

Programming In OpenScript

ToolBook II

INSTRUCTOR™



© 2000 click2learn.com, inc. All Rights Reserved.

The software described in this document is furnished under a Software License Agreement. Please read it thoroughly. The software may be used or copied only in accordance with the terms of this agreement.

Information in this document is subject to change without notice. No part of it may be copied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior written consent from click2learn.com, inc.

U.S. GOVERNMENT RESTRICTED RIGHTS

The Software and Documentation are provided with RESTRICTED RIGHTS. If you are an agency of the United States Government, the following clause applies to this license. The Software is commercial computer software developed at private expense. Software provided to the United States Government pursuant to solicitations issued on or after December 1, 1995 is provided with the commercial license rights and restrictions described elsewhere herein. All Software provided to the United States Government pursuant to solicitations issued prior to December 1, 1995 is provided with "Restricted Rights" as provided for in the Commercial Computer Software-Restricted Rights at FAR, 48 CFR 52.227-14 (JUNE 1987) or The Rights in Technical Data and Computer Software clause at DFAR, 48 CFR 252.227-7013 (OCT 1988), as applicable. Use, duplication or disclosure by the United States Government is subject to restrictions as set forth therein. The manufacturer is click2learn.com, inc., 110 -110th Ave. NE, Bellevue, Washington 98004-5840.

ToolBook, OpenScript, and click2learn.com are registered trademarks, and the click2learn.com logo, Ingenium, ToolBook II, ToolBook II Assistant, and ToolBook II Instructor are trademarks of click2learn.com, inc. ToolBook, click2learn.com, and OpenScript are registered in the United States of America and may be registered in certain other countries.

Microsoft and Windows are registered trademarks of Microsoft Corporation. Java is a trademark or registered trademark of Sun Microsystems, Inc.

Acrobat Reader Copyright © 1987–1998 Adobe Systems Incorporated. All rights reserved. Adobe and Acrobat are registered trademarks of Adobe Systems Incorporated, which may be registered in certain jurisdictions.



The ToolBook II Instructor team

Development Tim Barham, Arjay Cannata, Bryan Comstock, Charley Delaney, John Dietz, Dick Earl, Mike Florence, John Gossman, Carrie Groff, Tim Higgins, Ross Hunt, Joel Kittinger, Andrew Krois, Tod Mahony, Slade Mitchell, Ian Morrison, Michael Ormes, Claude Ostyn, and Erik Reitan

Software Testing Christian Becker, Mark Engel, Nevada Hamaker, Dick Hamilton, Aileen Ibershof, Curtis Laird, Edward Lau, Jonathan Seigal, and Juan Vera

Publications Jennifer Binford, Mary Anne Dane, Norman Gilinsky, Jeannie McGivern, Karyn Pallay, and Gaylene Zweigle

Special Thanks Brad Crain, Denny Dedmore, Rob Fink, Carter Knowles, Kathy Navarro, Rick Roche, Paul St. Pierre, Paolo Tosolini, Jan Utterstrom, and Ken Whiting



Contents

- 1** Finding the information you need
- 2** Understanding OpenScript programming
- 3** Learning OpenScript basics
- 4** Using the scripting tools
- 5** Handling messages, events, objects, and properties
- 6** Creating an OpenScript statement
- 7** Using variables, values, and text
- 8** Writing common scripts
- 9** Handling user events
- 10** Managing data
- 11** Using special effects
- 12** Using dynamic data exchange
- 13** Using dynamic-link libraries
- 14** Debugging OpenScript programs

Contents

Chapter 1

Finding the information you need

- Looking at what's inside 10
- Finding other resources 13

Chapter 2

Understanding OpenScript programming

- About OpenScript 16
- OpenScript development tools 17
- Comparing OpenScript with other languages 20
- Extending ToolBook II and OpenScript 22

Chapter 3

Learning OpenScript basics

- About scripting in ToolBook II 24
- Writing scripts 24
- The object hierarchy 28
- Objects and properties 33
- Types of handlers 38
- Entering and displaying data 39



Contents

1	Finding the information you need
2	Understanding OpenScript programming
3	Learning OpenScript basics
4	Using the scripting tools
5	Handling messages, events, objects, and properties
6	Creating an OpenScript statement
7	Using variables, values, and text
8	Writing common scripts
9	Handling user events
10	Managing data
11	Using special effects
12	Using dynamic data exchange
13	Using dynamic-link libraries
14	Debugging OpenScript programs

Chapter 4

Using the scripting tools

Using the script editor	44
Working in the Command window	54
Preventing common scripting errors	59

Chapter 5

Handling messages, events, objects, and properties

About messages in OpenScript	62
Working with system objects and properties	75
Defining user properties	77
Creating properties using scripts	79
Creating self-contained objects using notify handlers	84

Chapter 6

Creating an OpenScript statement

About OpenScript commands	90
Using OpenScript control structures	92
Using functions	100
Creating user-defined functions	102
Creating expressions with operators	104



Contents

1	Finding the information you need
2	Understanding OpenScript programming
3	Learning OpenScript basics
4	Using the scripting tools
5	Handling messages, events, objects, and properties
6	Creating an OpenScript statement
7	Using variables, values, and text
8	Writing common scripts
9	Handling user events
10	Managing data
11	Using special effects
12	Using dynamic data exchange
13	Using dynamic-link libraries
14	Debugging OpenScript programs

Chapter 7

Using variables, values, and text

Using variables	118
Assigning data types to variables	123
Using arrays	127
Working with literal values	134
Working with constants	140
Working with character strings	142
Working with lists	147
Referring to colors	149

Chapter 8

Writing common scripts

Designing navigation	154
Checking for results and errors	163
Creating scripts in a running application	166
Saving changes to applications	170
Creating libraries of handlers using system books	173
Initializing an application	181
Setting startup properties for ToolBook II	188

Chapter 9

Handling user events

Handling mouse events	190
Handling keyboard events	201



Contents

1	Finding the information you need
2	Understanding OpenScript programming
3	Learning OpenScript basics
4	Using the scripting tools
5	Handling messages, events, objects, and properties
6	Creating an OpenScript statement
7	Using variables, values, and text
8	Writing common scripts
9	Handling user events
10	Managing data
11	Using special effects
12	Using dynamic data exchange
13	Using dynamic-link libraries
14	Debugging OpenScript programs

Chapter 10

Managing data

Formatting data	214
Validating data	218
Reading and writing to files	223
Searching pages	228
Sorting pages	231

Chapter 11

Using special effects

Animating objects	234
Adding drag-and-drop behavior	249
Creating sound effects	267

Chapter 12

Using dynamic data exchange

About dynamic data exchange	270
Using ToolBook II as a client	272
Using ToolBook II as a server	279



Contents

1	Finding the information you need
2	Understanding OpenScript programming
3	Learning OpenScript basics
4	Using the scripting tools
5	Handling messages, events, objects, and properties
6	Creating an OpenScript statement
7	Using variables, values, and text
8	Writing common scripts
9	Handling user events
10	Managing data
11	Using special effects
12	Using dynamic data exchange
13	Using dynamic-link libraries
14	Debugging OpenScript programs

Chapter 13

Using dynamic-link libraries

About DLLs	284
Linking and declaring DLL functions	287
Allocating Windows memory	295
Calling Windows API functions	300
Translating Windows messages in ToolBook II	300
Writing DLLs to use with ToolBook II	309

Chapter 14

Debugging OpenScript programs

About debugging a ToolBook II script	322
Responding to the Execution Suspended message ...	328
Using the ToolBook Debugger	330
Using scripts for debugging	342

Index

349



- 1** Finding the information you need
- 2** Understanding OpenScript programming
- 3** Learning OpenScript basics
- 4** Using the scripting tools
- 5** Handling messages, events, objects, and properties
- 6** Creating an OpenScript statement
- 7** Using variables, values, and text
- 8** Writing common scripts
- 9** Handling user events
- 10** Managing data
- 11** Using special effects
- 12** Using dynamic data exchange
- 13** Using dynamic-link libraries
- 14** Debugging OpenScript programs

Chapter 1

Finding the information you need

Welcome to *Programming in OpenScript*,[®] the easy-to-use online guide that teaches OpenScript programming skills. This chapter provides an overview of what you will find in this guide, and describes the other resources available for learning about OpenScript.

CONTENTS

Looking at what's inside	10
Finding other resources	13



- 1** Finding the information you need
- 2** Understanding OpenScript programming
- 3** Learning OpenScript basics
- 4** Using the scripting tools
- 5** Handling messages, events, objects, and properties
- 6** Creating an OpenScript statement
- 7** Using variables, values, and text
- 8** Writing common scripts
- 9** Handling user events
- 10** Managing data
- 11** Using special effects
- 12** Using dynamic data exchange
- 13** Using dynamic-link libraries
- 14** Debugging OpenScript programs

Looking at what's inside

This online guide contains a wealth of information on OpenScript, the programming language that you can use to enhance the applications you create with ToolBook II Instructor.™ This guide provides a basic foundation for understanding OpenScript, and then builds on that foundation, chapter by chapter. In addition to this chapter, here is what you will find:

- ◆ **Chapter 2**, “Understanding OpenScript programming,” introduces OpenScript and provides an overview of the ToolBook II scripting tools.
- ◆ **Chapter 3**, “Learning OpenScript basics,” explains the fundamental concepts, vocabulary, and techniques you need to know to create scripts.
- ◆ **Chapter 4**, “Using the scripting tools,” describes how to use the ToolBook II scripting tools.
- ◆ **Chapter 5**, “Handling messages, events, objects, and properties,” teaches you how to work with built-in messages, system objects, and system properties, and how to create your own messages, user properties, and self-contained objects.
- ◆ **Chapter 6**, “Creating an OpenScript statement,” describes the components that make up a statement and the control structures you use to control blocks of statements.
- ◆ **Chapter 7**, “Using variables, values, and text,” focuses on how to use individual components of OpenScript programs, such as variables, constants, and literal values, and how to use text in your scripts.



Contents

- 1** Finding the information you need
- 2** Understanding OpenScript programming
- 3** Learning OpenScript basics
- 4** Using the scripting tools
- 5** Handling messages, events, objects, and properties
- 6** Creating an OpenScript statement
- 7** Using variables, values, and text
- 8** Writing common scripts
- 9** Handling user events
- 10** Managing data
- 11** Using special effects
- 12** Using dynamic data exchange
- 13** Using dynamic-link libraries
- 14** Debugging OpenScript programs

- ◆ **Chapter 8**, “Writing common scripts,” teaches you how to use OpenScript to accomplish many common tasks, such as checking for errors.
- ◆ **Chapter 9**, “Handling user events,” discusses how to work with events that a user generates by clicking a mouse button or typing at the keyboard.
- ◆ **Chapter 10**, “Managing data,” provides information on using OpenScript to perform several data-management tasks, such as validating user input.
- ◆ **Chapter 11**, “Using Special effects,” describes how to use OpenScript to include special effects, such as animation, in your application.
- ◆ **Chapter 12**, “Using dynamic data exchange,” explains how to send and receive data from other Microsoft® Windows® applications.
- ◆ **Chapter 13**, “Using dynamic-link libraries,” shows how to extend ToolBook II and OpenScript by calling Windows functions and other utility programs, and by translating Windows messages into ToolBook II messages.
- ◆ **Chapter 14**, “Debugging OpenScript programs,” presents tools and techniques for building more reliable applications.

Tip The chapter bar to the left of each page contains links to the chapters mentioned in this section, and to the main Table of contents and Index. For more information on how to navigate this book, click “How to use this online guide” at the top of the chapter bar.



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Identifying visual cues

To help you interpret the information it contains, this guide uses the visual cues listed in the table below.

Visual cues

Example format	Meaning
CTRL+ALT+DELETE	Keys separated by a plus sign indicate keys to press in combination.
<i>comboBox</i>	In a paragraph, italic type indicates OpenScript keywords.
button id 12	This font indicates an OpenScript code example.
\	A backslash indicates that an OpenScript statement continues on the next line.
↳	This symbol indicates that OpenScript syntax is continued from the previous line.
<isShift>	In a code example or syntax statement, angle brackets indicate a parameter that represents a literal value or expression.
--Initialize	In a code example, two hyphens precede a comment.
before after	In a syntax statement, vertical bars separate the options from which you can choose.
end [step]	In a syntax statement, square brackets indicate optional words and parameters.
...	In OpenScript syntax, an ellipsis indicates that the parameter can be repeated. In code examples, ellipses indicate the presence of any number of statements.
?	This icon is located by text that indicates where to look in online Help for more information.



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Finding other resources

In addition to this guide, there are a variety of other resources available for learning about OpenScript and ToolBook II. These are:

- ◆ the click2learn.com, inc. Web site, which contains a Knowledge Base, access to technical support, and information on sample files, patches, and newsgroups. You will also find information on other click2learn.com, inc. products and any upcoming developers' conferences. To visit the click2learn.com Web site, go to <http://www.click2learn.com/>.
- ◆ the click2learn FTP site, which contains sample files, updates, and utilities. You can visit the click2learn.com, inc. (formerly Asymetrix) FTP site at <ftp://ftp.asymetrix.com/>.
- ◆ the ToolBook II online Help system, which includes an OpenScript reference of objects, properties, keywords, and dynamic-link library (DLL) functions, and also a comprehensive glossary. To use online Help, press F1 from anywhere in ToolBook II, or select an item from the ToolBook II Help menu.



How to use this
online guide

Contents

- 1** Finding the information you need
- 2** Understanding OpenScript programming
- 3** Learning OpenScript basics
- 4** Using the scripting tools
- 5** Handling messages, events, objects, and properties
- 6** Creating an OpenScript statement
- 7** Using variables, values, and text
- 8** Writing common scripts
- 9** Handling user events
- 10** Managing data
- 11** Using special effects
- 12** Using dynamic data exchange
- 13** Using dynamic-link libraries
- 14** Debugging OpenScript programs



Contents

- 1 Finding the information you need
- 2 **Understanding OpenScript programming**
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Chapter 2

Understanding OpenScript programming

This chapter introduces OpenScript, the ToolBook II programming language, and describes how to use it to develop applications. This chapter also includes an overview of the tools you can use to create OpenScript programs, and a comparison of OpenScript and other popular programming languages.

CONTENTS

About OpenScript	16
OpenScript development tools	17
Comparing OpenScript with other languages	20
Extending ToolBook II and OpenScript	22



- 1 Finding the information you need
- 2 **Understanding OpenScript programming**
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

About OpenScript

OpenScript is the ToolBook II programming language. When you create applications in ToolBook II, you use ToolBook II built-in tools for tasks such as drawing objects on pages and creating hyperlinks. However, to take full advantage of ToolBook II capabilities, you use OpenScript to control the behavior of your application more directly and precisely.

OpenScript is a full-featured programming language that includes commands to accomplish a wide variety of tasks, from creating and managing new objects to linking functions in Windows dynamic-link libraries (DLLs). Though a powerful language, OpenScript is also easy to use because of its syntax (similar to English), its wide range of commands, and its object-oriented nature. Using ToolBook II and OpenScript, you can program sophisticated Windows applications in a fraction of the time and effort needed to create similar applications in C or C++.

OpenScript is fully integrated into the ToolBook II environment. You write scripts in the ToolBook II script editor or enter commands directly into the Command window. The programs are compiled using the ToolBook II compiler, and ToolBook II runs the finished scripts at Reader level as part of your application. Because OpenScript is an integral part of ToolBook II, the processes of creating, testing, and maintaining your programs are greatly simplified compared with the same processes in other programming languages.



Contents

- 1 Finding the information you need
- 2 **Understanding OpenScript programming**
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

OpenScript development tools

You enter OpenScript statements in the script editor or the Command window to create a script. One quick way to build a script is to copy an existing object that has the script you want (such as an object in a sample application), and then paste the object into your application.

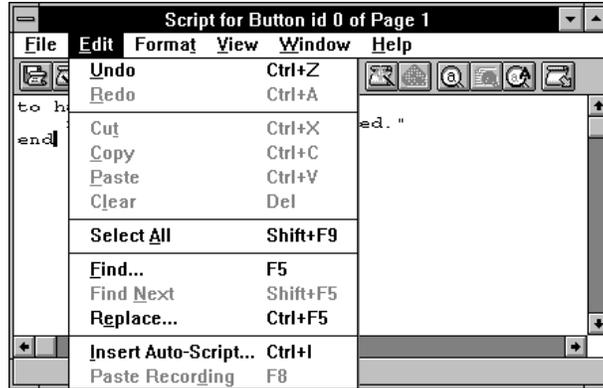
Working with the features of the Debugger, you can set breakpoints and trace the execution of statements in any script. On the following pages, you'll find a description of the Debugger as well as the other scripting tools.



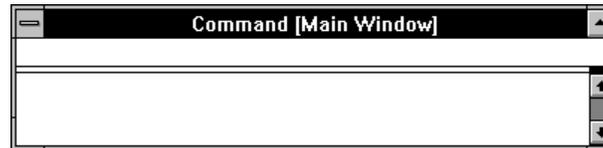
Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

- ◆ **Script editor** Use to type new scripts and handlers, edit existing scripts, or cut, copy, and paste script segments from the Clipboard. Editing a script in the script editor is much like editing a document in a simple word processing program. In addition, you can check a script's syntax, print the script, and display the ToolBook Debugger window.



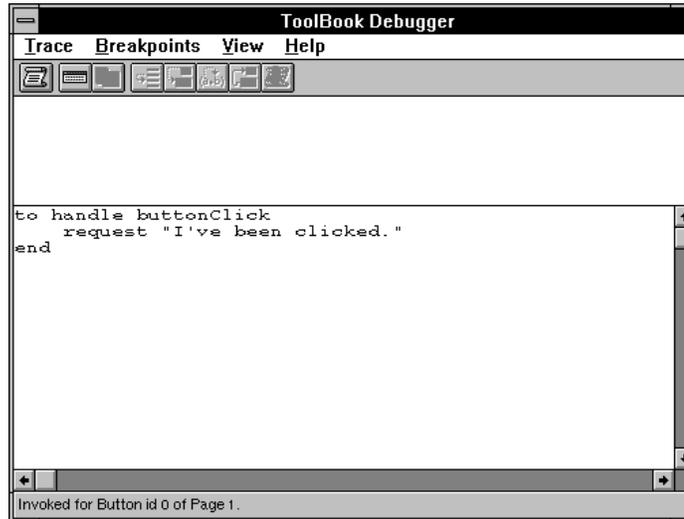
- ◆ **Command window** Use to send messages, display information, and test short script segments. You can enter one or more OpenScript statements in the Command window and execute them immediately.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

- ◆ **ToolBook Debugger window** Use to correct errors in your scripts. You can set and clear breakpoints, display the values of local and system variables, and trace execution of script statements and calls to other handlers.



You can use the sample applications provided with ToolBook II as a source of ideas, scripts, and objects for your own applications. Most sample applications include a button or special command on the Help menu to explain how the application works.

For details about using the script editor and Command window, see Chapter 4, "Using the scripting tools." For details about the ToolBook II Debugger, see Chapter 14, "Debugging OpenScript programs."



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Comparing OpenScript with other languages

If you have used another programming language, you will find that OpenScript provides virtually all the features you are familiar with from languages such as C, BASIC, and Pascal.

OpenScript compared with C, BASIC, and Pascal

OpenScript	C	BASIC	Pascal
<i>if/then/else</i>	if/else	If/Then/Else	If/Then/Else
<i>step</i>	for	For/Next	For/Do
<i>while</i>	while	While/Wend	While/Do
<i>do/until</i>	do/while	Do/Loop Until	Repeat/Until
<i>conditions/when</i>	switch/case	Select Case/Case	Case/Of
<i>increment</i>	+=		inc
<i>decrement</i>	-=		dec
<i>send <message></i>	void function	Sub	procedure
<i>to get <message></i>	function	Function	function
system books	libraries	libraries (or DLLs)	units
system variables	global variables	global variables	global variables
break	break, return		
format (number)	printf	print using	



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

The following list tells you where to look in this manual for more information on several important ToolBook II features.

- ◆ **Scope of variables** Variables can be local or global in OpenScript. For details, see Chapter 7, “Using variables, values, and text.”
- ◆ **Data typing** Variables are not typed by default, though you can assign data types for efficiency. If variables are not typed, you can use the data they contain alternately as strings and numbers. For details, see Chapter 7, “Using variables, values, and text.”
- ◆ **Arrays** You can define one-, two-, or many-dimensional arrays, or arrays containing variable numbers of elements. For details, see Chapter 7, “Using variables, values, and text.”
- ◆ **Data formatting** You can format numbers, dates, and times to alter how they are displayed and to make it easier to perform calculations with them. For details, see Chapter 10, “Managing data.”
- ◆ **User properties** Each object you create has built-in properties, and you can also define user properties to hold values that you create. For details, see Chapter 5, “Handling messages, events, objects, and properties.”
- ◆ **System books** You can establish any ToolBook II file as a system book, or library, making its functions and handlers available to any ToolBook II application. For details, see Chapter 8, “Writing common scripts.”
- ◆ **User-defined functions** In addition to using the built-in functions of ToolBook II, you can define your own functions to calculate values, perform operations, or carry out other tasks. For details, see Chapter 6, “Creating an OpenScript statement.”



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

- ◆ **Notify handlers** Using the *notifyAfter* and *notifyBefore* message handlers, you can create objects that you can put into any ToolBook II application without altering any scripts. For details, see Chapter 5, “Handling messages, events, objects, and properties.”
- ◆ **Dynamic script creation** Using the *execute* statement and *evaluate* function, you can create and execute scripts from within other scripts while your application is running. For details, see Chapter 8, “Writing common scripts.”

Extending ToolBook II and OpenScript

You can extend OpenScript by interacting with other instances of ToolBook II, other Windows applications, and even mainframe computers, by:

- ◆ using the Windows dynamic data exchange (DDE) protocol to incorporate the strengths of other Windows applications into your ToolBook II application. For example, an application might get the results of complex calculations from Microsoft Excel and display the results graphically in ToolBook II. For details, see Chapter 12, “Using dynamic data exchange.”
- ◆ using the *linkDLL* command to link a Windows DLL and call its functions. ToolBook II can use functions from any Windows DLL; you need to know what functions are in the DLL and the data types of their parameters, but after that you can use them like any ToolBook II function. For details, see Chapter 13, “Using dynamic-link libraries.”
- ◆ using the *translateWindow* command to translate Windows messages into ToolBook II messages so that your ToolBook II application can respond to events generated by the Windows system. For details, see Chapter 13, “Using dynamic-link libraries.”



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics**
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Chapter 3

Learning OpenScript basics

This chapter is a primer on OpenScript that explains the fundamental concepts, vocabulary, and techniques used to create scripts, and presents simple examples of all the OpenScript elements discussed.

The chapter assumes that you already know how to use the ToolBook II user interface to create objects and change their properties, and that you have a basic understanding of general programming concepts, including statements, variables, functions, and control structures (such as *if/then/else*). You can find details about how each of these elements is implemented in OpenScript later in this book.

CONTENTS

About scripting in ToolBook II	24
Writing scripts	24
The object hierarchy	28
Objects and properties	33
Types of handlers	38
Entering and displaying data	39



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 **Learning OpenScript basics**
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

About scripting in ToolBook II

ToolBook II is an object-oriented programming environment, which means that you build your application by creating objects using the ToolBook II user interface. For example, buttons, rectangles, and fields are all objects. They reside on pages and backgrounds, which are also objects, and all of these together reside in a book, which is an object as well. You set an object's properties to define its appearance (color, shape, text) and behavior (for example, what happens when the user clicks it).

Like Windows, ToolBook II is event-driven. ToolBook II waits for an event to occur, such as when a user presses a key, clicks the mouse, opens a ToolBook II file, chooses a menu item, moves the cursor across an object, or even does nothing (an *idle* event).

Each event creates one or more messages in ToolBook II. ToolBook II sends a message to the target, which is the object directly affected by the event. For example, when the user goes to a new page in the book, the *enterPage* message is sent to the new page, and the *leavePage* message is sent to the original page. When the user clicks an object, ToolBook II sends the *buttonClick* message to the object indicating that the user pressed and released the left mouse button. If the user is doing nothing, ToolBook II sends the *idle* message continuously to the current page.

Writing scripts

ToolBook II has built-in responses to many of the messages generated by events. For example, its response to the message sent when a user chooses a built-in menu command is to display a dialog box or perform an action.



Contents

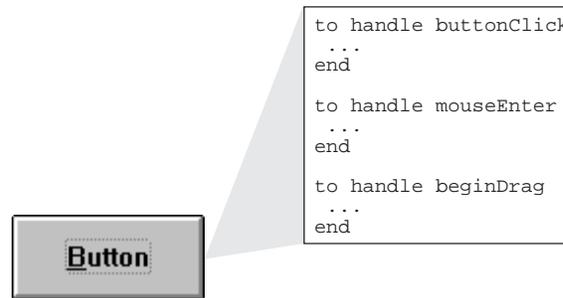
- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

For many events, you can rely on the built-in responses of ToolBook II. However, if you want to control directly how an object responds to an event, you create a handler for a message. A handler is a series of OpenScript statements that define the response to a particular message. For example, this handler defines how an object will respond when a user clicks it:

```
to handle buttonClick
  request "I've been clicked."
end buttonClick
```

Handlers are part of the *script* property of an object. A script can contain any number of handlers, each responding to a different message, as shown in Figure 1.

Figure 1 An object and its script



Any object in a ToolBook II application, including books, pages, and backgrounds, can have a script. To create a script, you use the script editor, which you invoke much the same way you do the dialog box for other properties of an object.

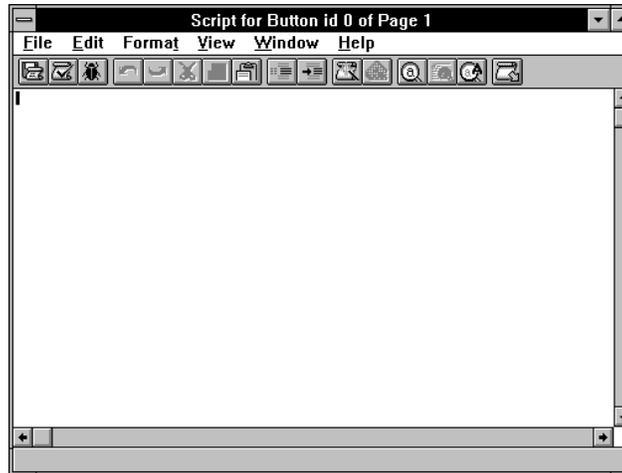


Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

► **To create a simple handler:**

- 1 In a new ToolBook II book, create a button on a page.
- 2 To display the script editor, right-click the button, and then click the Script button on the shortcut menu toolbar.



- 3 Enter this script:

```
to handle buttonClick
    request "I've been clicked."
end buttonClick
```

- 4 To save the script, choose Update Script & Exit from the File menu, or press CTRL+S.

When you save a script, ToolBook II automatically compiles it. If there is an error in the script, ToolBook II cannot compile it and displays an error message. You cannot run a script until it has compiled successfully.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

5 Switch to Reader level (press F3), and then click the button.

ToolBook II displays a message box with the text “I’ve been clicked.”

The handler you wrote above is for the *buttonClick* message, which is sent when you click the left mouse button. Without the script, the button does nothing when you click it; by adding this handler to the button’s script, you define the button’s response to the click.

