.mb: Server table column picker dialog

PickSel.mb: provides a selection picker dialog as part of the BuildSQL.mbx.

PickTab.mb: provides functions to get a list of server database tables, and table owners (schemas), and contains a generic function that provides a table selection dialog.

PrepSQL.mb: SQL Query prepare function that processes query parameters. The parameters are bound here (resolved and replaced with a value).

SQLPVW.mb: Given an SQL query string with embedded parameters of a specific format, resolves each parameter to a value and return the resolved SQL query string.

SQLUtil.mb: provides many utility functions that enable Mapinfo to access to ODBC data.

SQLView.mb: SQL DataLink application for testing the SERVER_COLUMNINFO function for all options (except VALUE).

Samples\MapBasic\Disperse Folder

disperse.mb: takes points at given coordinates and disperses them either randomly or systematically.

Samples\MapBasic\DMSCnvt Folder

DMSCnvt.mb: converts between columns of Degree/Minute/Second coordinates and columns of decimal-degree coordinates.

Samples\MapBasic\Georeg Folder

Georeg.mb: opens a GeoTIFF raster image, registers the image so it can be displayed within MapInfo, and then displays the GeoTIFF image.

Samples\MapBasic\Geoset Folder

Geoset.mb: enables you to create a MapX or MapXtreme Geoset from the layers and settings of a MapInfo Professional Map window, or to read a MapX or MapXtreme Geoset files to load the corresponding tables and layer settings to a MapInfo Professional Map window.

Samples\MapBasic\GridMakr Folder

GridMakr.mb: creates a grid (graticule) of longitude/latitude lines.

Samples\MapBasic\HTMLImageMap Folder

HTMLImageMap.mb: creates a clickable HTML image map from a MapInfo map window for use in a web browser.

Samples\MapBasic\IconDemo Folder

IconDemo.mb: demonstrates the built-in ButtonPad icons provided in MapInfo

Samples\MapBasic\Inc Folder

inc: contains include files that can be useful when programming in the MapBasic environment.

Among these files are:

Definition (.DEF) files used by various of the MapBasic tools installed with MapInfo Professional. AUTO_LIB.DEF and RESSTRNG.DEF are needed by the Tool Manager registration system and the tools' string localization module, respectively (both of these are stored in the \LIB folder.)

MAPBASIC.DEF contains, among other things, the definitions for general purpose macros, logical constants, angle conversion, colors, and string length. These are used as inputs for various MapBasic functions.

MENU.DEF contains the definitions needed to access and/or modify MapInfo Professional's dialogs, toolbars, and menu items.

MAPBASIC.H is the C++ version of MAPBASIC.DEF plus MENU.DEF.

MAPBASIC.BAS is the Visual Basic 6.0 version of MAPBASIC.DEF plus MENU.DEF.

Samples\MapBasic\Labeler Folder

labeler.mb: allows you to transfer your layers labels into permanent text objects, allow you to label the current selection, and allow you to use a label tool and individually label objects into permanent text objects.

Samples\MapBasic\Legends Folder

Legends.mb: allows you to manage two or more Legend windows in MapInfo. (The standard MapInfo user interface has only one Legend window.)

Samples\MapBasic\Lib Folder

lib: contains a library of functions and subroutines that can be useful when programming in the MapBasic environment.

In particular, two of these files are used by many of the MapBasic tools installed with MapInfo Professional:

AUTO_LIB.MB is used by most tools to help register themselves into the Tools directory.

RESSTRNG.MB is used by the localized tools to look up the appropriate language strings in the tools' .STR files.

Samples\MapBasic\Linesnap Folder

linesnap.mb: allows you to trim or extend a single-segment line to its intersection point with another chosen line.

Samples\MapBasic\Mapwiz Folder

mapwiz.mb: provides a template which can be used to create a Tool Manager application.

Samples\MapBasic\NorthArrow Folder

northarrow.mb: MapBasic program to create North Arrows.

Samples\MapBasic\Packager Folder

packager.mb: packages a copy of a workspace into a single directory for easier backups, compression, or transfer between computers.

Samples\MapBasic\Regvector Folder

regvector.mb: allows you to copy a table of vector objects (regions, polylines, points, etc.) from one location to another by specifying target locations for three points in the original table.

Samples\MapBasic\RingBuffer Folder

ringbuf.mb: allows you to create multiple "donut" ring buffers. It also will calculate sums and averages of underlying data within each ring.

Samples\MapBasic\RMW Folder

rotatemapwindow.mb: enables you to rotate the contents of the current map window a specific number of degrees.

Samples\MapBasic\RotateLabels Folder

rotatelabels.mb: allows you to rotate labels.

Samples\MapBasic\RotateSymbols Folder

rotatesymbols.mb: allows you to rotate symbols in a table.

Samples\MapBasic\SeamMgr Folder

seammgr.mb: creates and manages seamless map tables.

Samples\MapBasic\Send2mxm Folder

send2mxm.mb: allows you to write custom MapX Geoset and associated .tab files to create a user-defined subset of a map window's background data for display on a mobile device.

Samples\MapBasic\Shields Folder

Shields.mb: draws decorative frames around text objects. Note that this application only works with true text objects, not map labels.

Samples\MapBasic\Snippets Folder

The Snippets folder contains sample programs and code snippets that you can incorporate into your custom MapInfo applications.

Note: In addition to containing sample code snippets, this folder *also* contains three tools that are installed with MapInfo Professional Tool Manager. These are the Named Views tool [NIEWS.MBX], the Overview tool [OVERVIEW.MBX] and the Scalebar drawing tool [SCALEBAR.MBX].

acad.mb: uses DDE to communicate with AutoCAD for Windows.

addnodes.mb: adds nodes to objects. This can be useful if you intend to project a map; the added nodes prevent slivers from appearing between regions in situations where a large region has a long, straight edge.

geocode.mb: demonstrates how to geocode through MapBasic.

geoscan.mb: scans a table to predict a geocoding hit-rate.

get_tab.mb: this is a module, not a complete application. get_tab contains routines to display a dialog that presents the user with a list of open tables. For an example of using the get_tab routines, see the OverView application.

nviews.mb: a “named views” application; lets you enter a name to describe your current “view” of a map (current center point and zoom distance). Once a view is defined, you can return to that view by double-clicking it from the Named Views dialog. To link this application, use the project file nvproj.mbp.

objinfo.mb: displays descriptive information about an object.

overview.mb: opens a second Map window to show an overview of the area in an existing Map window. As you zoom in or out or otherwise change your view in the original map, the overview window adjusts automatically. To link this application, use the project file obproj.mbp

scalebar.mb: draws a distance scale bar on a map window. To link this application, use the project file sbproj.mbp.

textbox.mb: the sample program used as an example throughout this manual. A printout of the TextBox program appears in Appendix B. To link this application, use the project file tbproj.mbp.

watcher.mb: uses DDE to communicate with Microsoft Excel; sets up an Excel worksheet to monitor global variables in a MapBasic application.

Samples\MapBasic\Spider Graph Folder

Spider Graph: draws lines between objects in a single table, or the objects from two tables based on a join. It then creates a new table of lines that connect the objects from the original table(s) based on matching column names.

Samples\MapBasic\Srchrepl Folder

Srchrepl: performs search-and-replace operations within a table.

Samples\MapBasic\SWSpatialize Folder

sw_spatialize: allows an existing SQL Server table that has not been set up for spatial data to be spatialized. When a SQL Server table is spatialized, it can have spatial data inserted into and extracted from it.

Samples\MapBasic\Symbol Folder

symbol: allows you to create/edit/delete MapInfo symbols. Editor that lets you customize the MapInfo 3.0 symbol set.

Samples\MapBasic\SyncWindows Folder

syncwindows: synchronizes mapper windows, creates objects in all mapper windows, tiles windows, clears cosmetic layer in all map windows.

Samples\MapBasic\Tablemgr Folder

tablemgr: lists all open tables in a list box and provides more information about a table as the user clicks on it. Also allow the user to set some table properties and view table metadata.

Samples\MapBasic\Template Folder

templates

Samples\MapBasic\Winmgr Folder

winmgr: allows you to set the title of a document window title and set the default view for a table.