The entire script for an object is limited to that which can be displayed in the script editor (64 KB). However, handlers can be converted to simple wrappers that can call more detailed scripts stored in other objects. For example:

```
to handle buttonClick
    send MyHandleButtonClick to page 2
end
```

In this script, the detailed script to handle the *buttonClick* message is handled in the *MyHandleButtonClick* handler on page 2.

Using the Command window to enter scripts

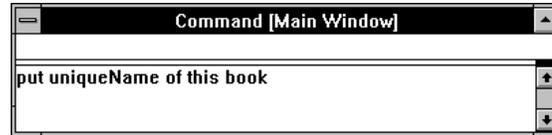
You can also run OpenScript statements at Author level using the Command window, shown in Figure 2 on the following page. You can type one or more OpenScript statements in the Command window to make an immediate change to your application or to see the effect of an OpenScript command. Use the Command window to quickly set properties for objects or to test statements as you build your application.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 **Learning OpenScript basics**
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Figure 2 The Command window



You can open the Command window by:

- ◆ choosing Command from the View menu.
- ◆ pressing SHIFT+F3.
- ◆ clicking the Command Window button on the toolbar.



For details about using the script editor and the Command window, see Chapter 4, "Using the scripting tools."

The object hierarchy

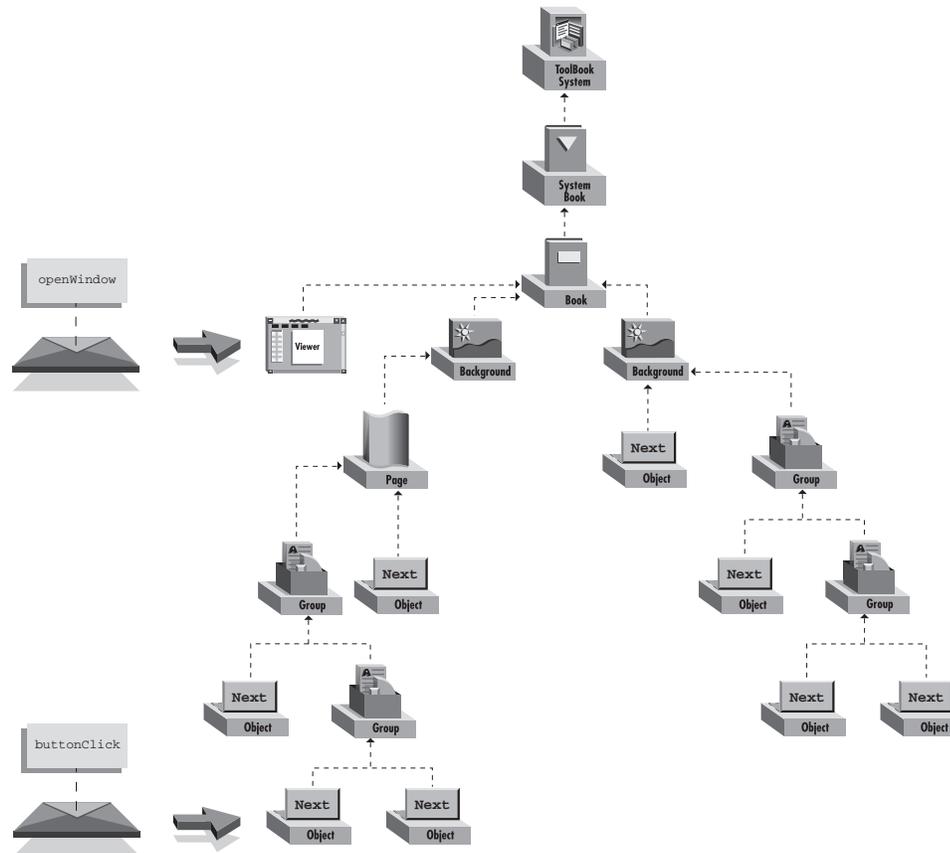
When an event occurs, ToolBook II sends the corresponding message to the target object. If the target object's script has a handler for the message, ToolBook II runs the handler and the message travels no further (unless the handler forwards the message, as explained later in this chapter). If the object does not have a handler for the message, the message passes to another object. The order in which a message goes from object to object is called the object hierarchy.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics**
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Figure 3 The object hierarchy



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

In the object hierarchy, a message passes from an object to its parent object. A page is the parent for objects and groups that reside on it, the background is the parent for the page, and the book is the parent for the background. If the object has no script for the message, the message moves further up the hierarchy until it either encounters the handler it is seeking or reaches the ToolBook II system at the top of the hierarchy. If the message reaches the system level, ToolBook II provides the default behavior (which might be to do nothing).

Messages enter the object hierarchy at different levels, depending on the message. For example, most mouse-event messages are sent to the object under the cursor. Menu-event messages enter the hierarchy at the page level, as do navigation messages such as *enterPage*, *leavePage*, *next*, and *previous*.

Note When a message enters the hierarchy at the page level, an object with a *notifyBefore* handler for that message will intercept the message before it starts at the page level. It's important to keep this in mind when writing self-contained objects. For more information on creating self-contained objects using notify handlers, see Chapter 5, "Handling messages, events, objects, and properties."

The following procedure demonstrates how the object hierarchy works. Before starting, create the button and script discussed in "Writing scripts" earlier in this chapter.

► **To show the object hierarchy:**

- 1 Add more objects to the same page: another button, a rectangle, and a circle.
- 2 Make sure no object is selected, and then click the Script button on the toolbar.



ToolBook II displays the script editor with the script for the page, which is the default when no objects are selected.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 **Learning OpenScript basics**
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

3 Enter this handler in the page's script:

```
to handle buttonClick
    request "Your click reached the page."
end buttonClick
```

4 To save the script, choose Update Script & Exit from the File menu, or press CTRL+S.

5 Switch to Reader level (press F3). Then, to see the results, click on each object and on a part of the page that contains no objects.

There are now two *buttonClick* handlers in your book: the one you created earlier for the button and the one you just entered as part of the page script. If you click the button with its own handler, ToolBook II runs the handler for that button. However, if you click an object without a handler for the mouse event, the message passes up the hierarchy to the page, where ToolBook II runs the handler you just entered.

Forwarding a message

If an object has a handler for a message, the message stops at that object. However, you can forward a message to send it further up the object hierarchy so that the message triggers additional handlers or the ToolBook II default response.

For example, if a button on the page handles a *buttonClick* message, the message stops with the button. If there is another handler for the *buttonClick* message higher in the hierarchy, the second handler will not run because it will never receive the message. If you want both handlers to run, use the *forward* command in a handler to pass the message up the hierarchy.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 **Learning OpenScript basics**
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Use the script editor to modify the script you created earlier for the button by adding a *forward* command:

```
to handle buttonClick
  request "I've been clicked."
  forward
end buttonClick
```

When you click the button at Reader level, you will get two messages: one produced by the handler you just modified and one produced by the handler you entered earlier for the page.

Forwarding a message is often necessary when ToolBook II already has a default response to a message, as for keystrokes. For example, you can capture keystrokes with a handler for the *keyChar* event in a page script. When you do, however, ToolBook II does not get the keystroke messages, and so it will not display the keystrokes unless you also forward the messages.

For example, create a field and ensure Enabled is selected on its right-click shortcut menu. Then, add this handler to the script of the field:

```
to handle keyChar
  beep 1
  forward
end
```

When you enter data into the field at Reader level, each keystroke causes a beep. In addition, the messages caused by the keystrokes pass up the object hierarchy to the ToolBook II system. The ToolBook II default response is to display the correct characters in the field. If you remove the *forward* statement, your handler runs, but ToolBook II does not receive the messages and no characters are displayed.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Using the object hierarchy effectively

You can take advantage of the object hierarchy to make writing and debugging scripts easier. For example, you can use the *forward* command to pass messages up the object hierarchy. This allows multiple handlers to respond to a single message. You can also put a message handler in a page or book script so that no matter what the target object is, you can handle the message with a single script. Finally, you can selectively disable events to prevent handlers at a higher level from taking effect by not including the *forward* command.

Objects and properties

Everything about an object, including its position, size, color, and script, is defined by a property. You can view and set many object properties using the ToolBook II user interface. For example, you can set the color of an object by clicking a tile in the color tray. However, you often get and set properties using OpenScript statements. For example, you can set the color of an object to red as a warning, or get the text of a field after the user enters data. This section explains how you refer to objects and properties using OpenScript statements.



For a complete list of ToolBook II objects and their properties, refer to the OpenScript reference in online Help.

Note ToolBook II has system objects with their own properties, such as the palettes. For details, see Chapter 5, “Handling messages, events, objects, and properties.”



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 **Learning OpenScript basics**
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Referring to objects in scripts

If you want to set or get the properties of an object in a script, you must identify the object by type and name. To identify the type, use OpenScript keywords such as *button*, *page*, and *roundedRectangle*. For example:

```
caption of button ID 3  
objectCount of page "Contents"  
bounds of roundedRectangle ID 1
```

To refer to a particular object, assign a name to the object in its Properties dialog box, and then use the name as follows:

```
to handle leavePage  
    hide field "User Tip"  
    go to page "Contents"  
    forward  
end
```

You must use quotation marks around an object name if it includes spaces or special characters other than an underscore. For example, you must use quotation marks around the name "*Planting Schedule*" but not around *Garden_Plan*. However, you should use quotation marks around names in all cases because ToolBook II runs faster if it does not have to determine whether the object name is a variable.

You can also refer to an object by its ID number, which is assigned by ToolBook II when you create the object, as in this example:

```
button ID 1 of page ID 0
```

When practical, however, you should use a name to refer to an object. If you copy and paste an object, its ID number changes because ToolBook II creates a new version of it, but its name remains the same.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Using implicit and explicit references

If the object you are referring to is on the current page (the page displayed in the current viewer), you can use an implicit reference to your object. An implicit reference identifies only the object type and name. This makes your scripts short and easy to read. However, if the object is on the background, on another page, or in another book, you need to use an explicit reference that includes the location of the object. The following statements illustrate explicit references:

```
--On the background  
bounds of recordField "Name" of background ID 1  
  
--On another page  
text of field "Name" of page "Introduction"  
  
--In another book  
script of group "clock" of page 3 of book "widgets.tbk"
```

To determine the correct explicit reference for an object, move the cursor over it at Author level. ToolBook II displays the explicit reference of the object in the status bar.

Every object has a *uniqueName* property that contains its explicit reference. The *uniqueName* is a formal object reference; when OpenScript commands and functions return the name of an object, the name is always the *uniqueName* property of the object.



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Referring to the current object

You can refer to the current object using the special terms *this*, *my*, *self*, *target*, and *selection*:

- ◆ **this** refers to the page or background displayed in the current viewer, or to the current book. For example:

```
bounds of recordField "Name" of this background  
pageCount of this book
```
- ◆ **my** and **self** in a handler refer to the object whose script contains that handler. For example:

```
request my uniqueName  
move self by 1000,1000
```

- ◆ **target** refers to the object that first received the message that triggered the current handler. For example, if the user clicks a pushbutton, that button is the target, even if the message is forwarded up the object hierarchy to the book. You can use *target* to write scripts that handle the same event for many different objects, as in this handler for a book script:

```
--Determines what type of object the user clicked  
to handle buttonClick  
    if target contains "button" then  
        request "You clicked a button."  
    end if  
    if target contains "field" then  
        request "You clicked a field."  
    end if  
end buttonClick
```

- ◆ **selection** refers to the currently selected object or objects. For example:

```
move selection by 2880,-2880
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Setting and getting the values of properties

You assign a value to a property of an object using the assignment operator (=) or the command *set* or *put*. All three methods accomplish the same task, so you can interchange them as you like. The following examples illustrate how to use all three:

```
fillColor of ellipse "face" = red
set fillColor of ellipse "face" to red
put red into fillColor of ellipse "face"
```

Many properties have constraints on the values you can assign to them. For example, the property *transparent* can only be set to the values *true* or *false*. If you attempt to assign an invalid value, an error results.

Except for ID numbers, the properties of objects are persistent. This means that the properties of an object are stored with and accompany the object when it is cut or copied. Some properties are read only; for example, a book's *pageCount* property. You can get the value of the property, but ToolBook II does not allow you to alter it.

Note If you misspell the name of a property when you assign a value to it, ToolBook II creates a user property with that name instead of reporting an error. For details, see Chapter 5, "Handling messages, events, objects, and properties."

To get the value of a property, use it in any expression:

```
if fillColor of ellipse "face" is red then
    fillColor of ellipse "face" = blue
end
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

You can also use a *get* command to put the value of a property into a special variable called *It*:

```
get fillColor of ellipse "face"      --Puts the value of fillColor into  
                                     --the variable It
```

For details about the variable *It*, see Chapter 7, “Using variables, values, and text.”

Types of handlers

The most common type of handler in ToolBook II responds to messages and begins with *to handle*. However, there are actually five types of handlers in ToolBook II:

- ◆ ***to handle*** responds to a message.
- ◆ ***to get*** returns a value for use in other handlers. Use this type of handler to create a user-defined function or to get the value of a property defined using OpenScript statements. For details, see Chapter 5, “Handling messages, events, objects, and properties” and Chapter 6, “Creating an OpenScript statement.”
- ◆ ***to set*** defines new properties with OpenScript statements. For details, see Chapter 5, “Handling messages, events, objects, and properties.”
- ◆ ***notifyBefore* or *notifyAfter*** defines a handler that is notified when a page receives a particular message. Use these handlers to create objects that can be copied into an application without having to change scripts anywhere in the book. For details, see Chapter 5, “Handling messages, events, objects, and properties.”



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Using parameters with handlers

A parameter is information that accompanies a message to provide more detail. For example, ToolBook II sends three parameters with a *buttonClick* message, two of which indicate whether the user was holding down the SHIFT or CTRL key when the button was clicked. You can include variables on the opening line of the handler to receive parameters:

```
--Displays a message if the user clicks while pressing SHIFT;  
--the variable's location, isShift, and isCtrl are parameters  
to handle buttonClick location, isShift, isCtrl  
    if isShift is true then  
        request "You clicked while the Shift key was down."  
    end if  
end buttonClick
```

Entering and displaying data

The user generally interacts with your application through the objects you create—clicking buttons, entering text into fields, or choosing menu items. At times, however, you might need a more direct way to get data from the user or to display text. For example, you might need to display the values of variables, display an error message, or prompt the user for a password. You can use the:

- ◆ Command window to enter OpenScript statements or display values.
- ◆ *ask* command to solicit text from the user.
- ◆ *request* command to display a message and prompt the user to click a button in response.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

- ◆ text of fields and record fields to display text or the values of variables.
- ◆ status bar, in which you can display any text.
- ◆ dialog boxes that you create to prompt the user for information.

Using the Command window to display information

You can execute OpenScript statements directly in the Command window; ToolBook II displays the results immediately. You can also use the Command window to view data. If you use the *put* statement without naming a container to put the data into, ToolBook II displays the data in the Command window:

```
put pageCount of this book      --Shows page count in  
                                --the Command window
```

Prompting for information with the *ask* command

To prompt the user to enter a single line of data, use the *ask* command, which displays a dialog box with OK and Cancel buttons. The user's response is returned in the variable *It*:

```
to handle enterBook  
    ask "Enter your name:"  
    if It is not null then  
        request "Welcome to the Tutorial," && It  
    end if  
    forward  
end enterBook
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 **Learning OpenScript basics**
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Asking a yes or no question with the *request* command

If you just want a quick yes-or-no answer from the user, use the *request* command to display a message and buttons for Yes and No. The text of the button the user clicks is returned in the variable *It*:

```
request "Do you wish to move to the next page?" with "Yes" or "No"  
if It is "Yes" then  
    send next  
end
```

Setting the text of fields and record fields

If you want to display information as part of the current page, set the text of a field or record field. The text is a property of the field or record field, so it retains its value until you change it again. For example, this handler shows the name of the current page in a record field, updating it as the user navigates from page to page:

```
to handle enterPage  
    text of recordField "Caption" = name of this page  
    forward  
end
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Setting the caption of the status bar

To display text in the status bar at the bottom of the ToolBook II main window, set its *caption* property. Using the status bar to display information is especially effective when you do not want to interrupt the flow of the application with a message box. To display the status bar at Reader level, press F12, or use the command *show statusBar* in a script or in the Command window. For example:

```
-- Identifies the clicked object in the status bar  
to handle buttonClick  
    caption of statusBar = "Current object name is" && target  
end
```

Creating dialog boxes

The most flexible way to collect information from a user is to display a dialog box. The dialog box can contain multiple fields, buttons, combo boxes, and other controls, and it can display default data and provide data validation of user entries. You can create viewers that act like dialog boxes.



For details about viewers, refer to the OpenScript reference and other viewer topics in online Help.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools**
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Chapter 4

Using the scripting tools

To make your scripting and application development easier, ToolBook II includes a script editor and the Command window. This chapter describes how to use these scripting tools and provides tips for preventing many common mistakes.

CONTENTS

Using the script editor	44
Working in the Command window	54
Preventing common scripting errors	59

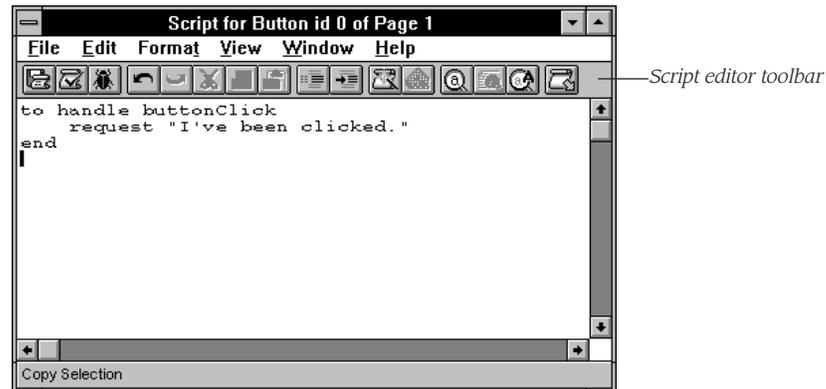


- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 **Using the scripting tools**
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Using the script editor

You can use the script editor to write, edit, check syntax for, and save the script of a selected object. Like the ToolBook II main window, the script editor has a toolbar with buttons for the most frequently used commands for writing and editing scripts.

Figure 1 The Script editor



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools**
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Buttons on the script editor toolbar

Default script editor button

Script editor button when CTRL key is pressed

	Save	
	Check Syntax	
	Debug	
	Undo	
	Redo	
	Cut	
	Copy	
	Paste	
	Comment	
	Indent	
	Find	
	Find Next	
	Replace	
	Parent	



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

You open the script editor at Author level by:



- ◆ selecting an object and then clicking the Script button on the toolbar.
- ◆ right-clicking an object and then clicking the Script button on the shortcut menu toolbar.
- ◆ selecting an object, choosing the appropriate Properties command from the Object menu, and then clicking the Script button on the script editor toolbar.
- ◆ typing *edit script of <object>* in the Command window.
- ◆ pressing CTRL and then double-clicking an object.



- ◆ in the active script editor, clicking the Parent button and choosing an object from the pop-up menu.

Tip To create a page script, press CTRL and click the status bar. To create a background script, press CTRL+ALT and click the status bar. To create a book script, press ALT and click the status bar.

Editing multiple scripts

You can open one script editor for each object you want to work with. Script editors are modeless, except when associated with shared scripts. Modeless script editors allow you to perform other ToolBook II tasks while they are open, and jump from one open script editor to another to work on the contents of several scripts simultaneously. To move between modeless script editors, click the one you want to work in, or choose one from the current script editor Window menu.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Tip You can display the script editor and modify a script at Reader level in two ways. You can execute the OpenScript statement *edit script of <object>* from the script of another object or from the Command window. Or, when Show right-click menus at Reader level is selected in the Options dialog box, you can right-click the object and click the Script button on the object's shortcut menu toolbar.

Note The ability to edit scripts at Reader level is disabled in the run-time version of ToolBook II.

Writing scripts

You can write a script for any ToolBook II object by opening the object's script editor and entering the script as you would with any text editor. Use these guidelines when you write a script:

- ◆ Lines can be any length, but lines over 80 characters are truncated when you print the script. Split a statement across more than one line by using the backslash (\) as a continuation character between expressions. (You cannot use a backslash in the middle of a string.) For example:

```
ask "Enter your complete address, including postal access code" && \  
    "and country of origin."
```
- ◆ Put multiple statements on one line by separating statements with semicolons (;).
- ◆ Indent handlers to make the script easier to read.
- ◆ If the previous line was indented, the script editor automatically indents to the same point when you press ENTER. Press BACK-SPACE to remove an indent.
- ◆ Insert comments in the script by typing two hyphens (--) before the comment text, or use block comments on selected text.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 **Using the scripting tools**
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

You can also write scripts by putting script text into the script property of an object. For details, see Chapter 8, “Writing common scripts.”

Using script editor commands

The following tables list the menu commands, toolbar buttons, and keystroke shortcuts you use in the script editor while writing scripts.

Use the following commands to move the insertion point.

Script editor commands for moving the insertion point

To move	Press
Any direction	Arrow keys
Left one word	CTRL+LEFT ARROW
Right one word	CTRL+RIGHT ARROW
Between the beginning of a line and the first non-space character of a line	HOME
To the end of a line	END
To the previous screen	PAGE UP
To the next screen	PAGE DOWN
To the beginning of the script	CTRL+HOME
To the end of the script	CTRL+END
To the top of the current screen	CTRL+PAGE UP
To the bottom of the current screen	CTRL+PAGE DOWN



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools**
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Use the following commands to select text.

Script editor commands for selecting text

To	Key command	Menu command
Select to the start of a line	SHIFT+HOME	
Select to the start of the script	SHIFT+CTRL+HOME	
Select a single word	Double-click	
Select a block of text	Click and drag	
Select the contents of the script editor	SHIFT+F9	Select All from the Edit menu
Extend a selection	SHIFT+arrow keys	
Cancel a text selection		Click elsewhere



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 **Using the scripting tools**
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Use the following commands to work with text and scripts.

Script editor commands for working with text

To	Key command	Toolbar button	Menu command
Undo changes	CTRL+Z		Undo from Edit menu
Reverse an Undo	CTRL+A		Redo from Edit menu
Indent line or selection to next TAB	CTRL+TAB		Indent from Format menu
Indent according to previous line	ENTER		
Move a line or selection back one TAB	CTRL+SHIFT+TAB		Outdent from Format menu
Find text	F5 or CTRL+F		Find from Edit menu
Find next instance of specified text	SHIFT+F5 or CTRL+N		Find Next from Edit menu
Search for and replace specified text	CTRL+F5		Replace from Edit menu
Print current script	CTRL+P		Print from File menu
Cut selection to Clipboard	CTRL+X		Cut from Edit menu
Copy selection to Clipboard	CTRL+C		Copy from Edit menu



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 **Using the scripting tools**
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Script editor commands for working with text, *continued*

To	Key command	Toolbar button	Menu command
Paste from Clipboard	CTRL+V		Paste from Edit menu
Import a text file into the script	F7		Import File from File menu
Export selected text to a text file	CTRL+F7		Export Selection from File menu
Write entire script to a text file	SHIFT+F7		Export Script from File menu
Make current line or selection into comments	CTRL+HYPHEN		Comment from Format menu
Uncomment current line or selection	CTRL+=		Uncomment from Format menu
Invoke Debugger	CTRL+D		Debug from File menu



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 **Using the scripting tools**
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Use the following commands to save changes to a script.

Script editor commands for saving scripts

To	Key command	Toolbar button	File menu command
Compile, update, and leave the current script	CTRL+S		Update Script & Exit
Compile and save without exiting	CTRL+U		Update Script
Compile, update, and save book	CTRL+B		Update & Save Book
Exit without saving changes	ALT+F4		Exit
Check the current script's syntax	CTRL+Y		Check Syntax

Use the following commands to perform additional functions.

Commands for performing additional functions

To	Menu command
Change display font	Font from View menu
Hide or show toolbar	Toolbar from View menu
Hide or show status bar	Status Bar from View menu



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Checking the syntax of a script

The syntax of a script must be correct before it will run. ToolBook II checks the syntax of a script automatically when you choose Update Script or Debug from the script editor File menu. You can also check the syntax of a script as you write it.

► To check the syntax of a script:



- 1 From the script editor File menu, choose Check Syntax. Or, click the Check Syntax button on the toolbar.

If ToolBook II finds a syntax error, it displays an error message and highlights the error. Otherwise, it displays the Syntax OK message. Assuming a syntax error is found:

- 2 Click OK to cancel the error message.

ToolBook II highlights the first syntax error it finds in the script.

- 3 Correct the error, and then choose Check Syntax again.

- 4 Continue checking and correcting the syntax of the script until ToolBook II displays the Syntax OK message.

For details about syntax errors, see Chapter 14, “Debugging OpenScript programs.”



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

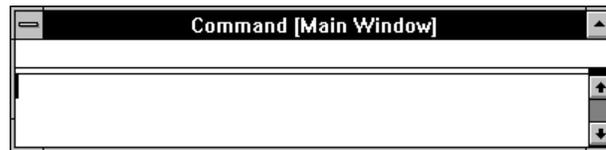
Working in the Command window

You can use the Command window to execute OpenScript statements without working in a script. You can also use shortcut keys to move commands or statements from scripts to the Command window. ToolBook II runs statements in the Command window as soon as you press ENTER. The Command window is available at both Reader level (if specified in the script of an object) and Author level, and from the ToolBook Debugger. For details about using the Command window from the ToolBook Debugger, see Chapter 14, “Debugging OpenScript programs.”

Use the Command window to test an OpenScript statement or handler, or to quickly get or set the value of a property, send event messages to objects, or send messages up the object hierarchy. Command window messages enter the hierarchy at the page level.

Tip Because you can immediately see what happens when statements are executed, typing commands in the Command window is a good way to become familiar with OpenScript and test parts of scripts.

Figure 2 The Command window



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 **Using the scripting tools**
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

You can use any of the following methods to display the Command window:



- ◆ Click the Command Window button on the toolbar.
- ◆ Choose Command Window from the View menu.
- ◆ Press SHIFT+F3.
- ◆ Use the *show commandWindow* statement in an OpenScript program.
- ◆ Use the *put* command without a target container in an OpenScript program.

You close the Command window as you close any other window. You can also move, resize, and maximize the Command window.

Running Command window scripts

ToolBook II executes script in the Command window within the context of the page displayed in the last active viewer (ignoring “always reader” viewers). This means that:

- ◆ viewer is the target window. For example, *this page* refers to the page displayed in the viewer.
- ◆ script executes as if it is in the page’s script. For example, *self* refers to the page.

The name of the current viewer is displayed in the Command window title bar.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Using variables in the Command window

By default, variables used in the Command window are untyped local variables that do not persist beyond the execution of the script in the Command window. You can, however, declare a variable as a system variable in the Command window. For the values that ToolBook II assigns to the special variables such as *It*, *my*, and *self*, see the following table.

Values of special OpenScript containers in the Command window

Variable	Value
<i>It</i>	Value set by the last OpenScript command executed in the Command window
<i>focus</i>	ID of object that had the focus at the time the Command window was displayed
<i>my</i> , <i>self</i> , <i>target</i>	<i>uniqueName</i> of <i>currentPage</i> of current execution context
<i>this page</i> , <i>background</i> , <i>book</i>	<i>uniqueName</i> of page, background, or book displayed in current execution context
<i>CommandWindow</i>	*Text entered in the Command window

Tip You can use the Command window to write script for an object without displaying the script editor. For example, create a button, select it, and then type the following in the Command window:

```
script of selection = "to handle buttonClick" & CRLF & \  
    "go next page" & CRLF & "end"
```

When you display the script for the button, the script text within the quotation marks appears.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools**
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Using the Command window to run and evaluate statements

Use the Command window to test commands before making them part of your scripts. Type an OpenScript statement in the Command window and then press ENTER. ToolBook II executes the statement.

To create multiline statements in the Command window, press CTRL+ENTER at the end of each line or use a semicolon (;) to separate statements.

After you are satisfied with the statements you have entered in the Command window, you can paste the contents of the Command window into the script of an object.

Figure 3 Multiline scripts in the Command window

A screenshot of a window titled "Command [Main Window]". The window contains a multiline script with the following text:

```
to handle enterPage
  system svFirstTime
  if svFirstTime = null then
    vWelcomeText = "Welcome to the Tutorial."
    text of field "banner" = vWelcomeText
    svFirstTime = false
  end if
end enterPage
```



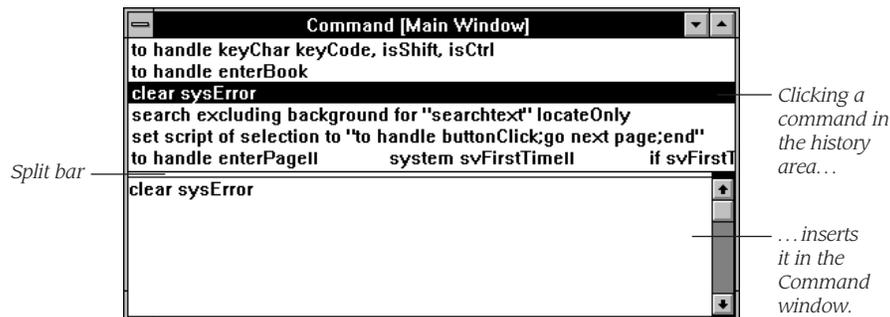
Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Command window history

The Command window keeps a history of the last 20 commands you entered in the current session. ToolBook II displays the most recent command in the history area above the area where you enter commands. To resize the history area and show more commands, drag down the split bar located just below the Command window title bar. Browse the commands in the history area using either the mouse or the PAGE UP and PAGE DOWN keys. Clicking a command in the history area inserts it in the Command window for you to edit or execute. Double-clicking a command in the history area executes it.

Figure 4 The Command window history area and split bar



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Debugging scripts in the Command window

To debug a Command window script, place it in the script of an object, and then debug that script. You cannot use the ToolBook Debugger to debug scripts in the Command window because Command window statements are read by ToolBook II as one line of script, which the debugger does not break down into intermediate lines.

For details about using the ToolBook Debugger, see Chapter 14, “Debugging OpenScript programs.”

Preventing common scripting errors

You can prevent certain common scripting errors and make the debugging process easier by following these suggestions:

- ◆ Use the Command window to execute statements so you can observe the results and determine which statements work properly.
- ◆ Ensure that ToolBook II displays the Execution Suspended message when an execution error occurs by setting the *sysSuspend* property to true (the default) while you are debugging a script. For details about debugging, see Chapter 14, “Debugging OpenScript programs.”
- ◆ Name objects consistently and descriptively. For example, if you have a page that contains a map of the United States, you might name the page *USMap*. If another page contains a map of Europe, you might name it *EuropeMap*.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

- ◆ Make certain you have named an object before referring to it by name in a script. Also, enclose names in quotation marks to speed execution and avoid any confusion with OpenScript keywords.
- ◆ Verify that the values for a function referred to in a script, whether literal or variable, are within the valid range. For example, if you use the square root function, you cannot specify *sqrt(-1)* because negative numbers are outside the valid range for this function.
- ◆ Do not use OpenScript keywords as variable names.
- ◆ Make certain that you declare and initialize variables before referring to them in a script.
- ◆ Use variable names that describe the variable's purpose. For example, if a variable holds the totals from a row in a spreadsheet, you might name the variable *rowTotal*. If another variable contains sales data from Japan, you might name this variable *japanSales*.
- ◆ Make sure that variable names are spelled correctly and consistently everywhere they occur in scripts. If you misspell the name of a variable, ToolBook II creates a new local variable with that name.
- ◆ Include parentheses in or around expressions to force evaluation in a particular order.
- ◆ Add comments to your scripts to explain the use of handlers, statements, and variables so that others can understand how your script works.
- ◆ Indent control structures when you write handlers to make the script easier to read.
- ◆ When you finish building an application, get rid of any unused code, old hidden objects, and obsolete user properties.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties**
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Chapter 5

Handling messages, events, objects, and properties

This chapter describes the fundamentals of OpenScript programming—handling messages, events, objects, and properties. Here you will learn to work with built-in messages, create your own messages, work with system objects and system properties, create user properties, and create self-contained objects that are automatically notified of events.

CONTENTS

About messages in OpenScript	62
Working with system objects and properties	75
Defining user properties	77
Creating properties using scripts	79
Creating self-contained objects using notify handlers	84



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 **Handling messages, events, objects, and properties**
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

About messages in OpenScript

ToolBook II has two kinds of messages: built-in and user-defined. Built-in messages are those ToolBook II sends automatically in response to an event, such as *buttonClick*, *enterField*, and *newPage*. User-defined messages are those you create and send.

Sending built-in messages

ToolBook II sends a built-in message when you perform the action corresponding to that message. For example, ToolBook II sends a *buttonClick* message when you click the left mouse button, and sends a menu-event message when you choose a menu item. ToolBook II sends different types of built-in messages, depending on what actions you performed, as explained in the following section.

You can also achieve the effect of a built-in message by sending the message explicitly with the *send* command. For example, you can cause a button to respond as if it has been clicked by running a command such as this one from the Command window:

```
send buttonClick to button "Next"
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Types of built-in messages

You can write handlers for these types of built-in messages:

- ◆ **Menu-event message** Sent to the current page when a menu item is selected from a menu at Reader or Author level. For example, if you select an object at Author level and choose Group Properties from the Object menu, the *properties* message is sent to the current page.
- ◆ **Mouse-event message** Sent at Reader level when the mouse button is clicked or the cursor moves over an object. For example, if a reader clicks a hotword in a field, ToolBook II sends *buttonDown*, *buttonUp*, and *buttonClick* messages to the hotword.
- ◆ **Keyboard-event message** Sent at Reader level when a keyboard key is pressed. For example, if a user presses ENTER while typing in a field, ToolBook II sends *keyDown*, *keyChar*, and *keyUp* messages to that field.
- ◆ **Enter/leave-event message** Sent at Reader or Author level to the viewer, page, background, book, or system when a user opens or closes an application, book, or viewer; navigates to another page; or clicks a menu. These event messages are also sent at Reader level to a button, combo box, or field when the object receives or loses the focus. For example, if a user presses TAB to move the focus to a field, the *enterField* message is sent to the field. If the user clicks another field, ToolBook II sends a *leaveField* message to the previous field and an *enterField* message to the new field. ToolBook II also sends a *mouseEnter* message to the new field.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

- ◆ **Notification message** Sent to notify the system of various events, such as when an object is created, destroyed, or moved. For example, if you paste a field into a book, ToolBook II sends the *make* message to the new field. If nothing is happening at Reader level, the *idle* message is sent repeatedly to the current page.
- ◆ **Drag-and-drop message** Sent before, during, and after a drag-and-drop operation. ToolBook II sends the *beginDrag* message to signal the start of the drag operation, and an *endDrag* message to indicate that the user has released the mouse button. During the drag operation, ToolBook II continually sends *enterDrop*, *stillOverDrop*, and *leaveDrop* messages to objects as the cursor passes over them.
- ◆ **Query message** Sent to an object to determine the object's state or the value of a particular property. For example, during a drag-and-drop operation ToolBook II sends an *allowDrop* query message to each object under the cursor in order to determine whether the object will accept a drop, which in turn determines whether ToolBook II displays the drag image or the no-drop image.
- ◆ **DDE-event message** Sent at Reader or Author level when information is exchanged between applications through dynamic data exchange (DDE). For example, if an instance of Microsoft Excel requests data from ToolBook II, the *remoteGet* message is sent to the current page.
- ◆ **Standard message** Sent at Reader level only to the ToolBook II system (not up the object hierarchy). For example, *editScript*.



For a complete list of messages, refer to the OpenScript reference in online Help.



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Sending user-defined messages

User-defined messages allow you to create scripts for specific tasks that are not already part of ToolBook II. Sending a user-defined message is like calling one of your own procedures or functions in another programming language.

You can send a user-defined message using the *send* command. You can also create menu items that send a user-defined message. In addition to creating the user-defined message, you must write a handler that responds to it.

For example, you might create an index page in your application, and then add a menu item named Index to the Page menu. ToolBook II automatically sends the message *index* to the page when a user chooses this menu item, or when you execute the command *send index* in a script. To define the response to the *index* message, write a handler like the following:

```
--Place this handler in the page script, the background script,  
--or the book script  
to handle index  
    go to page "Index" of book "overview.tbk"  
end
```

If you don't write a handler for a user-defined message, ToolBook II displays an error message:

- ◆ For menu-event messages related to a user-defined menu item, ToolBook II displays a message that there is no handler for the user-defined message.
- ◆ For other user-defined messages, ToolBook II displays the Execution Suspended message.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

When you create a user-defined message, make sure its name includes only letters and the underscore character, and that it is not an OpenScript keyword.



For details on keywords, refer to the OpenScript reference in online Help.

User-defined messages are particularly effective for creating modular applications because you can create individual handlers that are easier to code and maintain. For example, to streamline an *enterBook* handler, you can create separate handlers for different tasks, such as initializing menus, sizing windows, and establishing default values. Then, you can send user-defined messages from the *enterBook* handler to run these handlers, rather than including all of the statements required to do these tasks. The individual handlers will run automatically when the book is opened, or you can call them as needed with a *send* command.

```
to handle enterBook
  forward
  --The following are all user-defined messages
  send initMenus
  send sizeAppWindow
  send setSystemDefaults
end
```



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Placing handlers for built-in messages

You can place a handler for a built-in message in the target object's script or anywhere higher in the object hierarchy. Use these tips to help you decide where in the object hierarchy to place the handler for a specific message:

- ◆ Place handlers for built-in messages in the script of the target object or higher. For example, because *leavePage* is automatically sent to the page, don't put a *leavePage* handler in the script of a button.
- ◆ To program general behavior for several objects, place a message handler high in the object hierarchy. For example, if the handler defines behavior for several objects on a page, put the handler in the script for that page, or group the objects and place the handler in the script of the group. This strategy allows you to change, add, or remove the component items of the page or group without changing the scripts of the individual objects.

For example:

```
--Handler that asks for confirmation each time the user
--navigates to a new page; place this handler in a book script
to handle next
    request "Are you sure you want to proceed?" with "Yes" or "No"
    if It is "Yes" then
        forward
    end if
end next
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

- ◆ To program specific behavior for one object, or to cause various objects to respond differently to the same message, put the handler in the script of the target object. Or, put the handler in the page or book script, and then use the *target* property to distinguish which object received the message using a control structure.

For example:

```
--Handles navigation for buttons named "Next" and "Previous"
--and for hotwords whose name matches that of a page
to handle buttonClick
  conditions
    when object of target is "button"
      send (name of target)
    when object of target is "hotword"
      go to page (name of target)
    else
      forward      --User clicked some other object
  end conditions
end buttonClick
```

- ◆ To define both specific and general behavior for an object, put a handler in the target object's script to define specific behavior and forward the message. Then, put a handler for the same message higher in the message hierarchy to define general behavior. To allow a message to get to the ToolBook system while bypassing all scripts higher in the message hierarchy, use *forward to system*.

Note You can also create self-contained objects that are notified automatically of messages that are forwarded to the page they are on. For details, see "Creating self-contained objects using notify handlers" later in this chapter.



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Sending messages from scripts

Use the *send* or *forward* command to send a message regardless of whether the event that usually triggers the message occurred. By sending messages from scripts, you can:

- ◆ run a handler for a user-defined message.
- ◆ execute scripts of hidden objects.
- ◆ handle a message more than once.
- ◆ send menu-event messages without choosing the menu item.
- ◆ send mouse or keyboard messages at Author level.
- ◆ send enter- and leave-event messages (such as *enterBook* or *enterSystem*) without actually navigating.
- ◆ streamline handlers so they are easier to maintain.

Use the *send* command to run a handler in the current object's script or send a message to any other object, including objects lower in the hierarchy or in another book. For example, *send buttonDoubleClick to selection* runs the corresponding handler in the script of the selected object, and *send buttonClick to this page* runs the corresponding handler in the page script. When you send a message with the *send* command, ToolBook II sets the *target* property to the *uniqueName* of the object you sent the message to.

Use the *forward* command to pass the current message to the next highest object in the hierarchy. For example, the *forward* command in an *enterBook* handler in a page script sends the *enterBook* message to the background. Use *forward to system* to forward a message directly to the ToolBook II system, skipping any other objects in the object hierarchy. Forwarding a message does not change the *target* property.

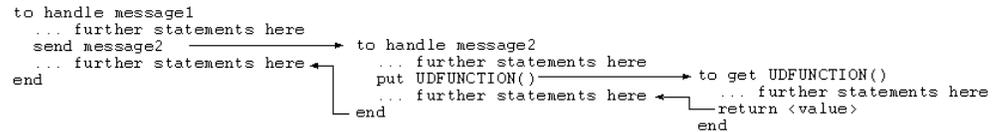


Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

A handler with a *send* or *forward* statement that runs another handler is referred to as a calling handler. The handler that is invoked is the called handler. When ToolBook II encounters a *send* or *forward* statement in a script, it suspends the calling handler until the called handler is finished, and any additional handlers that the called handler in turn calls. Eventually, control returns to the calling handler.

Figure 1 Returning control to the calling handler



Tip Sending a message to a specific object is faster than allowing the message to travel up the hierarchy.

Sending message names as variables

You can pass the name of a message as a variable. To force ToolBook II to recognize that you want the contents of the variable sent, not the variable name, put parentheses around the name of the variable that contains the message to be sent. For example:

```
ask "What message do you want to send?"
send (It)
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Changing or disabling built-in messages

Because many messages move up the object hierarchy before they reach the ToolBook II system, you can change a message or disable it altogether. For example, you can trap a keyboard-event message by writing a handler for it, and if you do not forward the message, ToolBook II doesn't ever receive and react to the message. For example:

- ◆ You can trap the *author* message and display a message instead of actually switching to Author level:

```
--Traps the author message and asks for a password before
--changing to Author level
to handle author
    ask password "Enter password for Author level:"
    if It is "uAKmKpw" then --The encrypted form of "ToolBook"
        forward
    end if
end author
```

- ◆ You can trap the right mouse click and disable that event:

```
--Traps and disables the rightButtonDoubleClick message
--(no forward here, so the handler stops the message)
to handle rightButtonDoubleClick
end
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

- ◆ You can trap keyboard events and substitute a character (for example, an uppercase letter for a lowercase one) before forwarding the message:

```
to handle keyChar key, isShift, isCtrl
  --The uppercase() function requires a letter, but key values
  --are passed as numbers, so they must be converted
  uppercaseKey = uppercase(ansiToChar( key ))
  key = charToAnsi(uppercaseKey)
  forward      --To display the character
end
```

Using parameters with messages

ToolBook II sends some messages with parameters that contain additional information about the message. For example, ToolBook II includes a parameter for the mouse position with the built-in message *buttonClick*. You can add parameter variables to handlers to capture this additional information. For example:

```
--"Position" is a parameter variable containing the location
--of the cursor when the button was clicked
to handle buttonClick pos
  request "You clicked at " && pos
end
```

When you send a built-in or user-defined message with the *send* command, you can send values or variables as parameters for the message. Parameters are always placed after the message name and multiple parameters are separated with commas:

```
--Sends a user-defined message with three parameters
send specialEffects location, color, grayTone
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

In the handler for the message, put the parameter variables in the same order, so you have access to the values that were passed:

```
--Handler for a user-defined message with three parameters  
to handle specialEffects location, color, grayTone
```

Naming variables for parameters

Parameters are placeholders, so only their position in the *to handle* statement is important. When you create a parameter variable, you can give it any valid variable name—it need not have the same name as the parameter in the calling routine, only the same position. For example, the first statement works as well as the second:

```
to handle specialEffects location, color, grayTone  
to handle specialEffects place, colorCode, gradation
```

Scope of parameters

A parameter is passed by value—that is, it is a copy of the value in the calling handler. Therefore, parameters become local variables in the handler to which they are sent. If you change the value of a parameter in the called handler, it does not affect the value of the corresponding parameter in the calling handler.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Declaring data types for parameters

You can add data-type declarations to parameters in a *to handle* statement. As with any variable for which you declare a data type, this can make the parameter more efficient because it requires less memory and ToolBook II can access the data faster. When you declare a data type for a variable, ToolBook II converts the parameter variables to the appropriate type, and enforces that data type for the rest of the handler. For example:

```
to handle buttonClick point loc, logical isShift, logical isCtrl
    loc = mousePosition of this window
    loc = "abc"      --Error (loc has been typed as a point, not a string)
    ...             --Further statements here
end
```

For details about data types, see Chapter 7, "Using variables, values, and text."

Counting and listing parameters

Besides referring to a parameter by its name, you can refer to a parameter in several other ways:

- ◆ Use the special variable *argList* to get a list of the parameters passed, and then refer to a parameter as an item in *argList*:

```
put item 3 of argList into field "MyList" of this page
```
- ◆ Use the special variable *argCount* to get the number of parameters passed to determine (for example) that all of the parameters you intended to pass were passed.
- ◆ Use *itemCount(argList)* to get the number of items passed with a message, which is different from using *argCount*. For example, if a message has a location parameter, the parameter will contain a list of two numbers. In this case, *itemCount(argList)* returns 2, but the value of *argCount* is 1.



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Working with system objects and properties

In addition to the objects that you create, such as buttons, fields, pages, and backgrounds, ToolBook II contains a number of system objects that belong to the ToolBook II system, but which you can manipulate to change the way ToolBook II looks and behaves. Examples of system objects are the Command window, tool palette, and polygon palette.

ToolBook II system objects differ from objects such as buttons and pages because you cannot create or destroy system objects. However, they have properties you can get or set. For example, you can hide and show these objects, set the caption of the status bar, and so forth. The object type names for the ToolBook II system objects are:

<i>colorTray</i>	<i>patternPalette</i>	<i>statusControls</i>
<i>commandWindow</i>	<i>polygonPalette</i>	<i>statusIndicators</i>
<i>lineEndsPalette</i>	<i>statusBar</i>	<i>toolBar</i>
<i>linePalette</i>	<i>statusBox</i>	<i>toolPalette</i>

Note Only the *statusBar* system property is available in the run-time version of ToolBook II.

System properties define default behavior for the entire ToolBook II system or identify objects that are currently receiving messages. By changing the values of certain system properties you can set default properties for new objects, set margins and other printer properties, or format system-wide values such as the current date and time.

For example, if you set the system property *sysFontStyle* to *bold*, the text in new fields and buttons is automatically bold. By getting the value of other system properties such as *target*, *focus*, and *selection*, you can determine which object is the target for user events.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

There are five types of system properties:

- ◆ **General system properties** These properties affect every book opened during the current ToolBook II instance. For example, if you set the *sysTimeFormat* property in the *enterBook* handler for the first book opened in a ToolBook II instance, that format will be used in all other books opened in that instance until another statement resets the format.
- ◆ **Event-focus properties** The *target* property indicates the object that receives the current message. The *targetWindow* property indicates the viewer that displays that object. The *focusWindow* property indicates the viewer that is currently active (which may or may not be the same as *targetWindow*).
- ◆ **Printer system properties** These properties affect various aspects of how a book prints. For example, if you set the *printerBorders* property to *true*, all pages and reports printed from that book will have a border around them.
- ◆ **International system properties** These properties can be used to automatically adapt books to the local conventions of specific countries. For example, the value of *sysDecimal* specifies which character separates the fractional part of formatted numbers.
- ◆ **Startup system properties** These properties affect all books in all instances of ToolBook II. When you set a startup system property, ToolBook II changes a value in the *Tb70.ini* file.



For details about the properties you can get and set for a specific system object, and for a list of all system properties, refer to the OpenScript reference in online Help.



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Defining user properties

You can assign a user property to an object to extend the kind of data the object can contain. For example, you can create a *lastScore* property for a book to store the final score in a game application. You can use user properties as global containers, much like system variables. However, unlike system variables, user properties are retained when you close the book and are associated with only one object.

Tip Data stored in user properties takes up space in a book; therefore, you should not include a lot of user properties if disk space is a consideration in your application.

Creating a user property

You create a user property by using it in an OpenScript statement as you would any built-in property. If the property you name does not already exist, ToolBook II automatically creates it and sets it to the value specified:

```
userScore of button "Play" = 0
```

Note If you misspell the name of a built-in property when setting its value (for example, *set fillColor to blue* instead of *set fillColor to blue*), ToolBook II will create a user property instead of reporting an error.

When you refer to a user property, you must always include a reference to the object that owns the property. If you omit the object reference, ToolBook II creates a local variable instead of a property. For example:

```
userScore of this book = 190    --Defines a user property  
userScore = 190                --Defines a local variable
```



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Getting the value of a user property

You get the value of a user property the same way you get the value of a built-in property. For example, the following statement gets the value of a *userScore* property:

```
get userScore of button "Play"      --Puts value into It
```

Listing user properties

You can use the keyword *userProperties* to list the properties you have created for any object. The most recently assigned user property appears at the front of the list. For example:

```
--Puts list of user properties into the Command window  
put userProperties of button "Play"
```

Deleting a user property

To delete a user property and remove its name from the list *userProperties*, you can set the property value to null or use the *clear* command.

For example, to set a user property value to null:

```
userScore of button "Play" = null
```

To clear a user property:

```
clear userScore of button "Play"
```

Note To recover the space used by a user property that has been deleted, choose *Save As* from the File menu or run the OpenScript *save as* command and give the book a new name. This reorganizes the book to its most efficient size.



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Creating properties using scripts

You can create properties with values determined by the handlers you write. The handlers are called when you use the *set* or *get* commands, just as if you were setting or getting a built-in or user property. However, instead of setting or getting a property value, ToolBook II calls your handler, which can set values, modify objects, or run any other process that your application requires. Using this technique, you can create a property that:

- ◆ filters or converts characters, such as a *ucText* property that returns the uppercase version of the text of a field.
- ◆ checks for specific values when you assign them to an object, such as a *dateText* property that sets the text of a field but only allows valid dates.
- ◆ sets a group of properties to a predetermined value, such as a property that resets a number of objects to a single color, or a menu to a default configuration.
- ◆ calculates a value using DDE calls to another Windows application.
- ◆ adjusts visual elements of the application as its value changes, such as the bars on a chart.

To create a property using a script, write a *to set* handler. When you use a *set* command to assign a value to a property, ToolBook II determines if the property is built-in. If so, ToolBook II assigns the value to that property. However, if the property is not built-in, ToolBook II sends a message to the target object and then up the object hierarchy. ToolBook II runs the *to set* handler that matches the property name. If there is no corresponding *to set* handler, ToolBook II creates a user property and assigns a value to it.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Creating a *to set* handler

The syntax for a *to set* handler is:

```
to set <propertyName> [parameters] to <value>
    ...      --Further statements here
end
```

When you call the handler with a *set* command, the value that you assign to the property is passed to the *to set* handler as the *<value>* parameter. For example, the following *to set* handler defines a property called *texture* for a filled object (such as an ellipse) or for the background. Possible values for the *texture* property are *bricks*, *waves*, *hearts*, or *none*. Based on the value you assign to the *texture* property, the *to set* handler changes the pattern and fill color of the target object to a new value.

```
to set texture to vTextureName
    validObjects = "ellipse,rectangle,polygon,pie," & \
        "background,irregularPolygon,roundedRectangle"
    --Check that this is a valid object type
    if object of target is not in validObjects
        break
    end if
    conditions
        when vTextureName is "bricks"
            pattern of target = 105
            fillColor of target = red
        when vTextureName is "waves"
            pattern of target = 94
            fillColor of target = blue
```

(continued)



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

```
when vTextureName is "hearts"  
    pattern of target = 114  
    fillColor of target = 300,50,100    --Pink  
when vTextureName is "none"  
    pattern of target = 254            --None  
    fillColor of target = white  
end conditions  
end texture
```

To call this handler, use a *set* command such as:

```
texture of ellipse "circle" = "bricks"
```

Placing *to set* handlers in the object hierarchy

You can place *to set* handlers in the script of a target object or anywhere higher in the object hierarchy. Use the same guidelines for placing the *to set* handler as you use for other handlers—for example, if you are creating a general-purpose handler, place it in the script of a book or system book. You can identify the object for which you are assigning the property using the *target* property.

Note When you write a *to set* or *to get* handler for a property, do not refer to that property in the body of the handler. If a statement in the handler tries to set or get the same property, ToolBook II calls the handler again, creating an infinite loop.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Getting a property's value with a *to get* handler

To define the value of a property created with a *to set* handler, create a corresponding *to get* handler. The syntax for a *to get* handler is:

```
to get <propertyName> [parameters]
  --(calculate value here)
  return <value>
end
```

The following example shows the *to get* handler that corresponds to the *to set texture* handler shown earlier:

```
to get texture
  validObjects = "ellipse,rectangle,polygon,pie," & \
    "background,irregularPolygon,roundedRectangle"
  --Check that this is a valid object type
  if object of target is not in validObjects
    break
  end if
  conditions
    when pattern of target = 105 and fillColor of target = red
      retVal = "bricks"
    when pattern of target = 94 and fillColor of target = blue
      retVal = "waves"
    when pattern of target = 144 and fillColor of target = 300,50,100 --Pink
      retVal = "hearts"
    when pattern of target = 254 and fillColor of target = white
      retVal = "none"
    else
      retVal = pattern of target & "," & fillColor of target
    end conditions
  return retVal
end texture
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

As with a built-in property, ToolBook II will invoke the *to get* handler whenever you get the value of the property. For example:

```
request texture of ellipse "circle"
```

or

```
text of field "display" = texture of ellipse "circle"
```

If there is no *to get* handler for a property created with a *to set* handler, ToolBook II returns *null*. ToolBook II searches the hierarchy for a *to get* handler the same way it searches for a *to set* handler, so it's a good idea to keep these together in the same script.