Samples\MFC Folder

FindZip: Demonstrates how Integrated Mapping allows you to integrate elements of MapInfo into a C++ program written using Microsoft Foundation Class (MFC).

mdimfc: contains header files and other supporting files.

Samples\PwrBldr Folder

Capitals: An Integrated Mapping application using PowerBuilder. Note: The PowerBuilder runtime libraries are not provided; you must already have PowerBuilder libraries installed to run this application.

Samples\VB4 Folder

Callback: OLE automation callbacks.

FindZip: Demonstrates how Integrated Mapping allows you to integrate elements of MapInfo, such as a Map window, into a Visual Basic program. Requires Visual Basic 3.0 or later.

VMapTool: A demonstration of advanced Integrated Mapping tasks, such as callbacks. Requires Visual Basic 4.0 Professional Edition or later.

Samples\VB6 Folder

Callback: OLE automation callbacks.

FindZip: Demonstrates how Integrated Mapping allows you to integrate elements of MapInfo, such as a Map window, into a Visual Basic program. Requires Visual Basic 3.0 or later.

VMapTool: A demonstration of advanced Integrated Mapping tasks, such as callbacks. Requires Visual Basic 4.0 Professional Edition or later.

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 271**
- ♦ **Comparison Operators. 272**
- ♦ **Logical Operators 272**
- ♦ **Geographic Operators 273**
- ♦ **Automatic Type Conversions 275**

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)	$a \setminus b$
Mod	remainder from integer division	$a \text{ Mod } b$
^	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 (for example, 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

Geographic 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	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	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).

Priority	MapBasic Operator
(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</i> + <i>Number</i> <i>Number</i> + <i>Date</i> <i>Integer</i> + <i>Integer</i> <i>Number</i> + <i>Number</i> <i>Other</i> + <i>Other</i>	Date Date Integer Float String
-	<i>Date</i> - <i>Number</i> <i>Date</i> - <i>Date</i> <i>Integer</i> - <i>Integer</i> <i>Number</i> - <i>Number</i>	Date Integer Integer Float
*	<i>Integer</i> * <i>Integer</i> <i>Number</i> * <i>Number</i>	Integer Float
/	<i>Number</i> / <i>Number</i>	Float
\	<i>Number</i> \ <i>Number</i>	Integer
MOD	<i>Number</i> MOD <i>Number</i>	Integer
^	<i>Number</i> ^ <i>Number</i>	Float

List of MapBasic Changes by Version

This section summarizes the enhancements made in recent versions of MapBasic. For details on each enhancement, see the appropriate discussion in the MapBasic *Reference* or online Help. Details of the current version can be found in the "What's New" section in the front of this user guide.

Sections in this Appendix:

- ♦ **Features Introduced or Changed in MapBasic 7.8 277**
- ♦ **Features Introduced in MapBasic 7.5 278**
- ♦ **Features Introduced in MapBasic 7.0 278**

Features Introduced or Changed in MapBasic 7.8

For more information about these statements and functions, see the *MapBasic Reference Guide*.

New Statements and Functions

MGRSToPoint Statement - Converts a string representing an MGRS coordinate to an object value representing a point.

Save MWS Statement - Saves the current workspace to an XML-based MWS file.

PointToMGRS Statement - Converts an object value representing a point to a string representing an MGRS coordinate.

Objects Pline Statement - Splits a single section polyline into two polylines.

WFS Refresh Statement - The WFS Refresh Table statement refreshes a WFS table from the server.

Enhanced Statements and Functions

Create Cartographic Legend Statement - The Create Cartographic Legend statement has a new clause that creates the small or large legend sample size for an active map window.

Export Statement - Now includes a CSV export option.

LegendInfo() function - The LegendInfo() function has a new attribute that returns the legend size information.

Objects Snap Statement - 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 settings specified in the Objects Snap statement are written to the input table's metadata when the Objects Snap statement is executed. These settings become the default values for the table when the Set Values for Node Snap and Thinning dialog box is opened.

PrintWin Statement - The MapBasic PrintWin statement has been enhanced so that a MapInfo window can be printed to a file.

Register Table Statement - When opening an Excel spreadsheet, the user can now specify how each column is imported. For example, the user might want a 'Date' column in Excel to be a Text column in MapInfo Professional. The MapBasic Register Table statement has been enhanced to support this new feature. Additionally, we have added a WFS Type to this statement so you can register WFS files you retrieve from the Internet.

Set Cartographic Legend Statement - The Set Cartographic Legend statement now allows you to control the sample legend sizes that appear in Cartographic Legend windows.

Shade Statement - The MapBasic Shade statement now allows for circle/square dot density thematics with user-defined size.

TableInfo() variable - The MapBasic TableInfo() variable includes a new value for the TAB_INFO_TYPE attribute to support Web Feature Service (WFS). The purpose of this the TableInfo() function is to return information about an open table.

Features Introduced in MapBasic 7.5

New Statements and Functions

Modifying Map Objects:

- Objects Move Statement

- Objects Offset Statement

Creating Map Objects – Advanced Functions:

- Offset() Function

- OffsetXY() Function

- CartesianOffset() Function

- CartesianOffsetXY() Function

- SphericalOffset() Function

- SphericalOffsetXY() Function

Enhanced Statements and Functions

WMS Support:

- LayerInfo() function

- Register Table Statement

- TableInfo() function

Mapping a Remote Database:

- Server Create Map Statement

Features Introduced in MapBasic 7.0

New Statements and Functions

Control DocumentWindow statement:

- Create Cutter statement

- CurrentBorderPen() function

- CurrentLinePen() function

- Rotate() function

- RotateAtPoint() function

- ServerCreateTable statement

- SessionInfo() function

- Set Style statement

- TextSize() function

Enhanced Statements and Functions

Shapefile Support:

- Register Table statement

Clone table structure:

- Create Map statement

Create Voronoi Polygons:

- Create Object statement

Create New Tables on DBMS Servers and Combine Objects using Column enhancements:

- Commit Table statement

- Create Table statement

- Server Create Map statement

DBMS Driver:

- Server Connect() function

Current Pen enhancement:

- CurrentPen() function

File Location enhancement:

- GetFolderPath\$() function

Import GML Files:

- Import statement

Labeling Partial Objects:

- LayerInfo() function

Raster Transparency enhancements:

- Set Window statement

ReadControlValue() Additional Types:

- ReadControlValue() function

User Defined Resolution and JPEG 2000 Support:

- Save Window statement

Supported ODBC Table Types

These are the ODBC data types that MapInfo supports:

- SQL_BIT
- SQL_TINYINT
- SQL_SMALLINT
- SQL_INTEGER:
- SQL_REAL
- SQL_BIGINT
- SQL_DECIMAL
- SQL_DOUBLE
- SQL_FLOAT
- SQL_NUMERIC
- SQL_BINARY
- SQL_LONGVARBINARY
- SQL_VARBINARY
- SQL_LONGVARCHAR
- SQL_DATE
- SQL_TYPE_DATE
- SQL_TIMESTAMP
- SQL_TYPE_TIMESTAMP
- SQL_TIME
- SQL_TYPE_TIME
- SQL_CHAR
- SQL_VARCHAR

Making a Remote Table Mappable

Sections in this Appendix:

- ♦ Prerequisites for Storing/Retrieving Spatial Data 282
- ♦ Creating a MapInfo Map Catalog 282

Prerequisites for Storing/Retrieving Spatial Data

There are four prerequisites for storing and retrieving points on an RDBMS table.

1. The coordinate values for the spatial data must be stored in columns of the table as numbers or a supported spatial data type.
Possible methods for accomplishing this include:
 - Existing data.
 - Use Easyloader to upload to the database. This application will work for all supported databases.
This is a data creation task and can be done at any time.
2. To increase performance on queries against the coordinates, a spatial index column can be included. This is done as part of the sample upload applications, if it is desired. This is a data creation task and can be done at any time.
3. MapInfo stores information about which columns are the coordinates in a special table on the RDBMS system known as the MapInfo Map Catalog. There must be one catalog per database. To create the Map Catalog use Easyloader or MIODBCAT.MBX. You can also follow the procedure for manually creating a map catalog, described in the next section. This is a once only task and is required before ANY tables on that database can be mapped in MapInfo.
4. MapInfo gets catalog information about mappable tables using the MapBasic statement **Server Create Map**. This is a once per table task and is required before this specific table can be mapped in MapInfo.

Creating a MapInfo Map Catalog

You cannot make an ODBC table mappable unless a MapInfo Map Catalog has been created for the database where the table resides. The MapInfo Map Catalog should be created by your database administrator.

1. Create the user MAPINFO with the PASSWORD ***** in the specific database where the mappable tables are located.
2. Create the table MAPINFO_MAPCATALOG in the database.
3. The Create table statement needs to be equivalent to this MapInfo create statement for the specific remote database.

```

Create Table MAPINFO_MAPCATALOG
    SPATIALTYPE          Float,
    TABLENAME           Char(32),
    OWNERNAME            Char(32),
    SPATIALCOLUMN        Char(32),
    DB_X_LL              Float,
    DB_Y_LL              Float,
    DB_X_UR              Float,
    DB_Y_UR              Float,
    COORDINATESYSTEM     Char(254),
    SYMBOL               Char(254),
    XCOLUMNNAME           Char(32),
    YCOLUMNNAME           Char(32),
    RENDITIONTYPE         integer
    RENDITIONCOLUMN       Char(32)
    RENDITIONTABLE        Char(32)

```

It is important that the structure of the table is exactly like this statement. The only substitution that can be made is for the databases that support varchar or text data types. These data types can be substituted for the Char datatype.

4. Create a unique index on the TABLENAME and the OWNERNAME, so only one table for each owner can be made mappable.
5. Grant Select, Update, and Insert privileges on the MAPINFO_MAPCATALOG. This allows the tables to be made mappable by users. The delete privilege should be reserved for database administrators.

Data Setting and Management

Sections in this Appendix:

- ♦ Upgrading Applications from Versions Prior to 6.5 285
- ♦ Application Data Files and Directories 287
- ♦ Default Preferences Paths 289
- ♦ Registry Changes 289
- ♦ Installer Requirements and Group Policies 290

Upgrading Applications from Versions Prior to 6.5

MapInfo Professional Data and Settings Management- Application Data Files

Application data (appdata) files are the non-executable, non-user data files that MapInfo Professional uses during execution. The following files/directories are considered appdata for version 6.5:

Filename	Description
mapinfow.prf	Preference file
mapinfow.wor	Default workspace
startup.wor	Startup workspace
mapinfow.clr	Color file
mapinfow.pen	Pen file
mapinfow.fnt	Symbol file
custsymb	Custom symbol directory
thmtmpl	Theme template directory
graphsupport	Graph support directory

Traditionally these files have been kept in the Windows directory or the Program directory. The strategy for 6.5 is to install application data files in a per user location, and search for them in other areas as well to allow support for sharing application data files between MapInfo versions/products. By relocating mapinfow.prf a user can share one custom projection file between different versions of MI Pro.

The following files remain in the Program directory:

Filename	Description
mapinfow.abb	Abbreviation file
mapinfow.prj	Projection file
mapinfow.mnu	Menu file

Keep in Mind:

- The installer never asks the user where they want to place application data files.
- The installer always runs the same way, whether the user has MI Pro 6.0 installed or not.
- There is not an "upgrade" install for 6.5 (i.e., you cannot install 6.5 into the same directory as 6.0, the installer will error).
- Application developers can move or copy files where they want, but MI Pro 6.5 will search for them only in these locations and in this order:
appdata_dir, local_appdata_dir, pref_dir, program_dir

A Glossary for Upgrading Applications

You will find the following definitions useful:

<user profile root>

The root the user directory structure. Each user has write access to the subdirectories of this area. The location varies depending on the Windows version:

Windows 2000 : c:\Documents and Settings\<username>

Windows 98: <Windows dir>

Windows NT 4.0: <Windows dir>\profiles\<username>

<My Documents>

Windows 2000: c:\Documents and Settings\<username>\My Documents

Windows 98: c:\MyDocuments

Windows NT 4.0: <Windows dir>\profiles\<username>\personal

Pref_dir

MI Pro writes out mapinfow.prf and mapinfow.wor by default.

6.0: Windows directory

6.5: <user profile root>\Application Data\MapInfo\MapInfo. If this directory does not exist at startup, then MI Pro creates it.

home_dir

Obsolete (pertained to prior version support of MacIntosh and UNIX)

6.0: Windows directory

6.5: Windows directory

program_dir

In 6.0 MI Pro expects to find many of the appdata files in this location.

6.0: location of mapinfow.exe

6.5: location of mapinfow.exe

appdata_dir

This is a per user directory introduced in 6.5. Many of the appdata files will be install in this location.

6.0: n/a

6.5: <user profile root>\Application Data\MapInfo\MapInfo\Professional\650.

If this directory does not exist at start up, MI Pro does not create it. Programmer's must not assume this represents a valid path.

local_appdata_dir

This is also a per user directory, similar to appdata_dir, except files here do not roam.

6.0: n/a

6.5: <user profile root>Local Settings\Application Data\MapInfo\MapInfo\Professional\650.

Note: If this directory does not exist at start up MI Pro does not create it. Programmer's must not assume this represents a valid path.

common_appdata_dir

This directory is shared by all users on a machine. By default users have read access to everything, permission to create files and write access to files they create. File in this directory do not roam. Support for this directory was added in version 7.0.

6.0: n/a

6.5: n/a

7.0 : <profile root>\All Users\Application Data\MapInfo\MapInfo\Professional\700

mydocs_dir

Refers to the My Documents directory of the current user.

6.0: n/a

6.5: <My Documents>

search_for_file

This function locates appdata files. It searches directories for them in the following pre-defined order:

6.0: pref_dir, home_dir, program_dir

6.5: appdata_dir, local_appdata_dir, pref_dir, program_dir

7.0: appdata_dir, local_appdata_dir, pref_dir, common_appdata_dir, program_dir

Application Data Files and Directories

The following list describes how both MI Pro 6.0 and 6.5 searches for the appdata files and directories.

mapinfow.prf

6.0: Uses search_for_file. Regardless of where the files was read from, always writes out to pref_dir.

6.5: Uses search_for_file. If found, then reads the file and remembers the location.

On exit, if file was found at start up and if the user has write access to it, then write it out to that location. Otherwise, write the file to pref_dir.

mapinfow.wor

6.0: Looks in pref_dir, then home_dir. Loads the first one it finds.

6.5: Uses search_for_file. If found, reads the file and remembers the location.

On exit, if file was found at start up, and if the user has write access to it, then write it out to that location. Otherwise, write the file to pref_dir.

startup.wor

6.0: Loaded in order from the following directories: program_dir, pref_dir.

6.5: Loaded in order from the following directories: pref_dir, appdata_dir, local_appdata_dir, pref_dir, program_dir. Unlike other appdata files, each startup.wor that is found is processed.

mapinfow.clr

6.0: Uses search_for_file. If not found, then displays dialog for user to find.

If user chooses to save custom defined colors, a new color file is written to pref_dir (or overwrites an existing one).

6.5: Uses search_for_file. If not found, then displays dialog for user to find.

If user chooses to save custom defined colors and the color file was located in a per-user directory, then MI Pro updates the existing file. If the color file was read from the program directory, or if the user does not have write access to the file, then MI Pro writes the file to the pref_dir.

mapinfow.pen

6.0: Uses search_for_file. If not found, then displays dialog for user to find.

6.5: Uses search_for_file. If not found, then displays dialog for user to find.

mapinfow.fnt

6.0: Uses search_for_file. If not found, then displays dialog for user to find.

6.5: Uses search_for_file. If not found, then displays dialog for user to find.

custsymb directory

6.0: Assumes it is under the program_dir.

6.5: Looks for symbol dir by using search_for_file. If not found, then assumes it is under the program_dir.

thmtmpl directory

6.0: If the template dir is specified in the preference file exists, then use it. Otherwise, try to create a template dir under the program_dir. If it cannot be created under program_dir, no template dir is set.

In all cases, including the last, MI Pro updates the preference file path.