Tip Remember that if you provide *to get* and *to set* handlers, ToolBook II will call the *to get* handler whenever it needs to get the property, and the *to set* handler whenever it needs to set the property. This happens in situations that are not immediately obvious. For example, consider a *brightness* property that changes the brightness of an object:

```
to get brightness
    return item 2 of fillColor of target
end
to set brightness to value
    --maximum brightness is 100, minimum is 0
    if value > 100
        value = 100
    else
        if value < 0
            value = 0
        end
    end
    item 2 of fillColor of target = value
end
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

If you execute a command such as:

```
increment brightness of ellipse "light"
```

ToolBook II will call the *to get* handler to get the value of *brightness*, increment it by one, and then call the *to set* handler to set it again.

Note You can also use *to get* handlers to create user-defined functions. For details, see Chapter 6, "Creating an OpenScript statement."

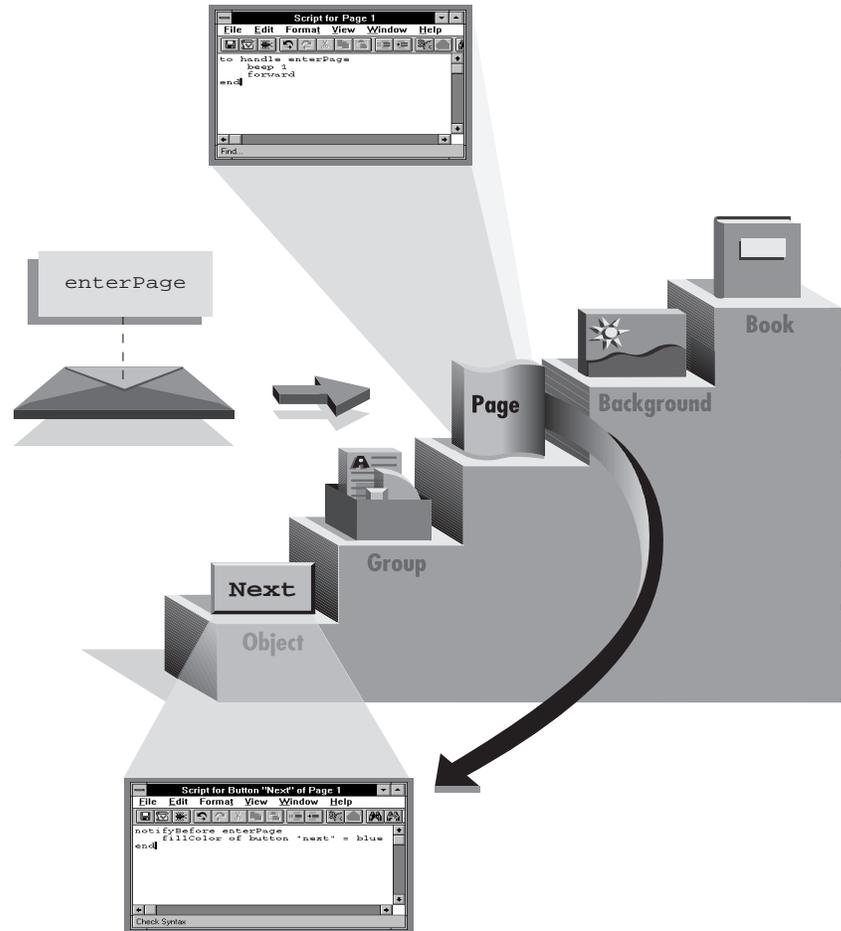
Creating self-contained objects using notify handlers

A notify handler is one that is automatically notified when the message it handles reaches the page. Usually, objects receive messages only if they are the targets of a user action or of the *send* command, or if another handler forwards the message to them. However, if an object contains a notify handler, the page sends a copy of the message to that object. For example, you can create a "Next" button that is automatically notified of any *enterPage* event. If the user navigates to the last page of the book, a notify handler in the button can disable the button.



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties**
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Figure 2 A message reaching a page also reaches the notify handler



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

The advantage of using a notify handler is that rather than simply responding to a message, the object can be completely self-contained—you do not have to add a handler elsewhere to send a message to the object. You can copy the object to another book with no other changes. In the example of the “Next” button, without notification you would have to change the book script to send an *enterPage* message to the button in addition to simply copying the button to the background.

Types of notify handlers

You can use two types of notify handlers: *notifyBefore* handlers and *notifyAfter* handlers. *NotifyBefore* handlers execute just before the message for which they are being notified is sent, and *notifyAfter* handlers execute just after the message is sent. You write a notify handler just as you write a normal handler, substituting the words *notifyBefore* or *notifyAfter* for the words *to handle* in the first line of the handler:

```
notifyBefore enterPage
...
end --Further statements here
```

As an example, you can create an on-screen clock by displaying the time of day in a field. Create a *notifyBefore* handler in the field that responds to the *idle* event by setting the field’s text to the current time:

```
notifyBefore idle
  system oldTime
  if oldTime is not sysTime then --Update if time has changed
    my text = sysTime
    oldTime = sysTime
  end if
end idle
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Another use for a notify handler is to create an object that is visible only at Author level. The following script illustrates how you can use notify handlers to ensure that the object is always hidden when the user switches to Reader level:

```
notifyBefore reader
  hide self
end
notifyAfter author
  show self
end
notifyBefore enterPage
  if sysLevel = "author"
    show self
  else
    hide self
  end if
end enterPage
```

Using notify handlers

When you use notify handlers, keep these points in mind:

- ◆ Multiple objects can be notified of the same message if each object contains a notify handler. However, you cannot specify the order in which the objects are notified of the message.
- ◆ Do not use the *forward* command in notify handlers. If you do, ToolBook II generates an error when you attempt to save the script.
- ◆ To stop a notify handler from processing a message in its script, use *break*.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

- ◆ You can put notify handlers into the script of any object on a page or background. However, you cannot put notify handlers into the scripts of pages, backgrounds, viewers, or books.
- ◆ Notify handlers respond only to messages that reach a page. If an object traps a message and does not forward it to the page, ToolBook II will not call the notify handler. Similarly, if the message goes first to an object on the background and the script in that object does not explicitly send it to the page, ToolBook II will not call the notify handler.

The table below shows the properties and OpenScript commands that you can use with notify handlers.

Properties and commands for notify handlers

Keyword	Description
<i>notifyObjects</i>	Page or background property that specifies a list of all objects that contain notify handlers
<i>notifyAfterMessages</i> , <i>notifyBeforeMessages</i>	Properties of an object that list the notify handlers written for that object
<i>sendNotifyAfter</i>	Command that activates a specific <i>notifyAfter</i> handler within a script
<i>sendNotifyBefore</i>	Command that activates a specific <i>notifyBefore</i> handler within a script



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement**
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Chapter 6

Creating an OpenScript statement

This chapter describes how to create a statement in OpenScript and includes detailed information about the components that make up a statement: commands, functions, expressions, and operators. You will also learn about control structures, such as *if/then/else* and *do/until*, which you use to control blocks of statements in a script and expressions.

CONTENTS

About OpenScript commands	90
Using OpenScript control structures	92
Using functions	100
Creating user-defined functions	102
Creating expressions with operators	104



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 **Creating an OpenScript statement**
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

About OpenScript commands

To create an OpenScript statement, use a command and the parameters the command requires. OpenScript includes commands to:

- ◆ declare variables and get and set the values of variables or properties (*get*, *increment*, *local*, *pop*, *push*, *put*, *set*, and *system*).
- ◆ manipulate objects (*align*, *clear*, *draw*, and *move*).
- ◆ interact with the user and navigate between pages (*ask*, *go*, *request*, *search*, *sort*, and *transition*).
- ◆ print, save, read, or write data in ToolBook II files (*export*, *openFile*, *print*, *readFile*, and *writeFile*).
- ◆ add and remove menu items (*addMenu* and *addMenuItem*).
- ◆ exchange data with other Windows applications (*executeRemote*).
- ◆ extend programming code to include routines and data within Windows DLLs.



For a complete list of commands and information on them, refer to the OpenScript reference in online Help.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 **Creating an OpenScript statement**
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Using commands versus sending messages

In some instances, you can either use a command or send a message to accomplish the same task. For example, commands such as *align*, *clear*, *export*, *import*, *run*, *save as*, *search*, and *sort* are also messages. Use these guidelines to determine when to use a command or message:

- ◆ Use commands such as *export* or *search* to execute the action without displaying the related dialog box. Use commands with keywords such as *author* or *reader* if you want to avoid running related message handlers. For example, the statement *set sysLevel to reader* does not execute actions in a *to handle reader* handler, but the working level is changed to Reader.
- ◆ Send messages such as *import* or *saveAs* if you want the user to perform the action by filling in a dialog box. Send a message such as *reader* or *clear* when you want any handlers for the message to run in addition to accomplishing the primary action. Sending a message results in the same behavior as choosing the related menu item.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 **Creating an OpenScript statement**
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Using OpenScript control structures

A control structure is a group of statements that runs under particular circumstances, such as when handling a message, looping (repeating) a block of statements, or testing a condition. The OpenScript control structures are shown below.

OpenScript control structures

Type	Commands	Description
Looping	<i>step</i>	Repeats a fixed number of times
	<i>do/until</i>	Repeats until a condition is met; always executed at least once
	<i>while</i>	Repeats while a condition is true
Establishing conditions	<i>if/then/else</i>	Executes statements if a condition is true; <i>else</i> branch optional
	<i>conditions</i>	Executes one from a set of mutually exclusive choices
Handling messages	<i>to handle</i>	Reacts to a system or user message
Creating properties and user-defined functions	<i>to set</i>	Defines a script to create or calculate a value for a property or user-defined function
	<i>to get</i>	Defines a script to return the value of a user-defined function or calculated user property

(continued)



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 **Creating an OpenScript statement**
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

OpenScript control structures, *continued*

Type	Commands	Description
Setting target window	<i>in <viewer></i>	Temporarily makes a viewer the target window for running scripts
Printing (spooling)	<i>start spooler</i>	Routes subsequent <i>print</i> commands to the current print device
Linking DLLs	<i>linkDLL</i>	Declares functions from a DLL to be called
Windows message	<i>translateWindowMessage</i>	Establishes handling of Windows-to-OpenScript message translation

You always begin a control structure with one of the control structures from the table, and close with the word *end*. The only exception is *do*, which ends with *until <expression>*. In between you can put any number of OpenScript statements. You can repeat the name of the control structure after *end* to make it easier to see where a specific control structure ends, as in this example:

```
step ctr from 1 to pageCount of this book
    flip
end step
```



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 **Creating an OpenScript statement**
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Examples of control structures

The most common control structures, such as *to handle* and *if/then/else*, are extensively illustrated throughout the documentation. This section shows you how to use some of the less common control structures, such as *conditions*, *do/until*, *in <viewer>*, *linkDLL*, and *while*.

For details about *to set* and *to get* handlers, see Chapter 5, “Handling messages, events, objects, and properties.” For details about user-defined functions, see “Creating user-defined functions” later in this chapter. For details about translating Windows messages, see Chapter 13, “Using dynamic-link libraries.”

Looping and testing conditions

Use a looping control structure to repeat a series of statements. The *while* control structure tests at the beginning of the loop. By contrast, the *do* control structure tests at the end, so the loop always executes at least one time. Use the *step* control structure when you know in advance how many times you want to repeat the statements.

```
--Uses the while control structure to get just the file name
--from a string that contains a path plus the file name
to get nameOnly fFile
    vOffset = offset("\", fFile)
    while vOffset > 0
        clear chars 1 to vOffset of fFile
        vOffset = offset("\", fFile)
    end while          --Repeats control structure name for clarity
    return fFile
end nameOnly
```

(continued)



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 **Creating an OpenScript statement**
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

```
--Uses the do control structure to display the cursor  
--position in the status bar at Reader level  
to handle buttonClick  
  if visible of statusBar is false then  
    show statusBar  
  end if  
do  
  caption of statusBar = mousePosition of this window  
  until keyState(keyLeftButton) is up  
end buttonClick
```

There are two ways to test conditions in OpenScript. For a simple test, use *if/then*, adding an *else* branch if you want to explicitly define the statements that should execute otherwise. To test for one of a series of mutually exclusive possibilities, use the *conditions/when/else* control structure. ToolBook II tests each condition in turn, executing the statements that follow the first condition that is true. As with an *if/then* control structure, you can add an *else* branch to the *conditions* to indicate the statements that should execute otherwise.

```
--Uses the conditions control structure to test for several  
--possible user responses  
to handle buttonClick  
  request "Do you want to proceed?" with "Continue" or "Quit" \  
  or "Restart"  
  conditions  
    when It is "Continue"  
      request "You chose Continue"  
    when It is "Quit"  
      request "You chose Quit"  
    when It is "Restart"  
      request "You chose Restart"  
    else  
      request "You closed without choosing anything"  
    end conditions  
end buttonClick
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 **Creating an OpenScript statement**
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Setting the target window

If you are running a script in one viewer but want to affect objects in another, you can use the *in* control structure to temporarily make the second viewer the target window. The statements inside the control structure then run as if they were running in the second viewer, and implicit object references and the generic references *this* and *my* refer to the second viewer. For example, you can use *in* if you want a user to click a button in the current viewer to navigate to a particular page in another viewer.

```
to handle buttonClick
  show viewer "Topics"
  in viewer "Topics"
    currentPage = "Contents"
  end in
end buttonClick
```

Spooling

Use the *start spooler* control structure to initialize the print spooler. The *print* and *print eject* commands work within this control structure only, and so you must use it to print from a script. For example:

```
to handle buttonClick
  start spooler      --Encloses print statements
  print all pages
  end spooler
end buttonClick
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 **Creating an OpenScript statement**
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Linking to DLLs

You use the *linkDLL* control structure to initialize and declare a link to a DLL.

```
--Link a DLL to call a function that returns a file size
to handle buttonClick
  linkDLL "tbdos.dll"
    LONG getFileSize(String)
  end linkDLL
  request getFileSize("c:\autoexec.bat")
end buttonClick
```

To determine if a DLL is already linked, you can check the value of the system property *sysLinkedDLLs*, which contains a list of the DLLs currently linked. For details about linking DLLs, see Chapter 13, "Using dynamic-link libraries."

Nesting control structures

You can nest all control structures within other control structures; however, you cannot put a *start spooler* control structure within another *start spooler* structure. You can nest control structures any number of levels deep.

Interrupting control structures

To interrupt or modify the flow of a control structure, use *break* and *continue* inside the structure. Use the *break* statement to quit a handler without finishing all of the statements in it, or in loops to quit the loop early, such as when an error is detected.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 **Creating an OpenScript statement**
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

If you use *break* to quit the handler, control returns to whatever process called the current handler. The *break to system* statement stops execution of all handlers currently running, including any that called the current handler. The following handler uses the *break* statement to prevent the handler from completing and exiting ToolBook II:

```
to handle buttonClick
  request "Do you want to quit?" with "Continue" or "Quit"
  if It is "Continue" then
    request "You have elected to continue."
    break          --Quits handler
  end if
  save changes to this book
  send exit       --Exits from ToolBook II
end buttonClick
```

To use *break* to quit a control structure within a handler (such as *step* or *do*), include the name of the control structure after the word *break*. For example:

```
step i from 1 to 1000
  --Checks room available on page
  if percentFreeSpace of this page < 10 then
    break step
  else
    ...          --Further statements here
  end if       --Otherwise continue processing
end step      --Handler resumes here after break statement
              --or if step has finished successfully
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement**
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

The *continue* statement is used to skip statements in a control structure (most often *step* or *do/until*) and continue directly to the next iteration. For example:

```
--This handler counts the objects on a page that are not buttons
to handle buttonClick
  vTotalItems = 0
  --Gets a list of all the objects on the page
  itemList = objects of this page
  step i from 1 to itemCount(itemList)
    --Gets the next object from the list of objects
    pop itemList
    --Tests to see if the current item is a button
    if object of It is "button" then
      --Skips to the "end step" statement
      continue
    end if
    --Otherwise increments the total object count
    increment vTotalItems
  end step
  request "total items = " & vTotalItems
end buttonClick
```



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 **Creating an OpenScript statement**
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Using functions

Functions are procedures that you use in your script to calculate a value or perform an operation. ToolBook II has over 200 built-in functions for arithmetic, statistics, string handling, financial calculations, and other purposes. For example:

```
--The function "textlineCount" returns the number of lines in a field  
headCount = textlineCount(text of field "Attendees")  
  
--The function "keyState" returns "up" or "down" for the  
--key value that you request  
if keyState(keyShift) is down then  
...    --Further statements here  
end
```



For more information on functions, refer to the OpenScript reference in online Help.

You can also write your own user-defined functions. For details, see “Creating user-defined functions” later in this chapter.

Using parameters

When you call a function, you include parameters for the values used to calculate the return value. For example, the variable *vName* is the parameter for the *uppercase* function in this script fragment:

```
vName = "Mary Jones"  
get uppercase(vName)    --Puts MARY JONES into It
```

Note that the variable *vName* still contains the mixed-case “Mary Jones” text.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 **Creating an OpenScript statement**
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Except for array parameters that use the *by reference* special term, OpenScript passes parameters by value, copying the value when you pass it in a function. The values of the parameters in the calling handler do not change.

Passing parameters by value can use up a lot of memory, especially when the variable is a large array. This is because ToolBook II makes a copy of the variable to use within the function. If you want to pass variables by reference that are not arrays, you must allocate memory and pass a pointer to the memory as a parameter to your function. To do this, use the *linkDll* command to link to the Windows KERNEL library. Link the *GlobalAlloc*, *GlobalLock*, *GlobalUnlock*, and *GlobalFree* functions. Use the *GlobalAlloc* function to allocate memory and the *GlobalLock* function to get a pointer to the memory. Use the ToolBook II internal pointer functions (*pointerInt*, *pointerChar*, etc.) to set and get variables within the memory block at specific offsets. Then, pass the memory pointer to any function accepting a POINTER type parameter.

For details about using DLL functions, see Chapter 13, “Using dynamic-link libraries,” and the Microsoft Windows software development kit (SDK) documentation.



For information on passing arrays by reference, see the entry for *by reference* in the OpenScript reference in online Help.



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 **Creating an OpenScript statement**
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Creating user-defined functions

A user-defined function is a handler you write that calculates a value or runs a procedure and then returns a result. For example, to calculate the same value often, such as cost of goods, you can write the calculation as a user-defined function, and then call the function as needed. User-defined functions are particularly useful when the calculation is complex or when you need to perform the same calculation in different applications.

You can use user-defined functions anywhere that you can use built-in functions. To create a user-defined function, write a *to get* handler. The handler returns the calculated value using the *return* statement. For example:

```
--Defines function name and parameters  
to get grossMargin Sales, COGS  
    --Calculates and returns the results to the calling handler  
    return Sales - COGS  
end
```

You can use several statements of varying kind in a *to get* handler—commands, messages sent to run other handlers, or other function calls. The *to get* handler must include a *return* statement that specifies the value to be returned. The *return* statement acts as a break for the function—no statements following a *return* statement are executed.



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 **Creating an OpenScript statement**
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Naming user-defined functions

You can assign a name to a user-defined function that is similar to (but not exactly like) the name of a built-in function. Names you assign to a user-defined function:

- ◆ must begin with a letter, an underscore character (`_`), or a prefix symbol, such as the at symbol (`@`).
- ◆ cannot contain spaces or any characters other than letters, numbers, the underscore character, or the prefix symbol.

The following example uses the at symbol (`@`) to distinguish a user-defined function from a built-in function.

```
--Calls the built-in function named average  
request average(1,2,3,4,5)  
  
--Calls the user-defined function named average  
request @average(1,2,3,4,5)
```

Placing user-defined functions in the object hierarchy

You can place a *to get* handler anywhere in the object hierarchy. However, it's good practice to define user-defined functions in the book script because it makes the scripts easier to maintain. If you define functions in the book script of a system book, you can use them with any application and avoid repeating the function definitions in other books. However, the slight overhead of searching up the hierarchy for the function might affect the performance of an application.



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 **Creating an OpenScript statement**
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Calling user-defined functions

After you define a function, you can call it from any handler the same way you call a built-in function. ToolBook II searches the hierarchy for *to get* handlers as for any message handlers. The following example calls the user-defined function *grossMargin()* created earlier:

```
to handle buttonClick
    local vSales, vCOGS
    vSales = text of field "Total Sales"
    vCOGS = text of field "Unit Cost" + text of field "Overhead"
    text of field "Gross Margin" of page 3 = grossMargin(vSales, vCOGS)
end
```

Creating expressions with operators

An expression is any combination of a formula, calculation, comparison, or set of values that results in a single value. You create an expression by linking its elements with an operator, which is a word or symbol that indicates the operation to be performed. For example:

```
--Multiplies two values
subTotal = vItems * vQty

--Joins three strings into one
greeting = "Dear" && title && lastName

--Divides the values of two functions
average = sum(allItems) / itemCount(allItems)

--Compares two values
if text of field "Amount" > vLimit then ...
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 **Creating an OpenScript statement**
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

You can compose expressions from any combination of object references, properties, references to strings or to variables, numbers, literal values, and other expressions, including:

- ◆ **containers** Anything that holds a value, such as variables, properties, the text of a field, or the Command window.
- ◆ **functions** References that call a predefined operation and return a value. For details, see “Using functions” earlier in this chapter.

- ◆ **literal values** Values entered character by character in a statement. For example:

```
text of field "Zip" = "Please enter a 9-digit zip code."
```

- ◆ **constants** Specific values referred to by name, such as *pi*, which refers to the value 3.141592653589793.

- ◆ **expressions with operators** A combination of operators and operands that results in a value. For example:

```
--Results in 250 if the current value of svCost is 100  
100 + svCost + 50
```

- ◆ **expressions with strings** References to parts of strings. For example, if the text of field *ABC* contains *primary colors*, then the following statement refers to the value *primary*:

```
word 1 of text of field "ABC"
```

The value returned by an expression can be a number, character string, or logical value (*true* or *false*).



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 **Creating an OpenScript statement**
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Operators in ToolBook II

You can create expressions using four types of operators: arithmetic, logical, string, or bitwise, as shown in the following tables.

Arithmetic operators

Use arithmetic operators to create mathematical expressions.

Operators for arithmetic expressions

Operator	Meaning
+	Addition. For example: It = 25.3 + vFirst --Puts result into It
-	Subtraction. For example: It = 25 - 4 --Puts 21 into It
*	Multiplication. For example: It = 25 * 4 --Puts 100 into It
/	Division. For example: It = 25 / 4 --Puts 6.25 into It
<i>div</i>	Division (returns only the integer). For example: It = 25 div 4.5 --Puts 5 into It
-	Negation. For example: It = -4 + 25 --Puts 21 into It
^	Exponentiation. For example: It = 2 ^ 4 --Puts 16 into It
<i>mod</i>	Modulo (returns the remainder). For example: It = 16 mod 5.0 --Puts 1 into It



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 **Creating an OpenScript statement**
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Logical operators

Use a logical operator to evaluate two expressions and return *true* or *false*. Expressions containing dates, currency, or other formatted data can be compared as *<type>* to force ToolBook II to convert the data in the comparison to the appropriate format first.

Operators for logical comparisons

Operator	Meaning
= or is	Equal. <i>True</i> if both expressions have same value. For example: It = (25=4) --Puts false into It if color is blue then color = red end
<	Less than. <i>True</i> if expression on left is less than expression on right. For example: if balance < lowLimit then text of field "odWarning" = "Overdraft" end
<=	Less than or equal to. <i>True</i> if expression on left is less than or equal to expression on right. For example: if text of field "Date" <= sysDate as date request "Too early to process" end
>	Greater than. <i>True</i> if expression on left is greater than expression on right. For example: get 25 > 4 --Puts true into It

(continued)



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 **Creating an OpenScript statement**
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Operators for logical comparisons, *continued*

Operator	Meaning
<code>>=</code>	Greater than or equal to. <i>True</i> if expression on left is greater than or equal to expression on right. For example: <pre>if vFirst >= vLast request "Last item is less than first." end</pre>
<code><></code> or <i>is not</i>	Not equal to. <i>True</i> if two expressions are different. For example: <pre>if svText <> text of field "Text" text of field "Text" = svText end</pre>
<i>and</i>	Boolean And. <i>True</i> if both expressions are true. For example: <pre>if text of field "input" > "01/01/94" as date and \ text of field "refAmount" < 25 request "The expression is true." end</pre>
<i>or</i>	Boolean Or. <i>True</i> if either expression is true. For example: <pre>if interest > 10.5 or interest < 9.5 then request "Interest rate is out of range." end</pre>
<i>contains</i>	String search. <i>True</i> if expression on right is found in expression on left. For example: <pre>if text of field "emperors" contains "Roman" go to book "c:\lessons\roman.tbk" end</pre>

(continued)



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 **Creating an OpenScript statement**
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Operators for logical comparisons, *continued*

Operator	Meaning
<i>is in</i>	String search. <i>True</i> if expression on left is found in expression on right. For example: <pre>if nameOnly(vName) is in text of field \ "chooseIndex" request "That index is already open." end</pre>
<i>is not in</i>	String search. <i>True</i> if expression on left is not found in expression on right. For example: <pre>if "Nero" is not in vAnswerText request "Please try again." end</pre>
<i>not</i>	Not. <i>True</i> if expression on right is false; <i>false</i> if expression on right is true. For example: <pre>--File name is in title bar if not captionShown of mainWindow request "Do you want to personalize this book?" \ with "Yes" and "No" if It is "Yes" send personalize -User-defined message end if end if</pre>



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 **Creating an OpenScript statement**
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

String comparisons are not case-sensitive, so *b* is equal to *B*. If you want to compare values by case, first use the `charToAnsi()` function to convert the values to their ANSI equivalents. The following is a user-defined function that performs a case-sensitive comparison on two strings:

```
to get compareCase string1, string2
  --If a case-insensitive check fails, the strings are different
  if string1 <> string2 then
    return false      --Return to calling handler
  end if
  --Compare character-by-character
  step ctr from 1 to charCount(string1)
    if charToAnsi(char ctr of string1) <> \
      charToAnsi(char ctr of string2) then
      return false    --Return to calling handler
    end if
  end step
  return true
end compareCase
```

Text values in comparison expressions are converted to uppercase and then compared according to their position in the ANSI character set. For *as <type>* comparisons, expressions are compared according to the current values of *sysNumberFormat*, *sysDateFormat*, and *sysTimeFormat* as required by the context.



For details about formats for *<type>* comparisons, refer to the “format” topic in online Help.

String operators

Use string operators to concatenate (join) two strings into a longer one, or to extract substrings from larger strings. For details about handling text strings, see Chapter 7, “Using variables, values, and text.”