6.5: If the template dir specified in the preference file exists, then use it. Otherwise, look for the template dir using search_for_file. If found, then use it. Otherwise, try creating template dir off appdata_dir, then program_dir. Otherwise, no template dir is set. In any case, MI Pro does not set the preference file path.

graphsupport directory

6.0: Use the directory specified in the preference file regardless of whether it exists. If the specified directory is invalid, then the user gets an error message when trying to create a new graph.

6.5: If the template dir specified in the preference file exists, then use it. Otherwise, look for the graph support dir using search_for_file. If found, then use it. If not, then assume it is off the program_dir, and the user will get error message when trying to create a graph).

Note: In version 7.0 the search_for_file routine includes common_appdata_dir.

Default Preferences Paths

The following table lists preference paths and their 6.0/6.5 /7.0 defaults:

Path	6.5 default	7.0 default
Tables	mydocs_dir	mydocs_dir
Workspaces	mydocs_dir	mydocs_dir
MapBasic Programs	<program_dir>\Tools	<program_dir>\Tools
Import Files	mydocs_dir	mydocs_dir
SQL Queries	mydocs_dir	mydocs_dir
Theme Templates	appdata_dir\thmtmpl if exists, program_dir\thmtmpl otherwise	uses search_for_file then program_dir if that fails
Saved Queries	mydocs_dir	mydocs_dir
New Grids	mydocs_dir	mydocs_dir
Crystal Report files	mydocs_dir	mydocs_dir
Graph Support files	local_appdata_dir if exists, program_dir otherwise	uses search_for_file then program_dir if that fails
Search Directories for Tables	mydocs_dir	mydocs_dir

Registry Changes

MapInfo Professional's use of the registry must be organized to allow each user to work with their own data. The following changes were made to support this organization:

- The Tool Manager entries are now installed under HKEY_CURRENT_USER.
- The graph engine now stores custom colors and number formats under HKEY_CURRENT_USER.

Installer Requirements and Group Policies

MapBasic 6.5

The 6.5 application data files should be installed to directories as specified in the following table:

Filename	6.5 Workstation
mapinfow.clr	Application Data\MapInfo\MapInfo\Professional\650
mapinfow.pen	Application Data\MapInfo\MapInfo\Professional\650
mapinfow.fnt	Application Data\MapInfo\MapInfo\Professional\650
mapinfow.abb	Program directory
mapinfow.prj	Program directory
mapinfow.mnu	Program directory
custsymb	Application Data\MapInfo\MapInfo\Professional\650
thmtmplt	Application Data\MapInfo\MapInfo\Professional\650
graphsupport	Local Settings\Application Data\MapInfo\MapInfo\Professional\650

MapBasic 7.0

The 7.0 application data files should be installed to directories as specified in the following table:

Filename	7.0 Workstation
mapinfow.clr	Application Data\MapInfo\MapInfo\Professional\700
mapinfow.pen	Application Data\MapInfo\MapInfo\Professional\700
mapinfow.fnt	Application Data\MapInfo\MapInfo\Professional\700
mapinfow.abb	Program directory
mapinfow.prj	Program directory
mapinfow.mnu	Program directory
custsymb	Application Data\MapInfo\MapInfo\Professional\700
thmtmplt	Application Data\MapInfo\MapInfo\Professional\700
graphsupport	All Users Application Data\MapInfo\MapInfo\Professional\700

The following MapBasic functions have been added to help with file locations:

GetFolderPath\$() function - Returns the path of a special MI Pro or Windows folder.

LocateFile\$() function - Return the path to one of MI Pro's application data files.

MapBasic Glossary

If you do not find the term you are looking for in this glossary, check the glossary in the MapInfo Professional *User Guide (Unabridged)* on the *Installation CD*.

Term	Definition
Aggregate functions	Functions such as Sum() and Count(), which calculate summary information about groups of rows in a table. See Select in the MapBasic <i>Reference</i> or online Help.
Alias	A name by which a MapInfo user (or a MapBasic program) refers to an open table. For example, if a table name is "C:\MapInfo\Parcels.Tab," the table's alias would be Parcels. Table aliases may not contain spaces; any spaces in a table name become underscore characters in a table alias. Alias is also a MapBasic data type; an alias variable can store a string expression that represents a column name (for example, "World.Population"). The maximum length of an alias is 32 characters.
Animation Layer	A special "floating" layer added to a map that allows for redraw of objects in that layer only. Modifying an object in the animation layer does not cause other layers to redraw.
Apple Events	Macintosh interapplication protocol that allows applications to exchange instructions and data. Both applications must support Apple Events for a successful exchange.
Argument	Also known as a parameter. Part of a statement or a function call. If a statement or function requires one or more arguments, you must specify an appropriate expression for each required argument. The argument that you specify is passed to the statement or function. In syntax diagrams in the MapBasic <i>Reference</i> and online Help, arguments are formatted in <i>italics</i> .
Array	A grouping of variables of the same type used to keep similar elements together.
Automation, OLE Automation	OLE Automation is technology through which one Windows application can control another Windows application. For example, a Visual Basic application can control MapInfo through MapInfo's Automation methods and properties. See Chapter 12: Integrated Mapping .
Bar Chart	A graph representing values from the user's table. Bar charts can be used in the graph window or can be displayed thematically on the map.
Breakpoint	A debugging aid. To make your program halt at a specific line, place a breakpoint before that line. To place a breakpoint in a MapBasic program, insert a Stop statement and recompile.
Brush Style	An object's fill pattern. The style is comprised of pattern, foreground color, and background color.
ButtonPad	Another word for "toolbar."
By Reference, By Value	Two different ways of passing parameters to a function or procedure. When you pass an argument by reference (the default), you must specify a variable name when you make the function call; the called function can modify the variable that you specify. When you pass an argument by value (using the ByVal keyword), you do not need to specify a variable name.
Client	An application that uses or receives information from another program. Often referred to in database connections or DDE connections.

Term	Definition
Column	Part of a table or database. A table contains one or more columns, each of which represents an information category (for example, name, address, phone number, etc.). Columns are sometimes referred to as "fields." Tables based on raster images do not have columns.
Comment	A programmer's note included in the program. The note has no use in the syntax necessary for compiling the program. In the MapBasic language, an apostrophe (single quotation mark) marks the beginning of a comment. When an apostrophe appears in a statement, MapBasic ignores the remainder of the line (unless the apostrophe appears inside of a literal string expression).
Compiler	A program that takes the text of a program, checks for syntax errors, and converts the code to an executable format.
Control	A component of a dialog box, such as a button or a check box.
Coordinate System	A set of parameters that specifies how to interpret the locational coordinates of objects. Coordinate systems may be earth (for example, coordinates in degrees longitude/latitude) or non-earth (for example, coordinates in feet) based; earth maps are referenced to locations on the earth.
Cosmetic Layer	A temporary layer that exists on every map window. This layer always occupies the topmost position on the layer control. MapInfo's Find command places symbols in the Cosmetic layer to mark where a location was found.
Cursor, Mouse Cursor, Row Cursor	The mouse cursor is a small image that moves as the user moves the mouse. The row cursor is a value that represents which row in the table is the current row; use the Fetch statement to position the row cursor.
DDE	See Dynamic Data Exchange.
Degrees	A unit of measure for map coordinate systems. Some paper maps depict coordinates in terms of degrees, minutes, seconds (for example, 42 degrees, 30 minutes); MapBasic statements, however, work in decimal degrees (for example, 42.5 degrees). See also: Latitude , Longitude .
Derived Column	A column in a query table, produced by applying an expression to values already existing in the base table. See the Add Column statement.
Disabled	A condition where part of the user interface (a menu command, dialog control, or toolbar button) is not available to the user. The disabled item is generally shown as "grayed out" to indicate that it is not available. See also: Enabled .
Dynamic Data Exchange (DDE)	Microsoft Windows-specific protocol that allows different applications to exchange instructions and data. Both applications must be DDE compliant for a successful exchange.
Dynamic Link Library (DLL)	Microsoft Windows files containing shared executable routines and other resources. DLLs are generally called from one program to handle a task which often returns a value back to the original program. DLLs created for use in Windows 3.1 have a 16-bit architecture; DLLs written for Windows NT or Windows 95 have a 32-bit architecture.