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 **Creating an OpenScript statement**
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Operators for string evaluations

Operator	Meaning
&	Concatenates two expressions with no space between them. For example: push (bookPath & "mysysbook.sbk") onto sysBooks
&&	Concatenates two expressions with a space between them. For example: request "The" && "End" --Displays "The End"
<i>char(s) or character(s)</i>	One or more contiguous characters of text, including letters, numbers, spaces, punctuation marks, tabs, and other special characters. For example: if char 1 of userResponse is "Y" then send exit end vFirst3 = chars 1 to 3 of fullName
<i>word or words</i>	One or more contiguous sequences of printable characters separated on each side by a space or a nonprintable character, or by the beginning or end of the expression that includes the word. For example: surname = last word of text of field "FullName"
<i>item or items</i>	One or more contiguous values in a comma-separated list. For example: xPosition = item 1 of mousePosition of this window yPosition = item 2 of mousePosition of this window

(continued)



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 **Creating an OpenScript statement**
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Operators for string evaluations, *continued*

Operator	Meaning
<i>textline</i> or <i>textlines</i>	A string of zero or more characters that ends with a carriage return/linefeed (CRLF), or two or more strings separated with CRLFs. For example: <pre>allLines = text of field "listbox1" step ctr from 1 to textLineCount(allLines) request textline ctr of allLines end</pre>

Bitwise operators

Bitwise operators allow you to perform binary operations on individual bits. They are most useful when you need to read or set the value of a specific bit in a parameter for a DLL function. For example, many Windows DLLs return a single value in which each individual bit designates a particular value. Use numbers to represent the values for bitwise operators.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 **Creating an OpenScript statement**
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Operators for bitwise operations

Operator	Meaning
<i>bitAnd</i>	Performs a bitwise AND operation on two numbers. For example: --Displays 2 (binary 11 AND 10 yields 10) request 3 bitAnd 2
<i>bitOr</i>	Performs a bitwise inclusive OR operation on two numbers. For example: --Displays 3 (binary 01 OR 10 yields 11) request 1 bitOr 2
<i>bitXOr</i>	Performs a bitwise exclusive OR operation on two numbers. For example: --Displays 1 (binary 11 XOR 10 yields 01) request 3 bitXOr 2
<i>bitNot</i>	Converts all bits to their opposite values (performs a one's complement of a number). For example: --Displays -2 (binary 0000001 NOT yields 11111110) request bitNot 1
<i>bitShiftRight</i>	Performs a bitwise shift to the right on a number by a specified number of bits. For example: --Puts 18 into It (144/(2^3)) get 144 bitShiftRight 3 --(10010000 yields 10010)
<i>bitShiftLeft</i>	Performs a bitwise shift to the left on a number by a specified number of bits. For example: --Puts 32 into It (4*(2^3)) get 4 bitShiftLeft 3 --(100 becomes 100000)



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 **Creating an OpenScript statement**
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

As an example, the following program calls the DLL function *GetWinFlags()* and uses bitwise operators to determine the type of processor in the computer:

```
to get processorType
  local dword winFlags
  local retValue
  linkDLL "kernel"
    DWORD  GetWinFlags( )
  end linkDLL
  winFlags = GetWinFlags( )
  conditions
    when winFlags bitAnd 2 <> 0
      retValue = "286"
    when winFlags bitAnd 4 <> 0
      retValue = "386"
    when winFlags bitAnd 8 <> 0
      retValue = "486"
  end conditions
  return retValue
end processorType
```

The *argument* operator

You can use a special OpenScript operator called *argument* to get the value of a parameter passed to the currently executing handler. A parameter can consist of more than one item, so *argument n* and *item n* of *argList* (where *n* ≤ the value of *argCount*) can return different values.

```
--Puts value of the 3rd parameter passed into a field
text of field "Name" = argument 3
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 **Creating an OpenScript statement**
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Precedence of operators

ToolBook II evaluates expressions according to the predefined operator precedence shown in the following table. Operators of equivalent precedence are evaluated left-to-right except for the exponentiation expressions, which are evaluated right-to-left. You can force ToolBook II to change operator precedence by using parentheses around any expression.

Operator precedence in OpenScript

Order	Operators
0	()
1	<i>not</i> , <i>argument</i> , - (negation)
2	<i>character</i> , <i>word</i> , <i>item</i> , <i>textline</i> (string operators)
3	<i>bitNot</i>
4	<i>bitShiftLeft</i> , <i>bitShiftRight</i>
5	<i>bitAnd</i>
6	<i>bitOr</i> , <i>bitXOr</i>
7	^ (exponentiation)
8	*, /, <i>div</i> , <i>mod</i>
9	+, - (subtraction)
10	&, &&
11	<i>is in</i> , <i>is not in</i> , <i>contains</i>
12	<, >, <=, >=, =, <>, <i>is</i> , <i>is not</i>
13	<i>and</i>
14	<i>or</i>



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 **Creating an OpenScript statement**
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Examples of operator precedence

Example	Result
request $1 + 20 / 5 * 40 + 2$	163
request $(1+20) / (5 * (40 + 2))$.1
request 3^{2*2}	18
request $3^{(2*2)+10}$	91
request characters 1 to 2 of $1000 + 2000$	2010
request characters 1 to 2 of $(1000 + 2000)$	30
request $1 + 1 \& 2$	22
request $1 + (1 \& 2)$	13



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text**
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Chapter 7

Using variables, values, and text

This chapter describes how you can use individual components of OpenScript programs, including variables, constants, and literal values, and discusses how to use text in your scripts.

CONTENTS

Using variables	118
Assigning data types to variables	123
Using arrays	127
Working with literal values	134
Working with constants	140
Working with character strings	142
Working with lists	147
Referring to colors	149



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 **Using variables, values, and text**
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Using variables

A variable is a container you define in a script to store data temporarily. Use a variable any time you are working with data that might change each time you run the script, as this example shows:

```
to handle buttonClick
    vYearOfBirth = text of field "birthYear"    --Stores a property value
    vThisYear = "94"                          --Stores a literal value
    vAge = vThisYear - vYearOfBirth           --Stores a result
    request "This year you will be" && vAge && "years old."
end
```

Variables in OpenScript are like variables in other programming languages, with these exceptions:

- ◆ You can declare variables with or without a data type.
- ◆ A variable declared without a data type can contain numeric, logical, string, and other values such as dates and times.
- ◆ With the exception of arrays, you do not need to declare local variables before you assign a value to them.
- ◆ You can define variables as global or local using the local or system lay words.

You can declare variables to be arrays in OpenScript. For details, see “Using arrays” later in this chapter.



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Rules for naming variables

When you name a variable, the name:

- ◆ must begin with a letter or underscore.
- ◆ can contain only alphanumeric characters, underscores, and the at symbol (@) as a prefix.
- ◆ cannot be an OpenScript keyword (for example, *request* or *fillColor*).

To distinguish names of variables from names of properties and OpenScript keywords, use a prefix such as *v* (variable) or *sv* (system variable), or use the reserved prefix @:

<code>local vTotal</code>	--Uses "v" to mark local variable
<code>system svVarName</code>	--Uses "sv" to mark system variable
<code>@text = text of field "name"</code>	--Uses "@" as prefix to distinguish --variable from a property

Note OpenScript is not case-sensitive, and so the variable names *vStudentName* and *vstudentname* are the same.

Assigning values to variables

Use *set*, *put*, or an equal sign (=) to assign a value to a variable; these commands have the same effect. You can put up to 64 KB of data into a single variable.

Note To get around this 64 KB limit, you can use the *linkDLL* command to link to the Windows KERNEL DLL. Use the *GlobalAlloc* Windows API function to allocate the desired amount of memory. For more information on the Windows *GlobalAlloc* API function, refer to the Microsoft Windows software development kit (SDK) documentation.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

If the value is text and has a space in it, you must enclose the text in quotation marks. Your script will work more efficiently if you enclose all text values in quotation marks.

```
firstPlayer = text of field "Player1"  
set PrimaryColors to "red,blue,green"  
put "$1,000 cash" into prizeMoney    --Contains space,  
                                     --requires quotation marks
```

Using local and system variables

You can use both local and system variables in handlers. A local variable keeps the value you put into it for the duration of the handler. A system variable retains its value for the duration of the ToolBook II instance. Use a system variable for information that needs to be available to more than one handler—for example, a user's name, the last keystroke entered, or the value of a property before you change it.

Note You must declare system variables in every handler or function that uses them.

Creating local variables

You do not have to declare local variables; when you assign the first value to a variable, ToolBook II both declares and initializes the variable. There is a *local* statement to declare local variables, but it is used primarily to assign data types to variables.

Tip To share local variable values between handlers, pass them as parameters.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Creating system variables

To create a system variable, declare it with the *system* statement. For example, the following *enterPage* handler uses a system variable to determine if the user is navigating to a page for the first time:

```
to handle enterPage
  system svFirstTime          --Declares a system variable
  if svFirstTime = null then  --Null if this is the first time
    vWelcomeText = "Welcome to the Tutorial."
    text of field "Banner" = vWelcomeText
    svFirstTime = false
  end if
  forward
end enterPage
```

To determine what system variables you have already declared, get the value of the system property *sysSystemVariables*, as in this example:

```
--Displays all current system variables
to handle buttonClick
  if itemCount(sysSystemVariables) > 0
    step ctr from 1 to itemCount(sysSystemVariables)
      request item ctr of sysSystemVariables
    end step
  end if
end buttonClick
```

Tip You can also maintain global values in user properties. User properties take less memory and are saved with the book, but can be slower to access.



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Using reserved variables

ToolBook II maintains certain variables, such as *It*, *argList*, and *argCount*, into which it puts the results of a statement or information about the current handler. You can use the following variables in your scripts. All are treated as local variables within a handler:

- ◆ A number of statements, including *ask*, *get*, *pop*, *request*, and *set*, put the results of their actions into the variable *It*. Be sure to copy the value of *It* to another container if you need the value for more than one statement.
- ◆ The variable *argList* contains the parameters passed to the current handler. You cannot change the value of this variable.
- ◆ The variable *argCount* contains a count of the number of values passed as parameters to the current handler. You cannot change the value of this variable.

For example, in the following handler, the *request* statement returns a user response in the variable *It*. The variable *argList* returns a list of four items (two for the location, and one each for the logical values in *isShift* and *isCtrl*), and *argCount* returns *three*, because a *buttonClick* handler takes three arguments.

```
to handle buttonClick loc, isShift, isCtrl
    request "Do you wish to see arguments?" with "Yes" or "No"
    if It is "Yes" then
        request "argList =" && argList
        request "argCount =" && argCount
    end if
end buttonClick
```



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Assigning data types to variables

A data type is a format you assign to a variable when you first declare it, such as integer, text, real number, and so on. Assigning data types to variables often makes handlers run faster and use less memory because data is already in the internal (binary) form that ToolBook II requires, and in its most efficient format. Assigning a data type to a variable also helps prevent errors because you cannot assign inappropriate data to the variable.

To declare a data type, use the statements *local* or *system* with a data type before referencing the variable:

```
local LONG i  
system INT pagesDisplayed, pagesLeft  
local COLOR oldFillColor
```

If you do not declare a data type for a variable, ToolBook II selects a data type appropriate to the statement in which the variable first appears. ToolBook II can change the data type in subsequent statements as necessary, but this can slow your scripts down. If you do declare a data type for a variable, ToolBook II assigns the appropriate data type to the variable when you compile the script. When the handler is running, ToolBook II checks that only the correct type of data is assigned to the variable; if not, ToolBook II displays an error message.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

The following table shows data types you can use in OpenScript.

OpenScript data types

Type	Description
<i>int</i>	16-bit signed integer, range -32768 to 32767
<i>long</i>	32-bit signed integer, range -2147483648 to 2147483647
<i>real</i>	64-bit floating point number; ToolBook II will accept a <i>long</i> value; all numeric functions and operators that accept real numbers use this type internally
<i>word</i>	16-bit unsigned integer, range 0 to 65535
<i>dword</i>	32-bit unsigned integer, range 0 to 4294967295
<i>string</i>	Character string (Avoid assigning <i>string</i> to data that is more efficiently represented by a different type.)
<i>logical</i>	<i>True</i> or <i>false</i>
<i>point</i>	<i>x, y</i> ; where <i>x</i> and <i>y</i> are integers (a list of two location values)
<i>color</i>	List of three non-negative numbers in HLS units
<i>stack</i>	List of comma-separated items
<i>date</i>	Date in system format
<i>time</i>	Time in system format
<i>page</i>	Explicit reference to a page
<i>background</i>	Explicit reference to a background
<i>layer</i>	Explicit reference to a page or background
<i>graphic</i>	Explicit reference to an object on a page or background
<i>field</i>	Explicit reference to a field, record field, or combo box
<i>object</i>	Explicit reference to a book, page, background, object, or graphic
<i>book</i>	Explicit reference to a book path



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Tip Use the function *isType()* to determine whether data matches a particular data type. For example:

```
ask "Enter starting date for report:"
if isType(date, It) then
    vStartDate = It
else
    request "Please enter a date."
end
```

ToolBook II initializes the value of an untyped variable or array to *null* when it is declared. ToolBook II initializes the value of a typed variable differently depending on the data type, as the following table shows.

Default values for variables declared with data types

Data type	Initial value
<i>dword, int, long, real, word</i>	<i>0</i>
<i>logical</i>	<i>false</i>
<i>point</i>	<i>0,0</i>
<i>color</i>	<i>0,0,0</i>
<i>stack, string</i>	<i>null</i>
All other types	Consider uninitialized



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 **Using variables, values, and text**
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Choosing the appropriate data type

Your application will run more efficiently if you assign a data type appropriate to how the variable will be used. For example, using the data type *string* for a logical value uses more memory than necessary. Similarly, using the data type *string* for a counter in a *step* statement forces ToolBook II to convert the data from a string to a numeric value with each iteration of the loop. The following table shows operations that function most efficiently when you use the suggested data type.

Recommended data types for operations

Operation	Data type
Numeric operations	<i>real</i>
Text operations	<i>word</i>
Logical operations	<i>logical</i>
Step loops	<i>long</i>

Declaring data types for system variables

You can declare a different data type for system variables in different handlers. When you assign a value to the variable, ToolBook II converts the data to the appropriate type if possible. Because of this conversion, system variables are slightly less efficient than local variables. Any reference to the variable assumes that the value is of the declared type.



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 **Using variables, values, and text**
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Declaring data types for parameters

If you pass a typed variable as a parameter to another handler, you can declare the data type of parameters in a *to handle* statement. This results in greater efficiency and type-checking of data passed into a handler. For example:

```
to handle buttonClick point loc, logical isShift, logical isCtrl
    ...          --Further statements here
end
```

Using arrays

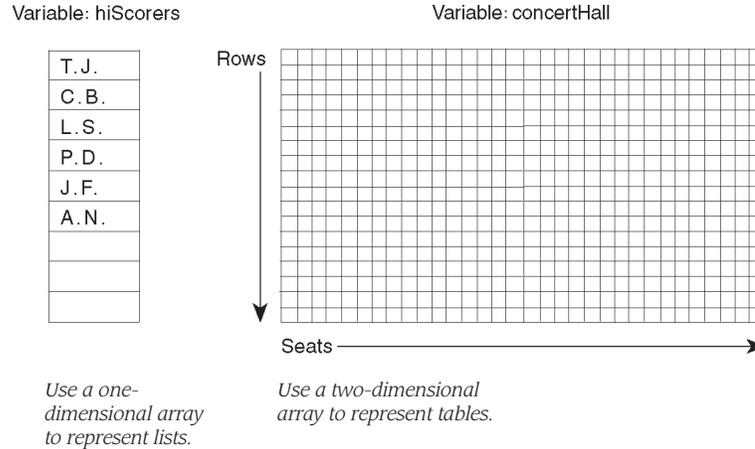
An array is a variable configured to work like a list or table of individual elements. For example, if you create a game and want to store the names of the 10 highest-scoring players, it is easier to have a single variable called *hiScorers* with 10 elements in it than it is to have 10 individual variables called *hiScorer1*, *hiScorer2*, *hiScorer3*, and so on. Or, if you are writing an application to take concert bookings for an auditorium, you can use an array to treat the concert hall as a matrix of rows and seats. Using arrays can help you create smaller and simpler code because arrays organize information accurately and systematically.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Figure 1 Using arrays to organize information



You can create one-dimensional arrays and multidimensional arrays of up to 16 dimensions. You can create arrays with a fixed size or size them dynamically as you add and remove elements. In general, use a dynamic array if you do not know in advance how many elements you require, or if the size of the array will change over time. You can have both fixed and dynamic arrays in a single handler, but you cannot mix fixed and dynamic dimensions in the same array. For example, you cannot use the following line of code:

```
local logical seat[32][ ]
```

Handlers often run faster if you use fixed rather than dynamic arrays. However, because dynamic arrays do not claim a predefined amount of memory, they sometimes work better under low memory conditions.



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Declaring an array

To declare an array, indicate the dimensions of the array in the *local* or *system* statement. To declare an array that is dynamically sized, use null values for the number of elements. ToolBook II generally runs faster if you assign a data type to the array.

```
local hiScorer[10]           --Declares array of 10 elements only
local logical seats[32][80] --Declares fixed-size array, 32 rows by 80
                             --columns; each element is true or false
system users[]              --Declares 1-dimensional dynamic array
local playerScores[][]      --Declares 2-dimensional dynamic array
```

The index number of an array begins at one (not zero, as in some systems).

Setting and getting values in arrays

You can assign values to elements of an array and read their values as you would any other variable:

```
auditorium[3][16] = true    --Fills one element of the array
if auditorium[5][10] is true
    request "That seat is booked."
end
```

To quickly assign values to all elements at once, use the *fill* command, which assigns a single value to the entire array, or assigns elements one-by-one. If you assign a data type to the array, be sure that the data type of the *fill* expression matches the array's. For example:

```
local int barrel[5]
fill barrel with 4          --Assigns the value of 4 to all 5 elements
                             --of array "barrel"
```



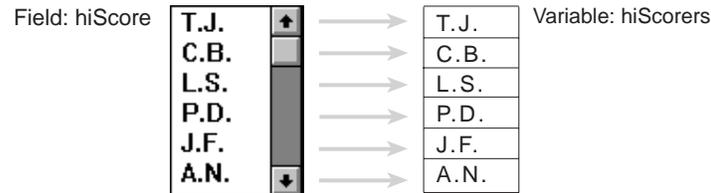
Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

You can also use *in ... order* with the *fill* command to assign individual units of text—lines, items, words, or characters—to elements in an array. For example, the *fill* statement in the following example assigns each line of a field to an element in an array:

```
local vHiScorers[]  
fill vHiScorers with text of field "hiScore" in [textline] order
```

Figure 2 Assigning text to an array



You can use up to four text operators (*char*, *item*, *textline*, and *word*) to subdivide text. Each operator you specify breaks the text into smaller components and assigns them to another dimension of the array. The number of text operators you use must match the array's dimensions, and the array must be large enough to accept all text components, or ToolBook II displays an error message. If the array is larger than the number of text components, the *fill* command leaves the existing contents of the unused elements as they were.

For example, if you are filling a two-dimensional array, you can assign textlines to one dimension and words to another. The following example script results in the array illustrated in Figure 3:

```
sampleText = "To be or not to be," & CRLF & "that is the question"  
local myArray[2][6]  
fill myArray with sampleText in [textline][word] order
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Figure 3 Assigning text units to an array
in *textline* and *word* order

```
myArray [1]      [2]      [3]      [4]      [5]      [6]
        [1] To      be      or      not      to      be,
        [2] that    is      the     question null    null
```

You can assign the elements of one array to another using the equal sign or the *set* or *put* command. The arrays must match in dimension and data type.

```
local array1[100]
local array2[100]
fill array1 with "a"    --Puts "a" into all elements of array1
array2 = array1         --Puts contents of array1 into array2
```

You cannot assign an array to a container that is not an array. For example, you cannot assign an array to the Command window.

Sizing arrays

When you declare an array, keep the following in mind:

- ◆ You can declare up to 65,536 elements in each dimension.
- ◆ An array can have up to 16 dimensions.
- ◆ Array size is limited by only the amount of available memory.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Getting an array's size

To determine an array's dimensions, use the *dimensions* function. Knowing an array's dimensions is useful if you need to step through the array element by element. For example:

```
--Use the value of "dimensions" to set a multiple dimension array
local myArray [5][4][3][2]
vMyDimensions = dimensions(myArray) --vMyDimensions contains
"5,4,3,2"
--Copies all the elements of an array into a text field with
--CRLF between elements
to get arrayToTextlines pArray[]
    local vTextlines
    get dimensions(pArray)          --Puts the size of the array into It
    step i from 1 to It
        textline i of vTextlines = pArray[i]
    end step
    return vTextlines
end arrayToTextlines
```

Passing arrays as parameters

You can pass an array variable as a parameter, as long as the called handler is written to accept an array for that parameter. To pass an array as a parameter, specify it the same way you would any variable. For example:

```
local vAlphabet[26]
get sortArray(vAlphabet)
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

In the called handler, you indicate an array by including a set of brackets for each dimension in the handler's opening statement. However, when working with dynamic arrays, you do not need to indicate the size of each dimension, only the number of dimensions, as in the following example:

```
--This handler accepts two arrays  
to handle modifyArrays array1[], array2[][]
```

In the following example, a dynamic array is passed as a parameter, modified, and passed back to the original routine.

```
--This is the calling handler  
to handle buttonClick  
    local vAlphabet[]  
    local vInvertedAlphabet[]  
    --(fill vAlphabet [] here)  
    vInvertedAlphabet = invertArray ( vAlphabet )  
    --(further processing here)  
end  
  
--The called handler flips the array upside down;  
--the array must be declared in the parameter list  
to get invertArray fixed vPassedArray[]  
    --A second array used temporarily in this handler  
    local vTempArray[]  
    myDim = dimensions(vPassedArray)  
    step i from 1 to myDim  
        --Inverts highest and lowest index numbers  
        index = abs((i-myDim)-1)  
        vTempArray[index] = vPassedArray[i]  
    end step  
    return vTempArray  
end invertArray
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Note By default, arrays are passed by value, which means that ToolBook II makes a copy of the array and any changes you make are not reflected in the calling handler. However, with arrays you can use the *by reference* special term to indicate that ToolBook II should pass only a pointer to the original array. This is faster and uses less memory, but if you change an array in a called handler, it is changed in the calling handler.

Assigning arrays as user properties

To preserve the values in an array between ToolBook II sessions, you can assign them to user properties just as you can ordinary variables. You can retrieve an array stored in a user property by assigning it to an array of the same dimensions. For example:

```
myUserProp of this page = vMyArray  
vMyArray = myUserProp of this page
```

Working with literal values

A literal value is a value you assign to a variable or use in an expression. For example:

```
vStartingValue = 0           --A value assigned to a variable  
vGreeting = "Dear" && vName  --A literal value in an expression
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Assigning literal values to variables with data types

If you assign a literal value to a variable that has been declared with a data type, the literal value must match the data type for that variable. If you try to assign a value to a variable that does not match the data type of that variable, ToolBook II will not compile the script. For example:

```
local int vHighScore --Declares an integer type
vHighScore = "1000" --Succeeds because "1000" is an integer
vHighScore = "abc" --Will not compile because "abc" is not an integer
```

Assigning literal values to untyped variables

When you assign a value to a variable that was not declared with a data type, ToolBook II determines the data type for the value based on how you are using the value. Depending on the context, ToolBook II interprets a value as a string, numeric value, logical (Boolean) value, time, date, list, color, or point in page units or pixels. For example:

```
show group "palmTree" at "100,100" --100,100 is a point in page units
text of field "Total" = sum(100,100) --100,100 is a list
```

You can take advantage of the ToolBook II ability to change the data type of a value as needed to use the same data in different ways. For example, you can use numbers as strings, as the following example shows:

```
vYear = 1957
request chars 3 to 4 of vYear --Displays "57"
request vYear + 30 --Displays "1987"
```



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Specifying numeric values

When you specify numeric values, use only the following characters:

- ◆ Numeric characters 1, 2, 3, 4, 5, 6, 7, 8, 9, and 0
- ◆ Decimal sign defined by *sysDecimal*, such as a period or a comma
- ◆ Negative and positive operators (- and +)
- ◆ Scientific notation exponentiation symbol (*E* or *e*)
- ◆ Currency sign defined by *sysCurrency*, such as \$
- ◆ Percent sign (%)

If you use any other characters, ToolBook II treats the value as a string. For example, ToolBook II considers dates in the format *mm/dd/yy* strings. If you include decimals when setting the value of a property that requires integers, ToolBook II rounds to the nearest integer. For example, if you set an object's *layer* property to 3.75, ToolBook II rounds the number to 4.

Use the *round()* function to round a numeric value to a whole number or to a number with a specific number of decimal places. For example:

```
x = 1.2345 / .333333  
request round(x, -4)           --Rounds result to 4 decimal places
```

You can format numbers, dates, and times to display correct characters and to ensure that ToolBook II recognizes the value as a number, date, or time. For details about formatting numbers, dates, and times, see Chapter 8, "Writing common scripts."



Additional information is located in the online Help topics *sysDateformat*, *sysNumberformat*, and *sysTimeformat*.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Specifying string values

Use string values for text when (for example) you want to assign captions or text to objects. You can use any combination of characters for a string value, including alphabetic characters, numbers, punctuation, and spaces. If you use a string value that contains a space, you must enclose the value in quotation marks. However, even if the value does not contain a space, it's wise to enclose it in quotation marks. ToolBook II will run faster if it does not have to determine whether the literal value is actually a variable. In addition, you are less likely to cause bugs by accidentally using the name of an existing variable. For example:

```
--No space so quotation marks not essential  
vMessage = Welcome  
  
--Runs faster with quotation marks  
vMessage = "Welcome"  
  
--Requires quotation marks because of spaces  
vMessage = "Welcome to the tutorial."
```

Logical (Boolean) values

Use *true* and *false* for logical (Boolean) values. Setting the value of an object property to *true* or *false* is often equivalent to turning an attribute on or off. For example:

```
sysUseWindowsColors = true
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 **Using variables, values, and text**
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Specifying page units and pixels

You use page units and pixels to specify values for screen coordinates for different types of objects. Use page units with graphic objects to specify an object's bounds and vertices, the location of an object relative to the upper-left corner of the page, or the destination or distance to move an object with the *move* command:

```
--Moves button down and right by 1 inch  
move button ID 0 by 1440,1440
```

There are 1440 page units per inch, or about 57 per millimeter. However, the number of pixels per page unit depends on the display device. Use the system property *sysPageUnitsPerPixel* to determine how many page units there are per pixel for the current display device. To convert page units for different display devices, use the *displayAspect()* function in *Tbwin.dll*.



For details, refer to "DLL reference" in the OpenScript reference in online Help.

Use pixels to specify the location or movement of viewers or palettes. For example:

```
--Sets maximum size of a viewer to 1/2 size of a VGA monitor  
maximumSize of viewer "Index" = 320,240
```

If you include decimals when specifying page units or pixels, ToolBook II rounds to the nearest integer.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Specifying hexadecimal, octal, and binary values

You can enter hex values using the prefix *0X* (zero plus x, not case-sensitive), as in the following example:

```
to handle keyChar key
    vQuestion = 0x3F
    if key = vQuestion then
        request "Do you wish to display Help?" with "Yes" or "Cancel"
        if It is "Yes" then
            ...    --Further statements here
        else
            forward
        end if
    else
        forward
    end if
end keyChar
```

To format numeric values as hexadecimal, octal, or binary for display, use the *format* command with a prefix to indicate the base, as the following table shows.

Prefixes for specifying numeric values

Base	Prefix
hex	@h
octal	@o
binary	@b
decimal	@d



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

The at symbol (@) must appear as the first character of the format string. For example:

```
vNumber = 10
format vNumber as "@h00"    --Hex conversion
request vNumber             --Displays "0A"

vNumber = 10
format vNumber as "@b#"    --Binary conversion
request vNumber             --Displays "1010"

vNumber = 10
format vNumber as "@o00"   --Octal conversion
request vNumber             --Displays "12"
```



For details on formatting characters, see the “Format” topic (*format* command) in online Help.

Working with constants

Constants are words that represent other values, often numbers, to make the values easier to read and enter. You can use these types of constants in OpenScript.

Types of constants used in OpenScript

Constant type	Examples
Color (HLS or RGB values)	<i>red, green, blue</i>
Control characters and punctuation	<i>quote, tab, space, CRLF, EOF</i>
Keystrokes	<i>keyA, keyUpArrow, keyLeftButton, keySlash</i>
Mathematical expressions	<i>pi</i>



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

The following examples illustrate constants:

```
my fillColor = green
--Uses CRLF as constant
text of field "Address" = vAddress & CRLF & vCity
--Uses keyRightButton as constant
if keyState(keyRightButton) is "down" then
    ...    --Further statements here
end
```



For a complete list of constants, refer to the “Color constants (table)” topic in online Help.

For details about the HLS and RGB values used in color constants, see “Referring to colors” later in this chapter.

About keystroke constants

OpenScript includes keystroke constants that are numeric equivalents for all keyboard keys and mouse actions (left-button and right-button clicks). Examples of keystroke constants include:

<i>keyA</i>	<i>keyB</i>	<i>keyLeftButton</i>	<i>keyTab</i>
<i>keyF1</i>		<i>keyNumpad1</i>	<i>keyUpArrow</i>
<i>keyHome</i>		<i>keyNumLock</i>	

Use the keystroke constants with the *keyState()* function to determine whether the user has pressed a specific key or mouse button.

For example:

```
--This handler displays the unique ID of the object under the
--cursor while the left mouse button is held down
to handle buttonDown
    while keyState(keyLeftButton) is down
        caption of statusBar = \
            objectFromPoint(mousePosition of this window)
    end while
end buttonDown
```



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Working with character strings

A character string, or string, is any combination of alphabetic, numeric, punctuation, and control characters that you assign to a variable, assign as the property of an object, or use in an expression. For example:

```
ask "Enter your complete address, including postal access code" && \  
    "and country of origin."
```

Note For details about working with lists, see “Working with lists” later in this chapter.

Using string operators in expressions

To refer to part of a string, use one of the string operators *char*, *item*, *textline*, or *word*:

```
item 1 of newNames    --Gets one item out of a list in a variable  
char 2 of word 3 of textline 4 of text of field "Note" of \  
    page 27 of book "home.tbk"
```

You can identify part of a string, or a substring, using an absolute number (as shown above) or using a keyword to indicate its position: *first*, *last*, *middle* (or *mid*), *second*, *third*, *fourth*, up to *tenth*. For example:

```
third character of vAlphabet    --Gets "C" from a local variable  
last item of vNames            --Gets last item in a local variable
```

For details about string operators, see Chapter 6, “Creating an OpenScript statement.”

Tip Whenever possible, use the *char* operator to refer to part of a string because it executes much faster than *word* or *textline*.



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Using a range

To identify a range within a string, use the keyword *to*. For example:

```
vIntro = words 1 to 3 of text of field "Preamble of Constitution"
```

This statement sets the variable *vIntro* to *We, the people*, including the commas.

Converting between strings and other data types

When you use string operators with data in untyped variables, ToolBook II automatically converts the data to a string. For example:

```
x = (1000*2)  
vFirstChar = first character of x    --Returns "2"
```

You can also use strings as numbers in expressions. In the following expression, ToolBook II converts the text of a field into a number:

```
request (word 1 of text of field "Tax Rate" of page 7) * vAmount
```

You can produce a list by joining strings with a comma and then assigning the string to a variable:

```
vAlphabet = "A,B,C,D,E"
```

You can use the list commands (*pop*, *push*, *clear*, *get*, *put*, and *set*) to manipulate lists.

Note You can convert between strings, numbers, and other data types only if the data is stored in variables that were not declared using data types.



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 **Using variables, values, and text**
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Inserting or replacing text in strings

You can use string specifiers to replace text or to insert text into a string, as the following table shows. The new value does not have to be the same size as the old value.

String specifiers for inserting or replacing strings

String specifier	Description
<i>into</i>	Replaces a string
<i>before</i>	Inserts text in front of another string
<i>after</i>	Inserts text behind another string

For example:

```
--To replace the first word of a string with a phrase  
word 1 of textline 2 of text of field "Habitat" = "Large shady trees"  
  
--To insert the word "shady" after the first word, and move any  
--other words to the right  
put "shady" after word 1 of textline 2 of text of field "Habitat"
```

Working with textlines

The string operator *textline* returns a line of text, but does not return the carriage return/line feed (CRLF) that delimits the line. If you insert a new textline before or after an existing textline, use the constant *CRLF* to include a carriage return/linefeed with the new textline. For example:

```
--To start a new line and insert the specified string  
put CRLF & "average annual rainfall: 40 inches" after \  
textline 3 of text of field "Habitat"
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

If you need to manipulate the text of a field, put the text into a variable first. Extracting data from a variable is faster than extracting it from properties of an object. For example:

```
vTextOfField = text of field "Introduction"  
step ctr from 1 to wordCount(vTextOfField)  
    request word ctr of vTextOfField  
end
```

Selecting text in a field or list box

To select a string within the text of a field, use the *select* command:

```
select words 2 to 5 of text of field "Instructions"
```

When text is selected, you can act on it by referring to the *selectedText* system property:

```
select word 1 of text of field "Instructions"  
strokeColor of selectedText = red    --Sets text to red
```

The system property *selectedTextState* returns a list of information (such as the offset within a field) about the currently selected text.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

To determine what lines have been selected in a list box, use the field's *selectedTextlines* property. You can also use *selectedTextlines* to select particular lines from within a handler to set, for example, default values in a list box. The property *selectedTextlines* contains a list of one or more line numbers representing the selected lines. For example:

```
--This handler is in a button (not in the script of the list box)
--and displays the text of lines that have been clicked in a list box
to handle buttonClick
    vTextOfField = text of field "listbox1"
    vSelectionList = selectedTextlines of field "listbox1"
    step ctr from 1 to itemCount(vSelectionList)
        vNextLineNo = item ctr of vSelectionList
        vNextLine = textline vNextLineNo of vTextOfField
        request vNextLine
    end step
end buttonClick
```

Functions used with strings

You can use the following OpenScript functions to refer to or act on specific parts of a string:

<i>ansiToChar()</i>	<i>lowercase()</i>	<i>uppercase()</i>
<i>charCount()</i>	<i>offset()</i>	<i>wordCount()</i>
<i>charToAnsi()</i>	<i>textFromPoint()</i>	
<i>itemCount()</i>	<i>textlineCount()</i>	

For example, the following statement puts a list of all but the first three finalists into a variable:

```
vConsolation = items 4 to itemCount(vFinalists) of vFinalists
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

When manipulating text in a field, improve processing speed by putting the text into a variable and performing the required operations on the variable. String functions are especially useful for such operations. For example:

```
--To find out if any character in a field is not a number
--vTxt is a local variable
vTxt = text of field "RefValues" of this background
step i from 1 to charCount(vTxt)
    if character i of vTxt is not in "0123456789"
        return false
    end if
end step
```

Working with lists

A list, or stack, is a series of items separated by commas. Lists are a good way to maintain data that consists of a series, such as names, especially if the number of items changes often. You can also use lists to store a series of items temporarily without having to create separate variables for each item.

ToolBook II has special commands and functions that work with lists, and often requires parameters to be passed as lists (such as page units or color values). You can refer to each element of a list using the keyword *item*:

```
vNames = "Amelia,Molly,Isabel,Roxanne"
get item 2 of vNames      --Puts the name "Molly" into It
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Use the commands *push* and *pop* to treat a list as a first-in-last-out stack. *Push* adds an item (including the comma delimiter) to the front of a container; *pop* removes it. For example:

```
local vGrades --Initializes to null
push "A" onto vGrades
push "B" onto vGrades --vGrades now contains "B,A"
--Puts "B" into the variable It (vGrades now contains only "A")
pop vGrades
pop vGrades into vLastItem --Puts "A" into vLastItem
```

You can use *push* to position items at a specific place in the stack:

```
push "dog" onto item 5 of vAnimals
push "cat" after item 5 of vAnimals
```

When you refer to items in a list, you can put a working copy of the list into a variable and pop items from the variable until it is *null* rather than use a step control structure to iterate through the original list. For example:

```
system svSysPositions
local vObjWinner, vMyList
--Puts value of system variable into a local variable
push svSysPositions onto vMyList
step i from 1 to itemCount(vMyList) --Checks each item in the list
    pop vMyList
    if i <> It
        break
    end if
end step
```



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Referring to colors

A color in ToolBook II is specified by a list of three numbers, representing either RGB (red, green, and blue) or HLS (hue, lightness, and saturation) units. You can set colors directly using these values or use predefined color constants for some colors such as *red*, *blue*, *black*, and *white*. You can also set a property for objects that causes them to use the Windows system colors.

You can set colors in OpenScript using either RGB or HLS units. To set a color using RGB units, set the *rgbFill* or *rgbStroke* properties of an object; to use HLS units, set the *fillColor* and *strokeColor* property. There is no difference in the result, so use the color system that you are most comfortable with. You can alternate RGB and HLS units in the same application, and even use RGB units one time and HLS units another time for the same object.

Specify RGB values as follows:

- ◆ Red values range from 0 to 255, with 0 representing 0 percent red and 255 representing 100 percent red. Pure red in RGB values is represented by 255,0,0.
- ◆ Green values range from 0 to 255, with 0 representing 0 percent green and 255 representing 100 percent green. Pure green in RGB values is represented by 0,255,0.
- ◆ Blue values range from 0 to 255, with 0 representing 0 percent blue and 255 representing 100 percent blue. Pure blue in RGB values is represented by 0,0,255.

White in RGB units is created by combining 100 percent red, green, and blue and is represented by 255,255,255. Black is created by combining 0 percent red, green, and blue and is represented by 0,0,0.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Specify HLS values as follows:

- ◆ Hue values range from 0 to 360, corresponding to the angle on a color wheel. For pure colors, 0° is *red*, 60° is *yellow*, 120° is *green*, 180° is *cyan*, 240° is *blue*, and 300° is *magenta*. Intermediate angles represent intermediate colors. 0° and 360° are equivalent.
- ◆ Lightness values range from 0 to 100, with 0 representing 0 percent lightness (*black*) and 100 representing 100 percent lightness (*white*).
- ◆ Saturation values range from 0 to 100, with 0 representing 0 percent saturation (*gray*) and 100 representing 100 percent saturation (pure color).

```
--Display in sequence all colors available on the current display
--(in HLS units)
to handle buttonClick
    step vHue from 0 to 360
        my fillColor = vHue,50,100
    end step
end buttonClick
```

Note You can get the RGB and HLS values of a color by using the Color dialog box.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Referring to color constants

You can use 10 ToolBook II color constants to refer to solid colors—*black*, *blue*, *cyan*, *green*, *magenta*, *red*, *yellow*, *white*, *gray*, and *lightGray*. Each constant is equivalent to the three numeric values that constitute either the HLS or RGB values for that color. ToolBook II assigns the correct values depending on whether you are assigning the color to a property that uses HLS or RGB values. For example:

```
my fillColor = green           --Sets to 120,50,100 (HLS)
rgbFill of rectangle "patch" = green  --Sets to 0,255,0 (RGB)
my strokeColor = 0,50,100         --Red (HLS)
```

Using Windows system colors

To color an object using the Windows-defined colors for that object type, set the object's *useWindowsColors* property to *true*. This overrides the current settings for *strokeColor* and *fillColor*, but preserves these values. If you later set the value of *useWindowsColors* to *false*, ToolBook II reverts to the values in *strokeColor* and *fillColor*.



How to use this online guide

Contents

- 1** Finding the information you need
- 2** Understanding OpenScript programming
- 3** Learning OpenScript basics
- 4** Using the scripting tools
- 5** Handling messages, events, objects, and properties
- 6** Creating an OpenScript statement
- 7** Using variables, values, and text
- 8** Writing common scripts
- 9** Handling user events
- 10** Managing data
- 11** Using special effects
- 12** Using dynamic data exchange
- 13** Using dynamic-link libraries
- 14** Debugging OpenScript programs

Index



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts**
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Chapter 8

Writing common scripts

When creating ToolBook II applications, you will need to write OpenScript programs to accomplish many common tasks, such as moving between pages, checking for errors, saving changes to applications, and creating libraries of routines to use in different applications. This chapter describes how to accomplish these tasks using OpenScript.

CONTENTS

Designing navigation	154
Checking for results and errors	163
Creating scripts in a running application	166
Saving changes to applications	170
Creating libraries of handlers using system books	173
Initializing an application	181
Setting startup properties for ToolBook II.	188



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Designing navigation

Applications commonly have buttons or hotwords that you click to navigate to other pages of your books. To move to a particular page, use the *go* or *send* commands in the script of a button or hotword. You can also navigate to a page relative to the current page (*send next*, *send previous*) or to an absolute location in the book. When possible, navigate to a page by name, so you do not have to change the script if you reorder the pages:

```
--Examples of navigating using relative locations
send next           --Goes to the next page
go previous page    --Goes to the previous page

--Examples of navigating using absolute locations
go to page 1
go to page 1 of this background

--Script for a "Contents" button
to handle buttonClick
    go to page "Contents"
end
```

If the user is already on the last page of the book when the *go next page* command is executed, ToolBook II automatically navigates to the first page of the book. You can create a handler that prevents ToolBook II from wrapping from the last to the first page:

```
to handle buttonClick
    if pageNumber of this page < pageCount of this book then
        go next page
    else
        request "This is the last page."
    end if
end buttonClick
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 **Writing common scripts**
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Using a variable as the name of a page

To create a flexible navigation system in your book, you can use a variable to pass the name of a page. For example, if you want a user to be able to click a hotword to display a page that defines the hotword, put the definition on a page named after the hotword and use a handler such as the following to navigate to the definition:

```
--Place this handler in a book script  
to handle buttonClick  
    if object of target is "hotword"  
        go to page(text of target)  
    end if  
end buttonClick
```

Initializing pages and objects when navigating

Typically, you need to set up pages so that they are in a predictable state each time they are displayed. Types of initialization you might need include:

- ◆ setting the focus to a specific button or field.
- ◆ building menus for specific pages or backgrounds.
- ◆ running animation and special effects related to changing pages.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

When you navigate from page to page, ToolBook II sends leave-event and enter-event messages to the affected pages and, if necessary, to the backgrounds. You can write handlers for these messages to initialize or clean up pages as you navigate to or from them. ToolBook II sends leave-event and enter-event messages in this order:

<i>leaveField</i> , <i>leaveCombobox</i> , or <i>leaveButton</i>	sent only at Reader level
<i>leavePage</i>	sent always
<i>leaveBackground</i>	sent if necessary
<i>enterBackground</i>	sent if necessary
<i>enterPage</i>	sent always
<i>enterField</i> , <i>enterCombobox</i> , or <i>enterButton</i>	sent only at Reader level

Placing scripts that initialize pages

In most cases, the best way to initialize a page is to set it to its original state in the *leavePage* handler for that page, or in the *enterPage* handler for the next page.

Set *lockScreen* or *sysLockScreen* to *true* in the *leavePage* handler to prevent a user from seeing the cleanup actions. Use *lockScreen* to prevent updates to a page in a viewer, or *sysLockScreen* to prevent updates to any page. (*lockScreen* and *sysLockScreen* are automatically set back to *false* when ToolBook II returns to an idle state.) For example:

```
to handle leavePage
  sysLockScreen = true           --Prevents screen updates
  --The following three lines restore the page to a predictable state
  move ellipse "animate1" to 100,100
  text of field "Student" = null  --Makes student's name null
  selectedTextlines of field "MoreInfo" = null
  forward
end
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 **Writing common scripts**
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Where you choose to place these handlers depends on whether you want the action to affect all pages in a book, all pages that share a background, or only a specific page. You can place handlers for *enterBackground* and *enterPage* messages in book, background, and page scripts.

Initializing an object automatically

You can create objects that respond automatically to enter-event and leave-event messages by using a notify handler. The advantage is that you do not need to change the scripts for the page or book because the object can automatically be notified for *enterPage* and *leavePage* messages. For example, the following notify handler for the *enterPage* message initializes a page by disabling the Next button if the user is on the last page:

```
notifyBefore enterPage
  if pageNumber of this page = pageCount of this book then
    enabled of button "Next" = false
  else
    enabled of button "Next" = true
  end if
end enterPage
```

For more information about notify handlers, see “Creating self-contained objects using notify handlers” in Chapter 5, “Handling messages, events, objects, and properties.”



Contents

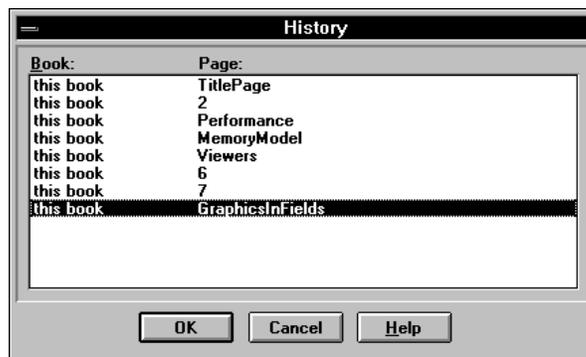
- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Using *sysHistory* to maintain a list of pages

Each time the user navigates from a page in the main window (not in a viewer), ToolBook II adds the *uniqueName* of that page to the front of the system property *sysHistory*. To return to the last page visited, send the message *back*, or use the command *go to item 1 of sysHistory*. Both methods use *sysHistory* to determine what page to visit next.

Note Using the *go* or *send back* command adds the name of the page you are leaving to *sysHistory*, so that repeated *send back* or *go* commands flip back and forth between two pages.

Figure 1 As a user navigates, ToolBook II adds page names to *sysHistory*



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Manipulate *sysHistory* as you would any list, adding pages, displaying them, or removing them as required by your application. For example:

```
--Puts each item from sysHistory item into a CRLF-delimited variable  
to handle buttonClick  
pageListing = null  
step ctr from 1 to itemCount(sysHistory)  
    --Checks for duplicates  
    if item ctr of sysHistory is not in pageListing then  
        put item ctr of sysHistory & CRLF after pageListing  
    end if  
end step  
--Puts the contents of the variable into a list box field  
text of field "Already done" = pageListing  
end buttonClick
```

Preventing ToolBook II from updating *sysHistory*

To prevent ToolBook II from adding pages to *sysHistory*, set the system property *sysHistoryRecord* to *false*. This restricts the user from navigating using the *send back* command. You can also navigate to a page in your script, such as an index page, but hide the navigation from the user.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Flipping pages automatically

You can use the *flip* command to move from page to page automatically. This command is useful if you are writing self-running demonstrations or tutorials, or if you are creating an animation sequence by moving rapidly from page to page. For example:

```
--This handler flips through all pages in a book,  
--pausing two seconds at each page  
to handle enterBook  
    done = false  
    while done is not true  
        flip 1  
        pause 2 seconds  
        --Quits when user holds the left mouse button for a few seconds  
        if keyState(keyLeftButton) is "down" then  
            request "Do you wish to quit?" with "Yes" or "No"  
            if It is "Yes" then  
                done = true  
            end if  
        end if  
    end while  
    send exit    --Quits ToolBook II  
    forward  
end enterBook
```

Note While the *pause* command is executing, no mouse clicks or keystrokes are processed; therefore, use *pause* to create a delay only when you are sure a user will not want to click objects or enter characters from the keyboard while the system is paused.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 **Writing common scripts**
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

As an optimization technique, you can also flip through all pages in a book to preload them into memory. For example:

```
--This handler preloads all the pages from a book into memory  
to handle enterBook  
  sysLockScreen = true      --Prevents pages from displaying  
  flip all                 --Flips through all pages  
  sysLockScreen = false  
  forward  
end
```

Restricting navigation with the *skipNavigation* property

You can prevent users from navigating to a page by setting its *skipNavigation* property. For example, in a tutorial application, you can set *skipNavigation* to *true* for the contents page so that if users move through the lesson page by page, they skip that page.

If the *skipNavigation* property of a page is set to *true*, you can navigate to it using explicit statements, such as *go to page "topics"*, *send back*, *go to previous page*, or *go to next page*. Implicit statements, such as *send next*, *send previous*, *send first*, *send last*, or *flip* will skip the page.



For details about *skipNavigation*, refer to the entry for *skipNavigation* in the Openscript reference in online Help.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Creating transition effects

When you navigate to another page, you can use the *transition* command, or the *fxDissolve*, *fxWipe*, or *fxZoom* commands, to have ToolBook II display a transition effect—such as fading or wiping—before displaying the next page. For example, if you are displaying bitmaps, you can use transition commands to fade to black between pages to avoid palette shift, as this example shows:

```
--This example shows a gradual dissolve to the next page  
to handle buttonClick  
    transition "dissolve slow" to next page  
end
```

```
--This illustrates fading to black, then splitting to the next page  
to handle buttonClick  
    transition "fade slow" to black  
    transition "split horizontal fast" to next page  
end
```

Note The *transition* command ignores any *skipNavigation* command set for a page.



For details about transitions, refer to the entries for *fxDissolve*, *fxWipe*, *fxZoom*, and *transition* in the Openscript reference in online Help.



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Checking for results and errors

You can determine whether ToolBook II successfully completed statements in your scripts by checking the values of the variables *sysError* (error message text) and *sysErrorNumber* (an error number only). ToolBook II puts values into these variables if a statement results in an error. In addition, some statements such as *request* and *ask* set these variables to indicate whether a user clicked Cancel in a message box.

Note You should check the number in *sysErrorNumber* rather than the text in *sysError* if you intend to translate your application into another language, such as German or French.

In the following example, if ToolBook II cannot complete the *go* command, *sysError* is set to *Cannot find page "Index"* and *sysErrorNumber* is set to *8012*:

```
sysSuspend = false           --Turns off execution-suspended messages
go to page "Index"           --If there is no such page, there is an error
request sysErrorNumber && sysError
--Shows how you can detect whether the user clicked "Cancel"
--in the message box displayed by the ask command
ask "Enter name"
if sysError = "cancel" then
    --User pressed the "Cancel" button instead of entering a value
    ...      --Further statements here
end
```

In general, ToolBook II sets *sysError* to *null* and *sysErrorNumber* to zero or *null* if a statement is successful. (The *request* and *ask* commands set *sysError* to *ok* instead of *null*).



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 **Writing common scripts**
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Scope of system error values

The values in *sysError* and *sysErrorNumber* persist between handlers. To ensure that you do not inadvertently read values put into these variables by earlier statements, clear the variables at the beginning of your script:

```
clear sysError  
sysErrorNumber = 0
```

Because these two variables are reset often by various commands, copy their values to other variables immediately after the command you are checking if you want to examine the value later.

Handling errors in your scripts

If ToolBook II cannot continue executing your script, it generates an error and displays the Execution Suspended message. To handle fatal errors as part of your application, you can suppress the display of this message. Handling errors in your script is useful when you want to create your own error messages, or when you are trying to force an error, such as when validating user input.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 **Writing common scripts**
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

To suppress the Execution Suspended message, set *sysSuspend* to *false*. ToolBook II will ignore any errors and instead load error values into *sysError* and *sysErrorNumber*. You can then check the values of these variables as part of your script. To be sure that ToolBook II continues to monitor errors, set *sysSuspend* back to *true* as soon as possible. For example:

```
--This handler is a user-defined function to validate a date
to get validateDate userInput
    returnValue = true                --Default return value
    --Clear residual value in error variable
    sysErrorNumber = 0
    --Disable default error checking
    sysSuspend = false
    format date userInput from "m/d/y" --Try to generate an error
    --Enable error checking right away
    sysSuspend = true
    if sysErrorNumber is 8050 then
        returnValue = false          --The date was not valid
    end if
    return returnValue
end validateDate
```

For details about how to use the options in the Execution Suspended message, see Chapter 14, "Debugging OpenScript programs."



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Creating scripts in a running application

You can create scripts in your application while it is running to add scripts to objects at Reader level, or to include information such as the name of a property to display. You can create scripts at run time in the following ways:

- ◆ Use an *execute* statement to create and run one or more complete OpenScript statements. For example:

```
execute "move button myButton by 500,500; \  
        move field myField by 200,200"
```

- ◆ Use the *evaluate* function to get the value of an expression while the script is running. For example:

```
x = evaluate(y bitOr 4)
```

- ◆ Set the *script* property of an object to add a new script to it.

Note Creating scripts at run time is a powerful feature of ToolBook II. However, because there is some performance overhead, we recommend that you create scripts at run time only when absolutely necessary.

Constructing statements at run time

Use an *execute* statement to construct and run OpenScript statements while the application is running. For example:

```
--This script gets the text property of a record field and  
--runs the text as a script  
myScript = text of recordField "Script"  
execute myScript
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

When creating statements to be run with the *execute* command, use these guidelines:

- ◆ Statements following *execute* run as if they are part of the script in which the *execute* statement appears. Variables in the *execute* statement act as local variables to the handler as a whole.
- ◆ You cannot include a *break to system* command in the *execute* statement.
- ◆ It can be more difficult to track errors in an *execute* statement because the ToolBook Debugger identifies the *execute* statement itself as the location of the problem.
- ◆ ToolBook II recompiles the code following the *execute* command each time the statement runs, so there is some performance overhead. Do not use *execute* inside a loop because the overhead of recompiling is compounded at each iteration of the loop.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 **Writing common scripts**
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Evaluating expressions

To evaluate expressions while an application is running, use the *evaluate* function. This function allows you to construct an expression—for example, the name of a property or a mathematical expression—and get its value while the application is running. For example:

```
--Asks user for an object and a property and displays the value
--of the requested property
to handle buttonClick
    ask "Examine what object? (use this format: object <identifier>)"
    if sysError = "cancel" then
        break
    end if
    vObject = It
    ask "What property do you want to see?"
    if sysError = "cancel" then
        break
    end if
    vProp = It && "of" && vObject
    request evaluate(vProp)
end buttonClick
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

The following example combines the *execute* and *evaluate* commands in a script that copies all of the user properties of one object to another object. This example is a good illustration of the use of these statements, as there is no other way to accomplish this task.

```
to handle copyProps fromObject, toObject
  --Get the list of userProperties from source object
  uProps = userProperties of fromObject
  step i from 1 to itemCount(uProps)
    pop uProps into curProp
    --Get the value of property in curProp
    curPropVal = evaluate(curProp && "of" && fromObject)
    --Construct the execute string
    ex = curProp && "of" && toObject && "=" && \
      quote & curPropVal & quote
    request ex      --Display for debugging to see if
                  --it looks valid

    execute ex
  end step
end copyProps
```

Evaluating the names of messages

If you need to evaluate the name of a message for use with the *send* command, do not use the *evaluate* function. Instead, place the name of the message in parentheses. For example, the following handler gets the text of a field and sends it as a message. If no parentheses surrounded *It*, ToolBook II would send the literal value *It* instead of the contents of the variable.

```
to handle buttonClick
  get text of field "message"
  send (It)
end
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Setting the *script* property of an object

Because the script of an object is a property, you can set it and get it like any other property. Set an object's *script* property to build scripts at run time and send messages to run them.

The following example shows how you can use a record field to enter scripts for testing. First, create a record field called "Script". Then, create a button and enter the following as its script:

```
to handle buttonClick
  local temporaryScript
  temporaryScript = text of recordField "SCRIPT"
  --The next two statements put a handler control structure
  --around the script
  put "to handle runMyScript" & CRLF before temporaryScript
  put CRLF & "end" after temporaryScript
  script of recordField "SCRIPT" = temporaryScript
  send runMyScript to recordfield "SCRIPT" of this background
end
```

Saving changes to applications

You can save changes to the current application at any point using the *save* or *save as* commands. You can also control whether ToolBook II saves changes when a user closes the book.