Term	Definition
Enabled	The opposite of Disabled; a condition where a menu command, dialog box control, or toolbar button is available for use.
Expression	A grouping of one or more variables, constant values, function calls, table references, and operators.
File input/output, File i/o	The process of reading information from a file or writing information to a file. Note that the MapBasic language has one set of statements for performing file i/o, and another set of statements for performing table manipulation.
Focus	In a dialog box, the active control (the control which the user is currently manipulating) is said to have the focus; pressing TAB moves the focus from one control to the next. Focus also refers to the active application that is running. Switching to a different application (for example, by pressing Alt-Tab on Windows) causes the other application to receive the focus.
Folder	An area for file storage; also called a directory.
Geographic Join	A relational link between two mappable tables based on geographic criteria (for example, by determining which point objects from one table are inside of regions in the other table).
Global Positioning System (GPS)	A hardware/software system that receives satellite signals and uses the signals to determine the receiver's location on the globe.
Global Variable	A variable defined at the beginning of a program that can be used in any procedure or function. Created using the Global statement.
Handler	A procedure in a program. When a specific event occurs (such as the user choosing a menu command), the handler performs whatever actions are needed to respond to the event.
Hexadecimal	A base-16 number system, often used in computer programming. Each character in a hexadecimal number can be 0-9 or A-F. In MapBasic, you must begin each hexadecimal number with the &H prefix (for example, &H1A is a hexadecimal number that equals decimal 26).
Integrated Mapping	Technology that allows MapInfo features, such as Map windows, to be integrated into other applications (such as Visual Basic programs). See Chapter 12: Integrated Mapping .
Keyword	A word recognized as part of the programming language; for example, a statement or function name. In the MapBasic documentation, keywords appear in bold .
Latitude	A type of coordinate, measured in degrees, indicating north-south position relative to the Equator. Locations south of the Equator have negative latitude.
Linked Table	A type of MapInfo table that is downloaded from a remote database. The data is taken from the remote database and transferred locally. The next time the table is linked back to the remote database, MapInfo checks time stamps to see if there are any differences between the two tables. Where differences occur, the table is updated with the new information.

Term	Definition
Linker	A program that combines separate modules from a project file into a single MBX application file.
Literal Value	An expression that defines a specific, explicit value. For example, 23.45 is a literal number, and "Hello, World" is a literal string. Also referred to as a hard-coded value.
Local Variable	A variable that is defined and used within a specific function or procedure. Local variables take precedence over global variables of the same name. Created using the Dim statement.
Longitude	A type of coordinate, measured in degrees, indicating east-west position relative to the Prime Meridian. Locations west of the Prime Meridian have negative longitude.
Loop	A control structure in a program that executes a group of statements repeatedly. Incorrect coding of a loop can create an infinite loop (a situation where the loop never ends).
MapBasic Window	A window in the MapInfo user interface. From MapInfo's Options menu, choose Show MapBasic Window. You can type MapBasic statements into the MapBasic window, without compiling a program.
MBX	A MapBasic executable file, which the user can run by choosing MapInfo's Tools >Run MapBasic Program command. Any MapInfo Professional user can run an MBX file. To create an MBX file, you must use the MapBasic development environment.
Metadata	Information about a table (such as date of creation, copyright notice, etc.) stored in the .TAB file instead of being stored in rows and columns. See Chapter 8: Working With Tables .
Methods, OLE Methods	Part of OLE Automation. Calling an application's methods is like calling a procedure that affects the application. See Chapter 12: Integrated Mapping .
Module	A program file (.MB file) that is part of a project.
Module-level Variable	A variable that can be accessed from any function or procedure in an MB program file, although it cannot be accessed from other MB program files in the same project. Created by placing a Dim statement outside of any function or procedure.
Native	A standard file format. Choosing MapInfo's File > New command creates a native MapInfo table, but a table based on a spreadsheet or text file is not in MapInfo's native file format.
Object	A graphical object is an entity that can appear in a Map or Layout window (for example, lines, points, circles, etc.). A MapBasic object variable is a variable that can contain a graphical object. The Object column name refers to the set of objects stored in a table. An OLE object is a Windows-specific entity (produced, for example, through drag and drop).

Term	Definition
Object Linking and Embedding (OLE)	Technology that allows objects created in one application to be used in another application. An object can be any information such as a map, chart, spreadsheet, sound effect, text, etc. Embedding is the process of inserting an object from a server into a container application.
Operator	A special character or word that acts upon one or more constants, variables, or other values. For example, the minus operator (-) subtracts one number from another.
Parameter	Another word for “argument.”
Pen Style	The line style set for an object. The style is comprised of width, pattern, and color.
Pie Chart	A circle divided into sectors representing values as percentages in comparison to one another. MapInfo can display pie charts in the Graph window or in thematic maps.
Platform	An operating environment for computer software (for example, Windows, Linux).
Procedure, Sub Procedure	A group of statements enclosed within a Sub ... End Sub construction. Sometimes referred to as a routine or a subroutine.
Progress Bar	A standard dialog box that displays a horizontal bar, showing the percent complete.
Project, Project File	A project is a collection of modules. A project file (.MBP file) is a text file that defines the list of modules. Compiling all modules in the project and then linking the project produces an application (MBX) file.
Property, OLE Property	Part of OLE Automation. A property is a named attribute of an OLE object. To determine the object's status, read the property. If a property is not read-only, you can change the object's status by assigning a new value to the property. See Chapter 12: Integrated Mapping .
Raster	A graphic image format that consists of rows of tiny dots (pixels).
Raster Underlay Table	A table that consists of a raster image. This table does not contain rows or columns; therefore, some MapBasic statements that act on tables cannot be used with raster underlay tables.
Record	An entry in a table or database. Each record appears as one row in a Browser window.
Recursion	A condition where a function or procedure calls itself. While recursion may be desirable in some instances, programmers should be aware that recursion may occur unintentionally, especially with special event handlers such as SelChangedHandler.
Remote Data	Data stored in a remote database, such as an Oracle or SYBASE server.
Routine	A group of statements that performs a specific task; for example, you can use the OnError statement to designate a group of statements that will act as the error-handling routine.

Term	Definition
Row	Another word for “record.”
Run Time	The time at which a program is executing. A runtime error is an error that occurs when an application (MBX file) is running.
Runtime	A special version of MapInfo that contains all of the geographic and data-base capabilities of a full version but does not include the specific menu and toolbar options in a standard package. Used to create customized versions of MapInfo.
Scope of Variables	Refers to whether a variable can be accessed from anywhere within a program (global variables) or only from within a specific function or procedure (local variables). If a procedure has a local variable with the same name as a global variable, the local variable takes precedence; any references to the variable name within the procedure will use the local variable.
Seamless Tables	A type of table that groups other other tables together, making it easier to open and map several tables at one time. See Chapter 8: Working With Tables .
Server	An application that performs operations for or sends data to another application (the client). Often referred to in database connections or DDE connections.
Shortcut menu	A menu that appears if the user clicks the right mouse button.
Source Code	The uncompiled text of a program. In MapBasic, the .mb file.
Standard	Standard menu commands and standard toolbar buttons appear as part of the default MapInfo user interface (for example, File > New is a standard menu command). Standard dialog boxes are dialogs that have a predefined set of controls (for example, the Note statement produces a standard dialog box with one static text control and an OK button). If a MapBasic program creates its own user interface element (dialog box, toolbar button, etc.) that element is referred to as a custom dialog, a custom button, etc.
Statement	An instruction in a MapBasic program. In a compiled MapBasic program, a statement can be split across two or more lines.
Status Bar	The bar along the bottom of the MapInfo program window which displays help messages, the name of the editable layer, etc.
Status Bar Help	A help message that appears on the status bar when the user highlights a menu command or places the mouse cursor over a toolbar button.
Subroutine	A group of statements; in MapBasic syntax, subroutines are known as procedures or sub procedures.
Toolbar	A set of buttons. The user can “dock” a toolbar by dragging it to the top edge of the MapInfo work area. The MapBasic documentation often refers to Toolbars as “ButtonPads” because ButtonPad is the MapBasic-language keyword that you use to modify toolbars.