Saving changes explicitly

Use the *save* command to save the current changes to a book:

```
save changes to this book
save changes to book "mysysbook.sbk"
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

To save the application under another name, use the *save as* command:

```
to handle buttonClick
    ask "Enter the name you want to save under:"
    if sysError is not "cancel" then
        overWrite = true
        save as It, overWrite
    end if
end buttonClick
```

Tip When you use the *save* command, ToolBook II simply adds changes to the end of the file. When you use the *save as* command, ToolBook II compacts the book to its most efficient size before saving it. Use *save as* periodically to ensure that your book runs optimally.

Saving changes when a book closes

You can control whether ToolBook II saves changes when a user closes a book by setting the properties *saveOnClose* and *sysChangesDB*. By setting these properties, you can also determine whether ToolBook II prompts the user before saving changes.

Set *saveOnClose* to one of the values listed in the table on the following page. The *saveOnClose* property is persistent, retaining its setting when you reopen the book.



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Values for the *saveOnClose* property

Value	Meaning
<i>Yes</i>	Save changes before closing.
<i>No</i>	Don't save changes before closing.
<i>Ask</i>	Display the "Save current changes?" message before closing.
<i>System</i>	Don't save changes; display a message according to the setting in <i>sysChangesDB</i> . This is the default value.

Note ToolBook II closes a book when you open or start a new book, exit ToolBook II, or close a viewer displaying a page from the book.

Setting *sysChangesDB* to suppress messages

If the value of *saveOnClose* is *system*, ToolBook II uses the value of *sysChangesDB* to determine whether to prompt the user to save changes when closing the book. If *sysChangesDB* is *true* (the default), ToolBook II displays the "Save current changes?" message. If *sysChangesDB* is *false*, ToolBook II closes the book without displaying any messages.

Important If *sysChangesDB* is set to *false* and *saveOnClose* is set to *system*, ToolBook II closes the current book and discards unsaved changes without warning.

The value of the *sysChangesDB* property applies to the current ToolBook II instance, even across books. However, it is set to its default value of *true* when you start your next instance of ToolBook II.



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 **Writing common scripts**
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Creating libraries of handlers using system books

A system book is any book that you add to the *sysbooks* property of the current book. You can use system books as libraries to store handlers, such as user-defined functions, for use with multiple applications.

Creating a system book

To create a system book, start a new book, and then place the handlers you want to share between books into the book script of the new book. Save the system book as you would any book. By convention, system books are saved with the extension *.sbk* (instead of *.tbk*), but you can use any file extension.

You can edit the script of a system book directly using the *edit* command in the Command window of any book:

```
edit script of book "c:\sysbooks\animate.sbk"
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Adding a system book to the current book

To use a system book, add its name, including the extension, to the *sysBooks* property of the current book:

```
push "c:\sysbooks\animate.sbk" onto sysBooks
```

Important Always use *push* or a similar command to add books to *sysBooks*. Simply setting *sysBooks* to a new value overwrites its contents, potentially removing system books already defined for the application. For details about how to set *sysBooks* without overwriting its contents, see the next topic.

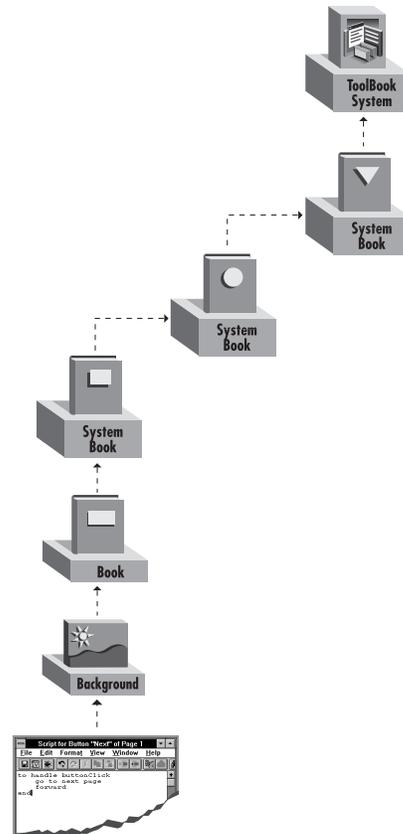
The book script of the system book becomes part of the object hierarchy for the current book. If a message is not handled in the current book, ToolBook II searches for a corresponding handler in the book script of the system book. Only the book script of the system book becomes part of the object hierarchy, although handlers in the system book script can redirect messages by sending them to other objects in the system book.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 **Writing common scripts**
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Figure 2 Object hierarchy



You can add as many system books as you require to the *sysBooks* property. ToolBook II searches for handlers in system books in the order the books appear in the *sysBooks* property.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

When ToolBook II adds the name of a system book to the *sysBooks* property, it sends a *linkSysBook* notification message to the book just linked. You can write a handler for this message in the system book to run any statements that the system book requires for initialization. For example, if the system book uses a DLL function, you can write a handler for the *linkSysBook* message to link the DLL.

ToolBook II sends an *unlinkSysBook* message when you remove a system book. You can write a handler for that message to unlink DLLs, clear variables, or perform other cleanup activities.

Note Do not forward the *linkSysBook* and *unlinkSysBook* messages from a system book, or other system books may rerun their handlers for these messages.

Loading a system book when starting an application

To make a system book available when you first start an application, write an *enterBook* handler for the application that loads *sysBooks*. The system book is then immediately available to the application, and the application can send messages to the system book for further initialization. For example:

```
to handle enterBook
    push "c:\mybooks\messages.sbk" onto sysBooks
    forward
end
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

You can define default system books in your Inst60.ini file by setting the value of the System Books entry in the Startup Options section.

If you have defined startup system books, be sure that any system book you want to add is not already in the System Books entry.

```
--Requests a system book to be added to startup system books  
to handle enterBook  
    send addToSysBooks "mybook.sbk" to this book  
    forward  
end  
  
--Adds a book to sysBooks if it isn't already there  
to handle addToSysBooks newBook  
    if newBook is not in sysBooks then  
        push newBook onto sysBooks  
    end if  
end addToSysBooks
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 **Writing common scripts**
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Removing a system book

You can remove one book from the *sysBooks* property and leave the others intact. For example:

```
to handle removeSysbook fBookName
  step i from 1 to itemcount(sysBooks)
    --The partial-string comparison in the next line
    --ignores the path
    if (fBookName is in item i of sysBooks)
      clear item i of sysBooks
      break step
    end if
  end step
end removeSysbook
```

Writing handlers for the system book

You should write handlers for a system book that are as generic as possible for multiple applications. For example, always forward built-in messages that you are handling, such as *enterBook*, unless you are writing a system book handler to trap a particular message. When writing system book handlers, keep these points in mind:

- ◆ If you handle enter-event messages (*enterBook*, *enterPage*) and leave-event messages (*leaveBook*, *leavePage*), always forward them. If there are multiple system books in the hierarchy, each system book can initialize or clean up in response to these messages.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 **Writing common scripts**
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

- ◆ If you link a DLL function in a system book, always assign a unique alias to the function so there is no conflict if another system book (or book handler) prototypes that function differently. For details about assigning aliases to DLL functions, see Chapter 13, “Using dynamic-link libraries.”

When a handler in a system book is running, the context for the keyword *this* is the current page of the target window. For example, if the main window is the target window, then *this book* refers to the book open in the main window. Or, if a viewer displaying a page from a system book is the target window, then *this page* refers to the page from the system book and *this book* refers to the system book. The keywords *my* and *self* always refer to the object whose script is currently executing—in this case, the system book.

About system books and performance

ToolBook II opens system books only when looking for a handler, and so the number of system books you specify does not affect ToolBook II performance. However, performance can slow if ToolBook II has to open and search many system books. To keep your scripts running as quickly as possible, specify only the system books you really need and make sure they contain only necessary handlers. Put the most heavily used system books at the front of *sysBooks*.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

To minimize the number of handlers in the hierarchy, keep only essential handlers in the book script of the system book. If you have handlers for user-defined functions or for messages that are unique to that system book, put them into page scripts of the system book, and then forward the message to those pages. For example:

```
--This handler must be in the hierarchy, so it belongs
--in the book script of the system book
to handle enterBook

--Forwards a user-defined message to a page in this system book
    send initializeMe to page 1 of self
    forward
end

--This script (in the script of page 1 of the system book)
--is not in the object hierarchy
to handle initializeMe
...    --Further statements here
end
```

If you need to call a function in a system book only occasionally, you can avoid putting it into *sysBooks* and call it directly instead:

```
get nextMultimediaDevice("videodisc") of book "mmutils.sbk"
```



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 **Writing common scripts**
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Initializing an application

Initializing an application is the process of running scripts when the user starts the application. During initialization, you can:

- ◆ establish the opening screen of the application. For example, show an opening screen by displaying a viewer that contains a page with a logo or other opening artwork.
- ◆ start the application at Reader level. Send the *reader* message to initialize menus and present the application to the user, ready to run.
- ◆ declare and initialize system variables. For example, prompt for a password at initialization and then store a security level in a system variable for later use.
- ◆ set system properties. For example, set *sysDrawDirect* or *sysFontFace* so that objects are created in styles you customarily use.
- ◆ define menu items. For example, add a menu item for a custom development tool to the Author-level menu when ToolBook II is initializing.
- ◆ link DLLs. For example, if your application works with DOS files, link functions from *Tbdos.dll* while the application is initializing. For details about linking DLLs, see Chapter 13, “Using dynamic-link libraries.”
- ◆ specify system books. For example, add a system book to handle messages from a custom menu. For details about system books, see “Creating libraries of handlers using system books” earlier in this chapter.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

- ◆ set the focus. For example, set the *focus* property to a specific button or field to prepare the application for data entry.
- ◆ define the main window size and location. For details, see “Defining window size and position” later in this chapter.

The following sections describe how to write handlers for initialization, streamline the initialization process, initialize separately for Reader and Author levels, position a window during initialization, and restore ToolBook II to its default state.

Writing handlers for initialization

ToolBook II sends three enter-event messages for which you can write initialization handlers: *enterSystem*, *enterApplication*, and *enterBook*. You can use these messages to write initialization handlers for ToolBook II as a whole, the primary book in an application, or additional books in an application, respectively.

Using *enterSystem*

ToolBook II sends the *enterSystem* message when a new instance of ToolBook II starts. Write handlers for this message to establish conditions that are true for ToolBook II as a whole, no matter what book you are running, including initializing system properties, linking DLLs, loading system books, or customizing Author level by adding menu items.

To use this message, put a handler for it in a startup system book or in the script of a book you specify in the command line when you start ToolBook II.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

You can determine whether the user included a file name or other information when starting ToolBook II by checking the value of the `<cmdLine>` parameter for the `enterSystem` message. You can use this parameter to define command-line switches for your application. For example:

```
--Initializes ToolBook II in different states depending on the
--command-line switch the user started with
to handle enterSystem cmdLine
  if "/a" is in cmdLine then      --Begin at Author level
    send author
  end if
  if "/r" is in cmdLine then     --Begin at Reader level
    send reader
  end if
  if "/co" is in cmdLine        --Begin with Color Tray visible
    visible of colorTray = true
  end if
  forward
end enterSystem
```

Note You can establish default values for a number of system properties by setting startup system properties. For details, see “Setting startup properties for ToolBook II” later in this chapter.

Using *enterApplication*

An *enterApplication* message is sent when a book opens in the main window and just before ToolBook II displays a page from that book in the main window. Use this message to write handlers for initializing the application, such as declaring and initializing application-specific system variables, customizing the Reader-level menu, showing an opening screen, setting the focus, and hiding or showing objects.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 **Writing common scripts**
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

ToolBook II sends the *enterApplication* message to the page displayed in the main window. You can put a handler for this message in the script of the opening page of the book or anywhere further up in the object hierarchy.

Using *enterBook*

ToolBook II sends an *enterBook* message when a book is opened in any viewer and just before it displays a page, whether in the main window or in a viewer. Use the *enterBook* message to write handlers that initialize a book by itself—regardless of whether it is the main book in an application—such as choosing text to display based on the current language set, changing the size of the background depending on the current display device, and setting colors depending on the colors in the main window.

The difference between *enterApplication* and *enterBook* is that ToolBook II sends the *enterApplication* message only when a page of a new book is displayed in the main window. For example, if you display a page from a book in a viewer other than the main window, ToolBook II sends an *enterBook* message to that book, but not an *enterApplication* message. Therefore, if ToolBook II sends an *enterApplication* message when you open a book, it will also send an *enterBook* message.

ToolBook II sends the *enterApplication* and *enterBook* messages to a book the first time it displays a page from that book, but it does not send these messages if you get a property from another book or edit a script from another book without displaying one of its pages.

ToolBook II closes the book and sends leave-event messages (*leaveApplication*, *leaveBook*) when no more pages from a book are being displayed. If a book is closed but you reopen it to display one of its pages, ToolBook II sends an *enterApplication* message (if you display the page in the main window) and an *enterBook* message again.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Streamlining the *enterApplication* handler

Because ToolBook II does not display the starting page of an application until it finishes executing the *enterApplication* handler, limit the *enterApplication* handler of a book to statements that determine the look and actions for an application. To streamline the *enterApplication* handler and speed display of the first page of the application, follow these guidelines:

- ◆ If a DLL or system variable is used only in a few places in the application, link the DLL or initialize the variable lower in the hierarchy, closer to where it is used.
- ◆ Move statements to an *enterPage* or *firstIdle* handler, if possible. The first page displays before the *enterPage* message is sent. To ensure that the process of displaying the page is complete, run media, such as an opening animation, in a *firstIdle* handler. Leave only the statements in the *enterApplication* handler that must run before ToolBook II displays the first page.

Initializing Reader and Author levels differently

You can initialize Reader and Author levels differently. At Reader level, you can set up the application so that a user can perform specific tasks—for example, data entry—right away. At Author level, you can set up custom tools for application development and set default values for system properties. To start the application at Reader level, send a *reader* message from the *enterApplication* handler.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

To initialize each level differently, create handlers for the *reader* and *author* messages. By programming these tasks into *reader* and *author* handlers, you ensure that they are accomplished every time a user switches between levels in the application. For example:

```
to handle reader
  hide menuBar
  hide scrollBar
  send sizeToPage
  clear sysBooks
  sysChangesDB = false
  forward
  focus = field "Quantity"
end

to handle author
  show menuBar
  send sizeToPage
  show scrollBar
  push "dev1.tbk" onto sysBooks      -- Adds a system book
  sysChangesDB = true
  sysLevel = author                -- Or "forward"
end
```

You can also control the working state at Author level by specifying the values for startup system properties in the Tb70.ini file, rather than in individual scripts. For details, see “Setting startup properties for ToolBook II” later in this chapter.



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Defining window size and position

By default, when a user first opens a book, the main window is centered and just large enough to display the starting page of the book. You can change the page size and both the size and location of the main window for a ToolBook II instance.

- ◆ To change the page size, set the book's *size* property, or set the page size in the Book Properties dialog box.
- ◆ To change the size of the main window so that it fits a new page size, send the *sizeToPage* message. Send this message directly after you change the page size from a script. This message is sent automatically when a book is first opened. Switching working levels and hiding and showing the menu bar also change the page size slightly.
- ◆ To change the position of the main window, set its *position* property. These statements align two open instances of ToolBook II on a VGA screen:

```
position of mainWindow = 0,0  
executeRemote "set position of mainWindow to 240,0" \  
  application "ToolBook" topic "book2.tbk"
```

- ◆ To change both the size and position of the main window, set its *bounds* property. This statement displays a page so that it covers the entire screen on a 640-by-480 VGA monitor:

```
bounds of mainWindow = 0,0,640,480
```

The dimensions of most VGA screens are 640 by 480 pixels; higher resolution VGA screens are 800 by 600 or 1024 by 768. You can get the resolution of the screen by calling the DLL functions *verticalDisplayRes()* and *horizontalDisplayRes()* in *Tbdos.dll*.



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Setting startup properties for ToolBook II

Default values for certain ToolBook II system properties are established by settings in the Tb70.ini file. You can get or set the settings in this file by changing the values of the startup system properties in any script. Examples of startup system properties include:

- ◆ *startupBook* The name of the book that opens automatically when a user runs ToolBook II.
- ◆ *startupReaderRightClick* A property that determines whether—by default—shortcut menus will appear at Reader level when a user right-clicks an object.



For a complete list of startup system properties, refer to the “Startup system properties” topic in online Help.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events**
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Chapter 9

Handling user events

This chapter discusses how to work with events that a user generates by clicking a mouse button or typing at the keyboard, and how to modify the ToolBook II default responses to these events.

CONTENTS

Handling mouse events	190
Handling keyboard events	201



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 **Handling user events**
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Handling mouse events

When a user presses or releases a mouse button, or moves the mouse pointer over an object, ToolBook II sends mouse-event messages. By writing handlers that respond to mouse-event messages, you can define what occurs in your application when a user clicks a button, field, or other object. For example, you can write a mouse-event handler that navigates between pages or triggers an animation. You can test for and respond to the following:

- ◆ Which mouse button has been clicked
- ◆ Whether the user has released the mouse button
- ◆ Whether the user has pressed a mouse button and is holding it down
- ◆ What object the pointer is over
- ◆ What object the user has clicked
- ◆ What line of a list box or field the insertion point is in when the user clicks the mouse

ToolBook II also sends mouse-event messages when a user presses keys as a substitute for mouse clicks. For example, ToolBook II sends a *buttonClick* message if the user presses the SPACEBAR while the focus is in a button, which is equivalent to clicking the button.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Handling mouse-event messages

You can write handlers for any of these mouse-event messages:

<i>buttonDown</i>	<i>mouseEnter</i>
<i>buttonUp</i>	<i>mouseLeave</i>
<i>buttonClick</i>	<i>rightButtonDown</i>
<i>buttonDoubleClick</i>	<i>rightButtonUp</i>
<i>buttonStillDown</i>	<i>rightButtonDoubleClick</i>

If you want to define how an object responds when clicked, write a handler for the *buttonClick* message. However, you can better control the reaction of an object to a click by writing handlers for a:

- ◆ *buttonDown* message, sent when the user presses the left mouse button.
- ◆ *buttonUp* message, sent when the user releases the left mouse button.

For example, you can write a handler for the *buttonDown* message to create animation sequences that run only when the mouse button is held down.

ToolBook II does not send the *buttonClick* message if the user clicks an object but moves the mouse pointer off the object before releasing the mouse button. Therefore, write a handler for the *buttonUp* message if you want the handler to run no matter where the pointer is when the mouse button is released. For example, you can use the *buttonUp* message in a page script so the user can click anywhere on the page to stop an OLE object from playing.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling red event events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

If the user holds the mouse button down, ToolBook II sends continuous *buttonStillDown* messages until the button is released. For example:

```
--This handler moves the target object across the screen as long
--as the user holds the left mouse button down
to handle buttonStillDown
  move self by -75,0
  --If the object reaches the left edge, move it back to the right
  if item 1 of bounds of self < 0 then
    move self to 7000,item 2 of bounds of self
  end if
end buttonStillDown
```

When a user clicks the mouse button, ToolBook II can generate other messages as well, depending on what object is clicked and how its properties are set. For example, if the user clicks a field, the *activated* property of the field determines which messages are sent. If the *activated* property is *true*, then the field gets the usual mouse messages. If the *activated* property is *false*, then normal mouse messages are not sent, but other messages, such as *enterField*, might be sent.

Note If the *fieldType* of a field is set to *singleSelect* or *multiSelect*, then the *activated* property is ignored and normal mouse messages— and other messages, such as *enterField*—are sent to the field.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Using mouse-event message parameters

ToolBook II sends three parameters with mouse-event messages:

- ◆ The location of the mouse pointer in page units at the time the user generates the mouse event
- ◆ A logical value (*true* or *false*) indicating whether the user held down the SHIFT key while clicking
- ◆ A logical value (*true* or *false*) indicating whether the user held down the CTRL key while clicking

You can include parameter variables in mouse-event handlers to make use of the values that ToolBook II passes with the messages. For example:

```
--This handler displays the location where the mouse is clicked
--and notes whether the user held down SHIFT and CTRL
to handle buttonClick loc, shift, ctrl
    message = null
    if shift is true then
        message = message && "with the shift key"
    end if
    if ctrl is true
        message = message && "with the ctrl key"
    end if
    request "You clicked at" && loc && message
end buttonClick
```

Note If a user does not click a button object but instead presses an access key, the SPACEBAR, or the ENTER key, ToolBook II sends a *buttonClick* message, but the *<location>* parameter does not return the pointer position. Instead, it returns the value *-1,-1*.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Displaying shortcut menus at Reader level

When you right-click an object at Author level, ToolBook II displays the shortcut menu for that object. You can have ToolBook II display shortcut menus at Reader level as well by setting the system property *sysReaderRightClick* to *true*. Displaying shortcut menus at Reader level makes it easy for you to change object properties while you are running or testing the application.

When a user right-clicks an object at Reader level (or in an *alwaysReader* viewer), ToolBook II sends a *rightButtonDown* message to that object. If the *rightButtonDown* message reaches the system, ToolBook II checks the value of *sysReaderRightClick*. If the property is *true*, ToolBook II displays the shortcut menu for the target object. If the property is *false*, ToolBook II does nothing.

Note If there is a *rightButtonDown* handler in the hierarchy between the object and the system, the message will get to the system only if the handler includes a *forward* command. An effective way to prevent users from accessing shortcut menus at Reader level (or in an *alwaysReader* viewer) is to use a *rightButtonDown* handler that does not include a *forward* command.

You can toggle the value of *sysReaderRightClick* by selecting Show right-click menus at Reader level in the Options dialog box, which is accessed from the View menu at Author level. The default value of *sysReaderRightClick* is specified by the *startupReaderRightClick* property; the default is *false*.



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Determining the position of the mouse pointer

You can use the position of the mouse pointer as user input. For example, if your application displays a pop-up menu, you can display the menu according to the pointer position. You can find the location of the pointer:

- ◆ at the moment you are testing.
- ◆ when a mouse event, such as a click, occurs.

The position of the pointer is indicated by a list of two numbers in page units. For example, *1440,1440* indicates that the pointer is one inch below and to the right of the upper-left window border.

To determine where the pointer is in the current viewer, use the *mousePosition* window property. For example:

```
--Place this handler in the book script to display a graphic
--(group of objects) that follows the pointer around the screen
--when the user moves the mouse, imitating the Mouse Trails setting
--in the Windows Control Panel
to handle idle
    system INT svOldX, svOldY
    newX = item 1 of mousePosition of this window
    newY = item 2 of mousePosition of this window
    if (newX = oldX) and (newY = oldY) then
        hide group "arrow"      --No motion, hides the graphic
    else
        show group "arrow"      --Shows the graphic and moves it
        move group "arrow" to newX+50,newY+200
    end if
    oldX = newX
    oldY = newY
end idle
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

To move the pointer to a new location, set *mousePosition* to a new value:

```
mousePosition of this window = 1440,1440
```

You can also use the *<location>* parameter of mouse-event messages to determine the position of the pointer at the time the mouse-event message is sent. For example:

```
--Put this handler in a page script  
to handle buttonClick location  
    request "You clicked at" && location  
end
```

Determining what the mouse pointer is over

You can determine what object is under the mouse pointer when the user moves the mouse or clicks an object. For example, you can display a message in the status bar as the user moves the pointer over objects in your application. You can determine:

- ◆ what object the user has clicked.
- ◆ what object the user has moved the pointer over.

To determine what object a user has clicked, check the *target* property in a mouse-event handler, which contains the *uniqueName* of the clicked object. For example:

```
--Shows what object was clicked; place this handler in a book script  
to handle buttonClick  
    request "you clicked" && target  
end
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 **Handling user events**
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

As a user moves the pointer over objects, ToolBook II sends *mouseenter* and *mouseleave* messages to the objects when the pointer crosses their boundaries. As this example shows, you can write handlers for these messages at the page level and use the *target* property to determine what object the pointer is over.

```
--Displays the name of the object under the pointer in the status bar
--as the user moves the pointer around the page
to handle mouseEnter
    if visible of statusBar is false then
        show statusBar
    end if
    caption of statusBar = target
end mouseEnter

--Clears the status bar as the pointer leaves an object's bounds
to handle mouseLeave
    clear caption of statusBar
end
```

Tip When a user presses and holds the mouse button, *mouseenter* and *mouseleave* messages are not sent.

If you need to determine what object is under the pointer while a handler that is not a mouse-event handler is running, use the *objectFromPoint()* function, as shown on the following page.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

```
--This handler in the book script displays (in the status bar) the
--name of the object under the pointer when the application is idle
to handle idle
    system oldObj          --Stores name of previous object
    if visible of statusBar is false then
        show statusBar
    end if
    currentPos = mousePosition of this window
    newObj = objectFromPoint(currentPos)
    if oldObj is not newObj then
        caption of statusBar = newObj
        oldObj = newObj
    end if
end idle
```

Determining and setting the location of the insertion point

You can determine the location of an insertion point in an editable field, set the location of the insertion point, and determine what text is next to the insertion point.

To get the location of an insertion point in an editable field, use the system property *caretLocation*, which returns a list of two numbers containing the textline and column location of the insertion point. For example:

```
--Displays location of the insertion point in a field called messages
--when user leaves the current field; place this handler in the
--script of a nonactivated (editable) field
to handle leaveField
    text of field "messages" = caretLocation
end
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

To set the location of the insertion point, set *caretLocation* to a new value:

```
caretLocation = It
```

To determine what text is next to the insertion point in a field, record field, list box, or combo box, use the *textFromPoint()* function. This function returns a list of two numbers: one represents the textline number, and the other the character offset of the insertion point. You can use the textline number returned by *textFromPoint()* with the *textline* operator to get the text of the line that the user clicked. For example:

```
--Script for a list box that shows what line was clicked  
to handle buttonClick loc  
    lineNumber = item 1 of textFromPoint(loc)  
    chosenItem = textline lineNumber of text of field "listBox"  
    request chosenItem  
end
```

Testing for mouse clicks within a script

To respond to mouse events, you usually write handlers that respond to specific messages such as *buttonClick*. However, sometimes you need to test for mouse clicks or other mouse behavior in a handler for a nonmouse event. For example, you might start specific behavior when the user presses the mouse button, continue it while the mouse button stays down, and stop when the user releases the button.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

You can use the *keyState()* function within a handler to determine whether a left or right mouse button is up or down. In the following handler from a game application, the “paddle” displayed on the screen moves either right or left, depending on which mouse button is down.

```
--Place this handler in the page script  
to handle buttonStillDown  
    if keyState(keyRightButton) is "down"  
        move rectangle "paddle" by 100,0  
        break  
    end if  
    if keyState(keyLeftButton) is "down"  
        move rectangle "paddle" by -100,0  
        break  
    end if  
end buttonStillDown
```

Changing the mouse pointer

You can change the shape of the mouse pointer to alert a user to the current state of the system. For example, many applications use an hourglass to indicate activity in process and a crosshair for drawing. You can change the pointer by setting the value of *sysCursor*:

```
to handle sortPages  
    saveCursor = sysCursor          --Saves current pointer  
    sysCursor = 4                   --Sets pointer to hourglass  
    sort pages 1 to pageCount of this book by \  
        ascending number text of recordField "postcode"  
    sysCursor = saveCursor          --Restores previous pointer  
end  
sysCursor = 14                     --Sets pointer to a big arrow
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events**
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs



For a complete list of the pointer types you can use, refer to the entry for *sysCursor* in the OpenScript reference in online Help.