Term	Definition
ToolTip	A brief description of a toolbar button; appears next to the mouse cursor when the user holds the mouse cursor over a button.
Transparent Fill	A fill pattern, such as a striped or cross-hatch pattern, that is not completely opaque, allowing the user to see whatever is “behind” the filled area. See Brush clause .
Variable	A small area of memory allocated to store a value.

Index

- (minus)
 - date subtraction [84](#)
 - subtraction [83](#)

Symbols

- (backslash)
 - integer division [271](#)
- (backward slash)
 - integer division [83](#)
- & (ampersand)
 - finding an intersection [156](#)
 - hexadecimal numbers [80](#)
 - shortcut keys in dialogs [130](#)
 - shortcut keys in menus [118](#)
 - string concatenation [83](#), [271](#)
- * (asterisk)
 - fixedlength strings [74](#)
 - multiplication [83](#), [271](#)
- + (plus) [271](#)
 - addition [83](#)
 - date addition [84](#)
 - string concatenation [83](#)
- , (comma) character
 - thousand separator [80](#)
- . (period) character
 - decimal separator [80](#)
- / (forward slash)
 - date string format [81](#)
 - division [83](#)
- / (slash)
 - division [271](#)
- = (equal sign) [84](#)
- > (greater than) [84](#)
- >= (greater than or equal) [84](#)
- ^ (caret) [83](#)
 - exponentiation [271](#)
- ð (less than or equal) [84](#)
- ≠ (not equal) [84](#)
- ' (apostrophe) [72](#)
- < (less than) [84](#)

A

- Accelerator keys
 - in dialogs [130](#)
 - in integrated mapping [237](#)
 - in menus [118](#)
- Accessing remote databases [173](#)
- Add Column statement [209](#)

- Add Map Layer statement [132](#)
- Adding columns to a table [157](#)
- Adding nodes to an object [194](#)
- Addresses, finding [156](#)
- Advise loops
 - MapInfo as DDE server [223](#)
- Aggregate functions
 - defined [292](#)
 - See MapBasic Reference
- Alias variables [153](#)
- Alias, defined [292](#)
- Alter Button statement [139](#)
- Alter ButtonPad statement [139](#), [217](#)
- Alter Control statement [128](#)
- Alter Menu Bar statement [113](#)
- Alter Menu Item statement [114](#)
- Alter Menu statement [112](#)
- Alter Object statement [194](#), [197](#)
- Alter Table statement [157](#)
- And operator [85](#)
- Animation layers [133](#)
- Any() operator [208](#)
- Area units [204](#)
- Area() function [188](#), [207](#)
- Arguments
 - passing by reference [93](#)
 - passing by value [93](#)
- Arithmetic operators [83](#)
- Array variables
 - declaring [75](#)
 - resizing [75](#)
- Ask() function [121](#)
- Assigning values to variables [74](#)
- auto_lib.mb (sample program) [147](#)
- AutoLabel statement [194](#)
- Automation
 - defined [292](#)
 - object model [244](#)

B

- Bar charts
 - in graph windows [135](#)
 - in thematic maps [132](#)
- Beeping
 - because window is full [59](#)
- Between operator [84](#)
- BIL (SPOT image) files [167](#)
- Binary file i/o [179](#), [182](#)
- Bitmap image files [167](#)

- Branching** 89
- Breakpoints (debugging)** 105
- Browser windows** 134
- Brush styles** 188
- BrushPicker controls** 126
- Buffers, creating** 196, 266
- Button controls (in dialogs)** 127
- ButtonPads**
 - adding new buttons 140
 - creating new pads 140
 - custom Windows icons 217
 - defined 292
 - docking 145
 - help messages for buttons 144
 - ICONDEMO.MBX 142
 - PushButtons 138
 - ToggleButtons 138
 - ToolButtons 138
- Byreference parameters** 93
- Byvalue parameters** 93
- C**
- C language**
 - sample programs 261
- Callbacks** 237
- Calling external routines** 65, 212
- Calling procedures** 92
- CancelButton controls** 127
- Case sensitivity** 72
- Character sets** 182
- Checkable menu items** 114
- CheckBox controls** 126
- Choropleth maps** 132
- Circles See Objects** 13
- Class name**
 - MapInfo.Application 229
 - MapInfo.Runtime 230
- Clicking and dragging** 141
- Client/server**
 - database access 173
 - DDE protocol 218
- Close Window statement** 131, 224
- Color values**
 - RGB() function 191
 - selecting objects by color 191
- Columns**
 - alias expressions 153
 - Obj (object) column 155, 185
 - RowID column 155
 - syntax for reading 152
- Command line arguments** 62, 253
- CommandInfo() function**
 - ButtonPads 139
 - DDE 222
 - detecting doubleclick in list 128
 - detecting if user clicked OK in dialog 123
 - determining Find results 156
 - ID of selected menu item 117

- Comments** 72
- Commit statement** 136, 157
- Commit Table statement** 133
- Comparison operators** 84
- Compiler**
 - defined 293
- Compiler directives** 100
- Compiling a program**
 - from the command line 62
 - in the active window 51, 60
 - without opening the file 67
- Concatenating strings**
 - & operator 271
 - + operator 271
- Confirmation prompt** 121
- Connecting to a remote database** 173
- Connection handle**
 - defined 173
- Connection number**
 - defined 173
- Constants**
 - date 81
 - defined 78
 - logical 81
 - numeric 80
 - string 80
- Contains operator** 86, 206
- Continue statement** 104
- Continuous Thematic Shading support** 132
- Control panels, effect on date formatting** 81
- Controls in dialogs** 125
- Conventions** 16
- Coordinate systems**
 - earth coordinates 203
 - Layout coordinates 162, 203
 - nonearth coordinates 203
- Copying programs from Help** 53
- Cosmetic layer**
 - defined 293
 - deleting objects from 162
 - selecting objects from 162
- Create ButtonPad statement** 139, 141, 217
- Create Frame statement** 135, 193
- Create Index statement** 157
- Create Map statement** 157, 186
- Create Menu Bar statement** 116
- Create Menu statement** 113
- Create Text statement** 135, 189
- CreateCircle() function** 194
- Creating map objects** 193
- Crystal Report writer** 151
- CurDate() function** 80, 84
- Cursor (drawingtool icon)** 145
- Cursor (position in table)** 152

D

- Data structures** 76
- database live access** 176

Date constants 81
Date operators 84
DBF (dBASE) files 151
DDE, acting as client 218, 223
DDE, acting as server 223
Debugging a program 104
Decimal separators
 in numeric constants 80
Decisionmaking
 Do Case statement 88
 If...Then statement 87
Declare Function statement 99, 211
Declare Sub statement 92, 211
Define statement 100
Degrees, defined 293
Degrees to DMS 264
Deleting
 columns from a table 157
 files 180
 indexes 158
 menu items 112, 119
 menus 116, 119
 part of an object 199
Delphi, sample programs 261
Dialogs, custom
 control types 125
 disabled controls 128
 examples 123–124
 lists based on arrays 129
 lists based on strings 129
 modal vs. modeless 130
 positions of controls 124
 reacting to user's actions 128
 reading final values 127
 setting initial values 127
 shortcut keys 130
 sizes of controls 124
 terminating 130
Dialogs, standard
 asking OK/Cancel question 121
 hiding progress bars 148
 opening a file 122
 percent complete 122
 saving a file 122
 simple message 121
Dim statement 74
Directory names 180
Disabled
 defined 293
Distance units 204
DLLs
 declaring 211
 defined 211
 Kernel library 214
 passing parameters 212
 search path 211
 storing ButtonPad icons 217
 string parameters 213
 User library 212

DMS to Degrees 264
Do Case statement 88
Do...Loop statement 90
Dockable ButtonPads 145
Drawing modes 141
Drop Map statement 186

E

Edit menu 67
Editing target 199
Edits
 determining if there are unsaved edits (7.5) 277
EditText controls 125
Embedding 228
Enabled, defined 294
End Program statement 91
EndHandler procedure 96
EOF() function (end of file) 181
EOT() function (end of table) 152
Erasing a file 180
Erasing part of an object 199
Err() function 106
Error\$() function 106
Errors
 compiletime 60
 runtime 104, 156
 trapping 106
ERRORS.DOC 236
Events, handling
 defined 95
 selection changed 143
 special procedures 96
 userinterface events 109
Excel files 151
Execution speed, improving
 handler procedures 98
 table manipulation 176
 user interface 147
External references
 routines in other modules 65
 Windows DLLs 211

F

features list
 7.0 278
 7.5 278
 7.8 277
Fetch statement 152, 197
File extensions 15
File input/output
 binary file i/o 182
 character sets 182
 copying a file 180
 defined 179
 deleting a file 180
 random file i/o 182
 renaming a file 180
 sequential file i/o 180

File menu 66
FileExists() function 180
FileOpenDlg() function 122
Files, external
 BIL (SPOT image) 167
 DBF (dBASE) 151
 GIF 167
 JPG 167
 PCX 167
 Targa 167
 TIFF 167
 WKS (Lotus) 151
 XLS (Excel) 151
FileSaveAsDlg() function 122
Fill styles (Brush) 188
Findandreplace
 in MapBasic editor 68
 sample program 267
Finding a street address 156
Fixedlength string variables 75
Flowchart
 SQL MapBasic Server statements 174
Focus
 defined 294
 within a dialog 128
Folder
 defined 294
Font styles 188–189
FontPicker controls 126
For...Next statement 90
ForegroundTaskSwitchHandler procedure 96
Foreign character sets 182
Format\$() function 134
FoxBase files 151
Frame objects 193
FrontWindow() function 131
Function...End Function statement 99

G

Geocoding
 automatically 156
 interactively 156
 MapMarker 156
Geographic objects
 See Objects
Geographic operators 86, 205
Get statement (file i/o) 182
GetMetaData\$() function 170
GetSeamlessSheet() function 173
GIF files 167
Global variables 77
GoTo statement 89
GPS
 defined 294
GPS applications 133
Graduated symbol maps 132
Graph windows 135
Graticules (grids) 265

Grid Thematic support 132
GroupBox controls 125
H
Halting a program 91
Header files 15
Height of text 189
Help files, creating, for Windows 223
Help files, using 53
Help menu 70
Help messages for buttons 144
Hexadecimal numbers
 &H syntax 80
 defined 294
Hot keys
 in dialogs 130
 in menus 118
Hot links 223
HotLinks, querying (7.5) 277

I

Icons for ButtonPads 140, 216
Identifiers, defining 100
If...Then statement 87
Images (raster) 167
Include statement 100
Indexes, creating 157–158
Infinite loops, preventing 98
Info window
 customizing 136
 making readonly 137
Input # statement 181
Input/output, See File input/output
Insert statement 135, 157, 195
Inserting
 columns into a table 158
 nodes in an object 194
 rows into a table 157
Installation instructions 14
Integer division 271
Integer math 83
Integrated Mapping
 defined 294
 error trapping 235
 introduction 227
 MFC 254
 object model 244
 online Help 242
 printing 235
 reparenting document windows 231
 reparenting legend windows 232
 resizing windows 232
 sample programs 229, 261
 starting MapInfo 229
 stopping MapInfo 236
 system requirements 228
 toolbar buttons 233
 using callbacks 237

International character sets 182

Intersection

- area where objects overlap 196
- Intersects operator 86
- of two streets 156
- points where lines intersect 199

IntersectNodes() function 199

Intersects operator 86, 206

Introduction to MapBasic 49

J

Joining tables 208

JPG files 167

K

Kernel (Windows DLL) 214

Keyboard shortcuts 57

Kill statement 180

Kilometers 203

L

LabelFindByID() function 200

LabelFindFirst() function 200

LabelFindNext() function 200

Labelinfo() function 200

Labels

- converting to text 201
- in programs 89
- on maps 194, 199

Latitude

- defined 294

LayerInfo() function (7.5) 277

Layers

- adding/removing layers 132
- Cosmetic layer 162
- reading settings (7.5) 277
- thematic layers 132

Layout windows

- object coordinates 203
- opening 135
- treating as tables 162

Legend windows, managing 265

Length of an object 206

Like operator 84

Line Input # statement 181

Line numbers in a program 69

Line objects

- See Objects

Line styles (Pen) 188

Linked tables 175

- defined 294

Linker

- defined 295

Linking a project

- after selecting a current project 64
- from the command line 62
- without opening the file 67

ListBox controls 126, 129

Literal value

- defined 295

Live remote database access 176

Local variables 74

Logical operators 85

Longitude 295

Looping

- Do...Loop statement 90
- For...Next statement 90
- While...Wend statement 91

Lotus files 151

M

Main procedure 92

MakePen() function 190

Map Catalog 282

Map objects

- See Objects

Map projections 133

Map windows 132

- labeling 199
- reading layer settings (7.5) 277
- See Layers

MapBasic 7.8

- Create Cartographic Legend statement 277
- Export statement 277
- LegendInfo() function 277
- MGRSToPoint statement 277
- Object Snap statement 277
- Objects Pline statement 277
- PointToMGRS statement 277
- PrintWin statement 277
- Register Table statement 277
- Save MWS statement 277
- Set Cartographic Legend statement 277
- Shade statement 277
- TableInfo() variable 277
- WFS Refresh statement 277

MapBasic Window 73

MapInfo documentation set 16

MapInfo menus file 119

MapInfo Runtime

- launching through OLE 230

MapInfo Test Drive Center 18

MapInfo-L, archive 19

MAPINFOW.MNU file 119

MapMarker product 156

MBX file, defined 295

Memory limitations 59

Menus, customizing

- adding menu items 112
- altering a menu item 114
- altering the menu bar 116
- creating new menus 113
- MapInfo menus file (Macintosh) 119
- MAPINFOW.MNU file (Windows) 119
- removing menu items 112
- shortcut keys 118

Merging objects [196](#)

Message window [136](#)

Metadata [169](#)

Methods

Application object [247](#)

defined [295](#)

MBAApplication object [249](#)

Metric units [203](#)

MFC

getting started [254](#)

sample programs [261](#)

Microsoft Excel

DDE conversations [218](#)

worksheet files [151](#)

Mod (integer math) [83](#)

Mod operator [271](#)

Modal dialog boxes [130](#)

Module

defined [295](#)

Modulelevel variables [66](#)

Mouse events

choosing a menu item [111](#)

clicking and dragging [141](#)

doubleclicking on a list [128](#)

Moving an object [198](#)

MultiListBox controls [126](#), [129](#)

Multiuser editing [163](#)

N

Nodes, determining coordinates [199](#)

Nodes, adding [194](#), [199](#), [267](#)

Nodes, maximum number of [194](#)

NoSelect keyword [98](#)

Not operator [85](#)

Note statement [121](#)

Number of

nodes per object [194](#)

objects per row [187](#)

polygons per region [187](#)

sections per polyline [187](#)

selected rows [159](#)

Number of open windows [131](#)

NumberToDate() function [81](#)

Numeric constants [80](#)

Numeric operators [83](#)

O

Object Model [244](#)

Object variables [185](#)

ObjectGeography() function [187](#)

ObjectInfo() function [187](#), [190](#)–[191](#)

ObjectLen() function [188](#), [206](#)

Objects, creating

based on existing objects [196](#)

buffers [196](#)

creation functions [194](#)

creation statements [193](#)

storing in a table [195](#)

Objects, deleting [186](#)

Objects, modifying

adding nodes [194](#), [199](#)

combining [196](#)

erasing part of an object [199](#)

position [198](#)

storing in a table [195](#)

style [198](#)

type of object [198](#)

Objects, querying

coordinates [187](#)

styles [188](#)

types [187](#)

ODBC connectivity

data types supported [280](#)

OKButton controls [127](#)

OLE Automation [244](#)

defined [292](#)

OLE Embedding [228](#)

OnError statement [106](#)

OnLine Help, creating for Windows [223](#)

OnLine Help, using [53](#)

Open File statement [179](#)

Open Window statement [131](#), [224](#)

Opening a table [150](#)

Opening multiple files [65](#)

Operators

comparison [84](#)

date [84](#)

defined [79](#)

geographic [86](#), [205](#)

logical [85](#)

numeric [83](#)

precedence [86](#)

string [83](#)

Optimizing performance

handler procedures [98](#)

table manipulation [176](#)

user interface [147](#)

Or operator [85](#)

Order of evaluation [86](#)

P

Pack Table statement [155](#)

Page layouts [135](#)

Paper units [204](#)

Parameters

passing by reference [93](#)

passing by value [93](#)

Pattern matching [84](#)

PCX files [167](#)

Pen styles [188](#)

PenPicker controls [126](#)

Percentcomplete dialog [122](#)

Performance tips

handler procedures [98](#)

table manipulation [176](#)

user interface [147](#)

Perimeter() function 188
Pie charts
 in graph windows 135
 in thematic maps 132
Point objects
 See Objects
Point styles (Symbol) 188
Points of intersection 199
Points, storing in a remote database 175
Polygon overlay 209
Polyline objects
 See Objects
PopupMenu controls 126, 129
PowerBuilder
 sample programs 261
Precedence of operators 86, 274
Print # statement 181
Print statement 136
Procedures
 calling 92
 defined 92
 Main 92
 passing parameters 93
 recursion 94
 that handle events 95
Product training 54
Program organization 102
Progress bar
 defined 296
Progress bars, hiding 148
ProgressBar statement 122
Project files
 creating 64
 defined 62
 examples 63
 linking 64
Project menu 69
Projections
 changing 133
Properties
 Application object 246
 defined 296
 MBAApplication object 248
 MBAApplications collection 248
 MBGlobal object 249
 MBGlobals collection 249
Proportional data aggregation 209
PushButtons defined 138
Put statement (file i/o) 182

Q

QueryN tables
 closing 160
 opening 159
Quick Start dialog 146

R

RadioGroup controls 125
Random file i/o 179, 182
Raster underlay table
 defined 296
Raster underlay tables 167
ReadControlValue() function 127, 129
Reading another application's variables 222
Realtime applications 133
Records
 See Rows
Recursion 94
 defined 296
ReDim statement 75
Redistricting windows 136
Region objects
 See Objects
Relational joins 186, 208
Remarks 72
Remote data
 defined 296
Remote database access 173
Remote database live access 176
RemoteMsgHandler procedure
 DDE 222
RemoteQueryHandler() function 222
Remove Map Layer statement 132
Rename File statement 180
Report writer 151
Responding to events
 See Events, handling
Resume statement 106
Retry/Cancel dialog 163
RGB color values 191
Rightclick menus
 destroying 117
 modifying 117
RollBack statement 157
Rotating a graphical object 266
Row cursor, positioning 152
RowID 155
Rows in a table
 displaying in Info window 136
 inserting new rows 157
 row numbers (RowID) 155
 setting the current row 152
 sorting 156
 updating existing rows 157
RTrim\$() function 85
Run Application statement 146
Run Menu Command statement 118, 136
Running a program
 from MapInfo 51, 61
 from the development environment 69
 from the startup workspace 146
Runtime errors 104
Runtime executable
 launching through OLE 230

S

Sample programs
 integrated mapping **261**
Save File statement **180**
Scope of functions **99**
Scope of variables **78**
Scroll bars, showing or hiding **133**
Seagate Crystal Report writer **151**
Seamless tables **171**
Search menu **68**
Search path for DLLs **211**
Searchandreplace
 in MapBasic editor **68**
 sample program **267**
SearchInfo() function **143**
SelChangedHandler procedure **96, 143**
Select Case (Do Case) **88**
Select statement **186–187, 192, 205–207**
Selection
 changing **160**
 clicking on an object **143**
 querying **161**
Sequential file i/o **179–180**
Server MapBasic statements **173**
Set CoordSys statement **162, 203**
Set Event Processing statement **134**
Set File Timeout statement **164**
Set Format statement **81**
Set Map statement **132, 134, 194**
Set Redistricter statement **136**
Set Shade statement **132**
Set Table statement **172**
Set Target statement **199**
Set Window statement **132, 224**
Shade statement **132**
Sharing conflicts **163**
Shortcut keys
 in dialogs **130**
 in menus **118**
Shortcut menus
 destroying **117**
 modifying **117**
Simulating a menu selection **118**
Size limitations **59**
Size of text **189**
Snap to Node **237**
Sorting rows in a table **156**
Source code
 defined **297**
Speed, improving
 handler procedures **98**
 table manipulation **176**
 user interface **147**
SPOT image files **167**
Spreadsheet files, opening **151**
SQL Select queries **156**
Startup workspace **146**
Statement, defined **297**

Statement handle, defined **173**
Statement number, defined **173**
StaticText controls **125**
Status bar help messages **144**
 in Integrated Mapping **237**
Stop statement **104**
Stopping a program **91**
Storing points on an RDBMS table **282**
Storing points on remote databases **175**
Street addresses, finding **156**
String concatenation
 & operator **271**
 + operator **271**
String constants **80**
String operators **83**
String variables, fixed vs. variablelength **75**
StringCompare() function **85**
Structures **76**
StyleAttr() function **189–191**
Styles (Pen, Brush, Symbol, Font) **188**
Styles, comparing **189**
Sub procedures, See Procedures
Subselects **207**
Subtotals, calculating **156**
Symbol styles **188**
SymbolPicker controls **126**

T

Table names
 determining table name from number (7.5) **277**
Table structure
 determining how many columns (7.5) **277**
TableInfo() function **155, 173, 186**
TableInfo() function (7.5) **277**
Tables
 adding dynamic columns **158**
 adding permanent columns **158**
 adding temporary columns **158**
 based on spreadsheets and database files **151**
 closing QueryN tables **160**
 column expressions **152**
 component files **166**
 Cosmetic **162**
 creating **157**
 joining **208**
 Layout **162**
 making mappable **157**
 metadata **169**
 number of open tables **159**
 Obj (object) column **155, 185**
 opening **150**
 raster image tables **167**
 reading values **152**
 row numbers **155**
 Selection **159**
 structure, modifying **157**
 structure, querying **159**
 writing values **157**

Tables, querying
table information (7.5) **277**

Targa files **167**

Target objects **199**

Technical Support
automated fax support **19**
MapInfo Test Drive Center **18**
services **17–18**

TempFileName\$() function **180**

Text editors **61**

Text height **189**

Text objects **188, 198**
See Objects

Text styles (Font) **188**

Thematic maps **132**

Thousand separators
in numeric constants **80**

TIFF files **167**

Timeout
DDE settings **219**

ToggleButton defined **138**

Toolbars See
ButtonPads

ToolButtons defined **138**

ToolHandler procedure **96, 139**

ToolTips **144**

Totals, calculating **156**

Transparent fill
defined **298**

Trapping runtime errors **106**

TriggerControl() function **128**

Type conversion **83**

Type...End Type statement **76**

Typographical conventions **16**

U

UBound() function **75**

Ungeocoding **186**

Units of measure
area units **204**
distance units **204**
paper units **204**

Update statement **157, 194–195, 197**

Updating remote databases **175**

User (Windows DLL) **212**

User interface
ButtonPads **138**
dialogs, custom **123**
dialogs, standard **121**
menus **111**
overview **109**
windows **131**

Userdefined functions **99**

Userdefined types **76**

V

Variablelength string variables **75**

Variables

declarations **74**
defined **73**
global **77**
list of data types **74**
object variables **185**
reading another application's globals **222**
restrictions on names **74**
style variables **190**

Version 6.5, new features **278**

Vertices

See Nodes

Visual Basic

sample programs **229, 261**

Visual C++

getting started **254**
sample programs **261**

W

Warm links **223**

While...Wend statement **91**

Wildcards (string comparison) **84**

WIN.INI file

DDE timeout setting **219**
querying settings **214**

WinChangedHandler procedure **96**

WinClosedHandler procedure **96**

Window identifiers **131**

Window menu **70**

WindowID() function **131**

WindowInfo() function **131, 162**

Windows, customizing

Browser **134**
Graph **135**
Info window **136**
Layout **135**
Map **132**
Message **136**
Redistricter **136**
size and position **132**

Windows, querying

map window settings (7.5) **277**

WinFocusChangedHandler procedure **96**

Within operator **86, 206**

WKS files, opening **151**

Workspaces

startup **146**
using as sample programs **53**

Write # statement **181**

X

XLS files, opening **151**