You can also create a graphic pointer by assigning the value of an imported resource to *sysCursor*:

```
sysCursor = cursor "hand"
```

Handling keyboard events

By writing handlers for keyboard events, you can determine what keyboard keys the user has pressed and disable or change them, or route them to another object. For example, if the user enters a password in a record field, you can capture the characters as the user types them and display asterisks in their place.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

About keyboard events and messages

Each time a user presses a key, including keys such as SHIFT and CTRL, ToolBook II sends *keyDown* and *keyUp* messages. When a user presses a printable character, ToolBook II also sends a *keyChar* message. With these messages, ToolBook II sends the parameters: *<keyCode>*, *<isShift>*, and *<isCtrl>* to indicate which key was pressed and whether SHIFT and CTRL were also pressed. For example, the following handler displays keys in the status bar as they are typed:

```
to handle keyChar keyCode, isShift, isCtrl
  if visible of statusBar is false then
    show statusBar
  end if
  message = "key =" && keyCode
  if isShift is true then
    put space & "plus Shift" after message
  end if
  if isCtrl is true then
    put space & "plus Ctrl" after message
  end if
  caption of statusBar = message
  forward --Passes the keystroke to ToolBook II
end keyChar
```

If the focus is in a field, combo box, or button when the user presses a key, ToolBook II sends a keyboard-event message to the object. If the focus is not in a field, combo box, or button, ToolBook II sends a keyboard-event message to the page. However, if a user presses an access key, or presses the SPACEBAR when the focus is in a button, ToolBook II sends a *buttonClick* message instead of a keyboard-event message.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

If the focus is in a field or combo box when a keyboard event occurs, the default response of ToolBook II is to display the corresponding character. Otherwise, the default response is to do nothing, with these exceptions:

- ◆ If a user presses a menu accelerator key such as ALT+ARROW KEY (to navigate pages), CTRL+S (to save the current book), or F12 (to display the status bar at Reader level), ToolBook II sends the corresponding menu-event message.
- ◆ If a user presses TAB, ToolBook II moves the focus to the object on the next layer.

If you want to capture the effects of these keys, write handlers for the appropriate messages. For example, to disable the ALT+ARROW KEY combinations, write handlers for the *next* and *previous* messages.

Note You can define custom behavior for characters that are not associated with a menu item or button. For details, see “Defining access keys and custom behavior” later in this chapter.

When the user tabs between buttons and fields, the focus moves; the target of the *keyDown* message is the first object, and the target of the *keyUp* message is the second object. Therefore, be sure to test which object has the focus when using a *keyDown* or *keyUp* handler.

Tip Typing in a field slows down if ToolBook II has to call keyboard-event handlers before displaying characters. If you have a keyboard-event handler in a page script, typing will slow down for all objects on that page. Place handlers for keyboard-event messages as low as possible in the object hierarchy to affect the fewest number of objects.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events**
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Deciding which keyboard-event message to handle

Whether you write handlers for the *keyChar* message or for the *keyDown* and *keyUp* messages depends on how much detail you need about what keys the user has pressed. These messages differ in:

- ◆ what information is provided in the `<keyCode>` parameter that ToolBook II sends with the keyboard-event message.
- ◆ the sequence in which the messages are sent.

As a rule, use the *keyChar* message to write handlers that react to letters and numbers that a user types. Use *keyDown* and *keyUp* to write handlers that react to control, movement, or function keys, or that distinguish between when the user pressed and released the key.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events**
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

About values for the `<keyCode>` parameter

You can determine what key was pressed by checking the value of the `<keyCode>` parameter for a keyboard-event message. However, ToolBook II sends different `<keyCode>` values for the `keyChar` message and the `keyDown` and `keyUp` messages. For the `keyChar` message, the `<keyCode>` parameter:

- ◆ returns the ANSI value of the key the user pressed. For example, if the user types a lowercase *t*, the `keyCode` parameter for the `keyChar` message is *116*; for an uppercase *T*, the value is *84*.
- ◆ is not sent for keys that have no ANSI value, including control keys such as SHIFT, movement keys such as HOME, and status keys such as NUM LOCK. However, some control keys such as TAB and ENTER do have ANSI values, which `keyChar` returns.
- ◆ does not distinguish numbers on the number pad from those on the top row of the keyboard.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

For the *keyDown* and *keyUp* messages, the `<keyCode>` parameter:

- ◆ returns values for control, movement, function, and status keys. For example, the `<keyCode>` value for SHIFT is *16*; for the left arrow, *37*; for F8, *119*; and for NUM LOCK, *144*.
- ◆ does not distinguish uppercase from lowercase values. For example, the `<keyCode>` value in a *keyDown* handler is *84* for both uppercase and lowercase *T*. However, you can distinguish uppercase from lowercase using the `<isShift>` parameter of the message, unless the CAPS LOCK key is on. You use the *getState (keyCapital)* command to determine if the CAPS LOCK key is on.
- ◆ returns an OpenScript constant value. For example, if the user types *t*, the value of the `keyCode` parameter for the *keyDown* message is *84*, and the value for a comma is *188*.
- ◆ distinguishes numbers on the number pad from those on the keyboard's top row. For example, the `<keyCode>` value in a *keyDown* handler for *5* on the number pad is *101*; for *5* on the top row, the value is *53*.

The following handler for a *keyDown* event converts the text of a field or record field to all uppercase when the user presses SHIFT+F3. Because the handler depends on detecting a function key, you cannot trap this key combination in a *keyChar* handler:

```
--Place this handler in a book  
to handle keyDown key, isShift, isControl  
    if object of target is not in "field,recordfield,listbox" then  
        break  
    end if  
    if key is keyF3 and isShift is true  
        text of target = uppercase(text of target)  
    end if  
    forward  
end keyDown
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs



For a list of ANSI values and key constant integers, refer to the “ANSI values” and “Key constants” topics in online Help.

About the sequence of keyboard-event messages

ToolBook II sends the *keyDown* message when the user presses a key, and the *keyUp* message when the user releases it, so you can use these messages to write handlers that react separately to each portion of a keystroke. By contrast, ToolBook II sends the *keyChar* message when the user presses a key, so you can use it to write handlers that react to complete keystrokes.

The following table shows the sequence of messages ToolBook II sends when a user types *Shift+T*.

Order of keyboard-event messages sent for a single keystroke

Action	Messages sent
SHIFT pressed	<i>keyDown 16, true, false</i>
T key pressed	<i>keyDown 84, true, false</i> <i>keyChar 84, true, false</i>
T key released	<i>keyUp 84, true, false</i>
SHIFT released	<i>keyUp 16, false, false</i>

PC keyboards are auto-repeating, so when a user holds down a key, ToolBook II repeatedly sends *keyDown* and *keyChar* messages followed by a single *keyUp* message when the user releases the key.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Checking for keystrokes with an OpenScript statement

Ordinarily, if you want your application to respond when a user presses a key, you write handlers for the *keyDown*, *keyChar*, and *keyUp* messages. However, to check for keystrokes in a handler for a different event (such as a *buttonClick* event), use the *keyState()* function. For example:

```
--This handler flips all pages in a book until the user presses  
--Esc or clicks the left mouse button  
to handle buttonClick  
do  
    flip  
until keyState(keyEscape) is "down" \  
    or keyState(keyLeftButton) is "down"  
end
```

Sending keyboard-event messages from a script

You can emulate keyboard activity by sending a keyboard-event message from a script. For example, in tutorial or demonstration applications, you can send keystrokes if the user wants to watch the lesson without typing.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

To emulate keyboard activity, send the *keyChar* message. You don't have to use the ANSI value for the *<keyCode>* parameter. Instead you can:

- ◆ use the *charToAnsi()* function. For example, to send a *keyChar* message for an uppercase *T*:

```
send keyChar charToAnsi("T"), true, false
```
- ◆ substitute the OpenScript key constant for the character. For example, instead of typing *send keyChar 84, true, false* in a script, type:

```
send keyChar keyT, true, false
```

If the values of *<isShift>* and *<isCtrl>* conflict with the value of *<keyCode>* when you send a message with these parameters, ToolBook II handles the message based on the value of *<keyCode>*. For example, the statement *send keyChar charToAnsi("T"), false, false* inserts an uppercase "T" in the field that has the focus, even though the *<isShift>* parameter is *false*. If you omit the *<isShift>* and *<isCtrl>* parameters when sending a keyboard-event message, ToolBook II assumes the parameters are *false*.



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events**
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Defining access keys and custom behavior

You can define access keys for your applications that allow users to access a menu, menu item, or button by pressing the ALT key plus another character. You specify the character to be used by placing an ampersand (&) before it in the menu, menu item, or button name. For menus and menu items, you specify this character using the *setMenuName()* and *setMenuItemName()* functions. For buttons, you specify this character using the *caption* property.

You can also define custom behavior for characters not already reserved by ToolBook II. To define custom behavior for characters, write a handler for the *keyMnemonic* message and include a parameter to receive the ANSI value of the key pressed with the ALT key.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

The handler in the following example defines ALT+PLUS SIGN and ALT+MINUS SIGN as controls to change the brightness of a background color.

```
--Uses ALT+PLUS SIGN and ALT+MINUS SIGN to lighten and darken
the background
--color; place this handler in a book script
to handle keyMnemonic keyCode
  conditions
    when keyCode = 61                                --PLUS SIGN
      vColor = fillColor of this background
      if item 2 of vColor <= 95 then                  --Must be < 100
        increment item 2 of vColor by 5             --Sets lightness
        fillColor of this background = vColor
      end if
    when keyCode = 45                                --MINUS SIGN
      vColor = fillColor of this background
      if item 2 of vColor >= 5 then                  --Must be > zero
        decrement item 2 of vColor by 5
        fillColor of this background = vColor
      end if
    else
      forward
    end conditions
  end keyMnemonic
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events**
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Discarding pending keystrokes and mouse events

Users sometimes press keys or click the mouse while waiting for a process to finish. For example, if a user clicks a button to start an animation sequence but there is a short delay before it begins, the user might click the button again thinking that the first click did not work. ToolBook II stores these extra keystrokes or mouse clicks and sends them when the process is finished. However, you can throw away extra keystrokes and mouse clicks so that they do not interfere with the application by using the *flushMessageQueue()* function. For example:

```
to handle buttonClick
    send runAnimation          --Runs an animation sequence
    get flushMessageQueue()   --Throws away pending keystrokes
end
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data**
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Chapter 10

Managing data

Most applications involve data-management tasks such as formatting dates and similar data, validating user input, reading and writing files, and sorting and searching pages. This chapter describes how you can use OpenScript for these tasks.

CONTENTS

Formatting data	214
Validating data	218
Reading and writing to files	223
Searching pages	228
Sorting pages	231



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data**
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Formatting data

If you work with dates, times, and decimal or currency numbers, you can format the data to specify exactly how it should appear. Setting the format for these types of data allows you to:

- ◆ display the data in your choice of output formats. For example, you can display dates in a format such as Jan 1, 1993, or in another format.
- ◆ calculate correctly by converting the data to a format suitable for doing math. For example, you can strip all currency punctuation from a number before adding to it.
- ◆ compare values correctly by putting them into a consistent format.

Formatting data for output

Use the *format* command to convert dates, times, or numbers to a different format for display, as in this example:

```
totalValue = "2000000"  
format number totalValue as "$0,0.00"  
request totalValue           --Displays "$2,000,000.00"
```

For dates and times you must include the *from* clause with the existing format in the *format* command so that ToolBook II can correctly interpret the date and time to be reformatted:

```
today = "31 December 97"  
format date today as "mm/dd/yy" from "d M yy"  
request today           --Displays "12/31/97"
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 **Managing data**
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

The values of *sysDate* and *sysTime*, and the values of any variables declared with the *date* or *time* data types, are formatted automatically using the formats in *sysDateFormat* and *sysTimeFormat*:

```
sysTimeFormat = "h24:min"  
request sysTime           --Displays "18:00" (no seconds)  
local date today  
sysDateFormat = "m/d/y"  
today = "9/30/97"  
sysDateFormat = "M d, y"  
request today             --Displays "September 30, 1997"
```

Formatting data for calculating

Convert numbers, dates, and times to a suitable format before attempting to use them in calculations.

Formatting numbers for calculations

Format numbers as *null* before calculating. This converts the *sysDecimal* character to a period and removes:

- ◆ leading and trailing spaces.
- ◆ percent signs (%).
- ◆ the *sysCurrency* character.
- ◆ the *sysThousand* character.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 **Managing data**
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Formatting a number as *null* does not strip any other characters, so avoid using other characters for display. Commas are not valid in arithmetic calculations, but ToolBook II can delete a comma from a number only if the comma is the value of *sysDecimal*, *sysThousand*, or *sysCurrency*. The following example illustrates how *format* is used before calculating:

```
vTotal = "$1,500.00"  
format number vTotal as null           --Strips punctuation  
vTax = vTotal * .15  
format number vTax as "$0,0.00"  
request vTax                           --Displays "$225.00"
```

Formatting dates and times for calculations

For arithmetic involving dates or times, use the *seconds* format. This formats a date as the absolute number of seconds from January 1, 1970, and a time as the number of seconds since midnight. Each day has 86,400 seconds, so you can add and subtract days as a multiple of this value:

```
--This example shows you how to add 30 days to today's date  
local date dueDate  
local date today  
sysDateFormat = "seconds"           --Formats dates as seconds  
today = sysDate                     --Puts today's date into the  
                                     --variable today  
dueDate = today + (30 * 86400)      --Adds 30 days to the date  
sysDateFormat = "m/d/y"           --Formats the date for output  
request dueDate
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 **Managing data**
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

For some date arithmetic, it is easier to use the string format of a date instead of formatting it as *seconds*, as long as you can exactly predict the format of the string. For example, using string operators with a date is often easier when you want to add a month to a date rather than exactly 30 days. In the example below, the date is formatted using a comma as a delimiter, making it easy to extract the month and then increment it.

```
sysDateFormat = "m,d,y"      --Allows date to be a list
today = sysDate              --Declares today as a date
thisMonth = item 1 of today   --Gets only the month value
thisYear = item 3 of today    --Gets the year value
increment thisMonth          --Increments the month
if thisMonth > 12 then        --Adjusts if the month was December
    thisMonth = 1
    increment thisYear
end
dueDate = thisMonth & "," & item 2 of today & "," & thisYear
--Changes format back to slashes
format date dueDate as "m/d/y" from "m,d,y"
request dueDate
```

Formatting data for comparisons

Before you compare the values of dates, times, and numbers, you must make sure that the values to be compared are in the same format. Use the *as* clause in an *if* statement to establish the correct format for comparisons. For example:

```
--Works if the string in record field "dueDate" matches
--the format in sysDateFormat
dueDate = text of recordField "dueDate"
if sysDate > dueDate as date then
    request "Overdue!"
end
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data**
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

If you are comparing two values that ToolBook II considers dates (*sysDate* and variables that are typed as *date*), ToolBook II formats the dates using *sysDateFormat* and then does a character-by-character comparison between the two. As a result, you can compare any two dates, even if they were entered in different formats. For example:

```
local date vDate          --Declares as a "date" data type
local date vDate2
sysDateFormat = "m/d/y"   --Entered using one format
vDate = "4/18/97"
--Changes the format of all "date" data types
sysDateFormat = "M d, y"
vDate2 = "April 18, 1997" --Entered using a different format
if vDate = vDate2 then
    --Because both are dates the comparison returns true
    request "same!"
end
```

Validating data

To prevent a user from entering inaccurate or incomplete information, you can write handlers that test for such errors and then help the user correct them. The general steps for data validation are:

- 1 Get the value that the user entered.
- 2 Test that the value is valid.
- 3 Respond to the user. For example, report errors.
- 4 If the data was invalid, reset the focus to the field where the user entered data, erase the invalid value, or otherwise make sure that the invalid entry is taken care of.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 **Managing data**
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Generally, the best time to check data is immediately after it is entered. Therefore, you usually validate data in *leaveField* or *leaveRecordField* handlers.

Checking what the user entered

ToolBook II provides two general validation functions for checking values, *isType* and *isObject*. For example, you can check whether a user has entered an integer with statements such as these:

```
ask "Enter a number:"
if isType(int, It) is false
    request "Please enter an integer value."
end
```

For more specific checks, you must check for the exact values or types that you require. The following list provides some possibilities:

- ◆ To check for dates:

```
if isType(date, text of self) is false then
    ... --Further statements here to specify action
end
```

Note For details about formatting dates, see “Formatting data” earlier in this chapter.

- ◆ To check for one of several possible values:

```
get my text
if It is not in "Monday, Tuesday, Thursday"
    ... --Further statements here to specify action
end
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 **Managing data**
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

- ◆ To check for a specific number of characters or items:

```
get charCount(text of recordField "Amount" of page ID 3)
if It is not > 2 then
    ...    --Further statements here to specify action
end

get itemCount(newPosition)
if It is not 2 then
    ...    --Further statements here to specify action
end
```

- ◆ To check whether a character, word, or textline is valid:

```
if textline 3 of recordField "City" is "Seattle"
    ...    --Further statements here to specify action
end
```

Using *sysError* for validation

If you cannot check an exact value or data type, you can validate a user entry by attempting to use it in a predictable way, and then checking whether you generated an error. For example, if your application prompts the user to navigate to a particular page, your handler can attempt to go to the page that the user specifies and check whether ToolBook II generates an error. If there is an error, your handler can indicate that the specified page is invalid.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 **Managing data**
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

To hide errors from the user, set *sysSuspend* to *false* before checking the data (but be sure to set it back to *true* as soon as possible). For example:

```
to handle buttonClick
    targetPage = text of field "where"
    clear sysError      --Initialize to null
    sysSuspend = false
    go to page targetPage
    sysSuspend = true
    if sysError is not null then
        request "No such page number."
    end if
end buttonClick
```

Responding to the user

To inform the user of an error or to prompt for a correct response, use the *request* or *ask* command, as shown in the following example:

```
to handle leaveField
    get text of target
    if It is not null and isType(int,it) is false
        beep 1
        request "Please enter an integer value." with OK
    end if
    forward
end leaveField
```

When you display a dialog box with the *ask* or *request* command, it receives the focus. When the user closes the dialog box, the ToolBook II default response is to set the focus to the last object that had it before the dialog box was displayed—usually the field where the error occurred.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data**
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

If you put data validation statements in *leaveField* or *leaveRecordField* handlers, it is a good idea to allow users a graceful way to leave the field without supplying correct data. To allow users to continue without fixing errors, set *sysSuspendMessages* to *true* so that you do not trigger the *leaveField* handler again, set the focus manually to another object or to *null*, and then set *sysSuspendMessages* back to *false*. For example:

```
to handle leaveField
  get text of target
  if isType(int,It) is false
    beep 1
    request "Please enter an integer value." with "OK" or "Cancel"
    if It is Cancel
      sysSuspendMessages = true
      focus = null
      sysSuspendMessages = false
      --Break to the end of this handler
      break
    end if
    --Break out of all handlers on the stack and hand
    --control back to the ToolBook II system code
    break to system
  end if
  forward
end leaveField
```

You can also restore the field to the value it had before the user changed it. To do this, put the value of the field in a system variable or user property in an *enterField* handler. After validating the data, set the field to this value if the user chooses not to fix an entry error.



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data**
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Reading and writing to files

OpenScript provides a number of commands that allow you to read and write directly to operating system files. You use these commands for writing custom import and export programs, or writing data out to disk to save memory. The OpenScript commands for reading and writing to files are:

<i>createFile</i>	<i>seekFile</i>
<i>openFile</i>	<i>writeFile</i>
<i>readFile</i>	<i>closeFile</i>

Before you read or write to a file, you must open it. When you are finished with the file, you must close it to save pending data to disk and free memory and other system resources. For example:

```
to handle buttonClick
  vFileName = "c:\autoexec.bat"
  openFile vFileName
  readFile vFileName to EOF
  text of field "DOS File" = It
  closeFile vFileName
end
```

You can open up to 10 files in each instance of ToolBook II. However, if you are using files in other applications or in other ToolBook II instances, you might reach the limit established with the MS-DOS FILES command in the Config.sys file before you reach the ToolBook II limit of 10 per instance. For details about the FILES command, refer to your MS-DOS documentation.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data**
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Creating a new file

To create a new file, use the *createFile* command:

```
createFile "myfile.txt"
```

The *createFile* command both creates and opens the file, so you do not need to run a separate *openFile* command. If the file name you use matches an existing file, ToolBook II overwrites the existing file.

Tip To avoid overwriting an existing file, verify that the file name is not in use by opening the file. If you get an error, the file does not exist, so you are free to use that name.

Reading from a file

Using the *readFile* command, you can read a fixed number of characters from a file, or you can read through the file to a particular character, known as a delimiter. The syntax is:

```
readFile <file name> for <count>  
readFile <file name> to <delimiter>
```

where *count* is the number of characters to read and *delimiter* is a character to which you want to read.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data**
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Delimiters can include any single character, including the constants *CR*, *EOF*, *LF*, or *quote*. The *readFile* command returns the data it has read in the variable *It*. This example shows how to read through a file in 40-character increments:

```
to handle buttonClick
  clear sysError
  fileName = "c:\temp\fixed.txt"
  openFile fileName
  --Checks if the file exists and was opened successfully
  if sysError is not null then
    request sysError
    break
  end if
  do
    readFile fileName for 40      --Reads the next 40 characters
    newRecord = It
    request newRecord
    until newRecord is null
    closeFile fileName
  end buttonClick
```

When you read from a file, ToolBook II maintains a pointer to your current location in the file. If you scan forward to a delimiter, ToolBook II positions the pointer one character beyond the delimiter so that the delimiter is not included in the next string of data that is read. By default, the next *readFile* command will start from where the previous one left off. To reposition the file pointer, use the *seekFile* command. For details on the *seekFile* command, see “Moving to a set position within the file” later in this chapter.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data**
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Writing to a file

After you have opened a file, you can update it with the *writeFile* command. By default, the *writeFile* command appends data to the end of the file. For example:

```
--This handler updates a log file every time the book is opened
to handle enterBook
  clear sysError
  fileName = "books.log"
  openFile fileName
  --Checks that the file name exists and can be opened
  if sysError is not null then
    --If the file doesn't exist, create it (which also opens it)
    createFile fileName
  end if
  logEntry = name of this book & ", " & sysDate & ", " & sysTime & CRLF
  writeFile logEntry to fileName
  closeFile fileName
  forward
end enterBook
```

If you want to write to a point before the end of the file, use *seekFile* to set the pointer to a particular position within the file. For details, see the next section, "Moving to a set position within the file."



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data**
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Moving to a set position within the file

Use the *seekFile* command to set the current position in the file you want to read or write to. For example, the following handler reads 40-character records from a file using *seekFile* to set the pointer to the appropriate position:

```
--This script reads any record from a fixed-length ASCII file
to handle buttonClick
  clear sysError
  fileName = "c:\temp\fixed.txt"
  openFile fileName
  --Checks that the file exists and was opened successfully
  if sysError is not null then
    request sysError
    break
  end if
  recordLength = 40
  done = false --Used as flag to quit the loop
  --Loops until user enters null as the record number
  do
    --Prompts the user to identify the record to read
    ask "What record number?"
    if It is null then
      done = true --Set quit flag to true
    else
      if isType(int, It) then --Check for numeric input
        offSet = (It-1) * recordLength
        seekFile fileName for offSet from beginning
        readFile fileName for recordLength
        request It
      end if
    end if
  until done is true
  closeFile fileName
end buttonClick
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 **Managing data**
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

The *seekFile* command uses the variable *It* to return the position to which it has moved. To determine where the pointer is located, use statements such as these:

```
seekFile fileName for 0 from current    --Does not move, but sets It  
request It
```

To force the pointer to the end of the file (for example, to make sure that data is written to the end of the file), use this statement:

```
seekFile fileName for 0 from end
```

Searching pages

You can use the OpenScript *search* command to find text in a book, starting with the field or record field that currently has the focus. For example, the following statements search every visible field and record field on the current page:

```
--To search without displaying the ToolBook II Search dialog box  
to handle search  
  clear sysError  
  focus = null  
  ask "Enter the text you want to search for:"  
  searchText = It          --Set local variable to search string  
                          --specified by user  
  search page for searchText  
  if sysError is not null  
    request "Search text not found."  
  end if  
end search
```

Note ToolBook II always skips list boxes and combo boxes when searching.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data**
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

If a button has the focus, the search starts with the first field that follows the button in the layer order. To ensure that the search starts with the first field or record field on the current background, set *focus* to *null* before searching.

To search every field and record field of every page and background, use the statement *search for searchText* without specifying *page*, as in the example above. To search for the next occurrence of the search string, use *search again*. For faster searches, use *excluding background*, or specify the record fields you want to search.

```
search excluding background for searchText  
search in recordField "Name", recordField "Middle Name" for searchText
```

When ToolBook II finds matching text, it navigates to the page that contains that text. If the field can be edited, the text is selected.

```
search for searchText  
if sysError is null  
    --System property contains the selected text  
    get selectedText  
    ...    --Further statements here to manipulate the found text  
end
```

To search hidden fields, use *locateOnly* with the *search* command. For details, see the next section, "Searching without changing pages."



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data**
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Searching without changing pages

By default, when ToolBook II finds text during a search, it navigates to the page where the text is and highlights the text. If you do not want to navigate to another page or select the found text, use *locateOnly* with the *search* command. This causes ToolBook II to set the variable *It* to a value that shows where the text was found, rather than navigate to the page. The value in *It* is a comma-separated list containing the field ID, and the character positions of the beginning and ending letters that match the search text. For example:

```
search excluding background for searchText locateOnly
if It is null
    request quote & searchText & quote && "could not be found."
else
    --Show where the text was found
    request "Found" && quote & searchText & \
        quote && "in" && item 1 of It
end
```



For details about the *search* command, refer to the OpenScript reference in online Help.



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 **Managing data**
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Sorting pages

You can sort pages using the Sort Pages command on the Tools menu or the OpenScript *sort* command. The Sort Pages command is limited to sorting by record fields, but the *sort* command can sort pages by any expression that can be evaluated in the context of a page—for example, page name or page ID. You can also specify a range of pages with an expression—for example, *page 5 to pageCount of this book*:

```
to handle buttonClick
    sort pages 1 to pageCount of this book by text name of this page
end
```

You can specify multiple sort criteria, separated by commas. ToolBook II sorts pages by the first criterion, the second criterion, and so on until all criteria are met. The following example sorts a range of pages by two criteria:

```
to handle buttonClick
    sysCursor = 4    --Hourglass
    sort pages 2 to pageCount of this book by \
        ascending text text of recordField "priority", \
        ascending date text of recordField "Date Due"
    sysCursor = 1
end
```



How to use this online guide

Contents

- 1** Finding the information you need
- 2** Understanding OpenScript programming
- 3** Learning OpenScript basics
- 4** Using the scripting tools
- 5** Handling messages, events, objects, and properties
- 6** Creating an OpenScript statement
- 7** Using variables, values, and text
- 8** Writing common scripts
- 9** Handling user events
- 10** **Managing data**
- 11** Using special effects
- 12** Using dynamic data exchange
- 13** Using dynamic-link libraries
- 14** Debugging OpenScript programs

Index



Contents

- 1** Finding the information you need
- 2** Understanding OpenScript programming
- 3** Learning OpenScript basics
- 4** Using the scripting tools
- 5** Handling messages, events, objects, and properties
- 6** Creating an OpenScript statement
- 7** Using variables, values, and text
- 8** Writing common scripts
- 9** Handling user events
- 10** Managing data
- 11** Using special effects
- 12** Using dynamic data exchange
- 13** Using dynamic-link libraries
- 14** Debugging OpenScript programs

Chapter 11

Using special effects

You can use OpenScript to include special effects such as animation, sound, and drag-and-drop behavior in your application. This chapter provides information on each of these techniques.

CONTENTS

Animating objects	234
Adding drag-and-drop behavior	249
Creating sound effects	267



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Animating objects

To animate objects, you change their position, size, or shape, or you display a changing set of pictures that give the illusion of motion. Use these techniques to animate objects:

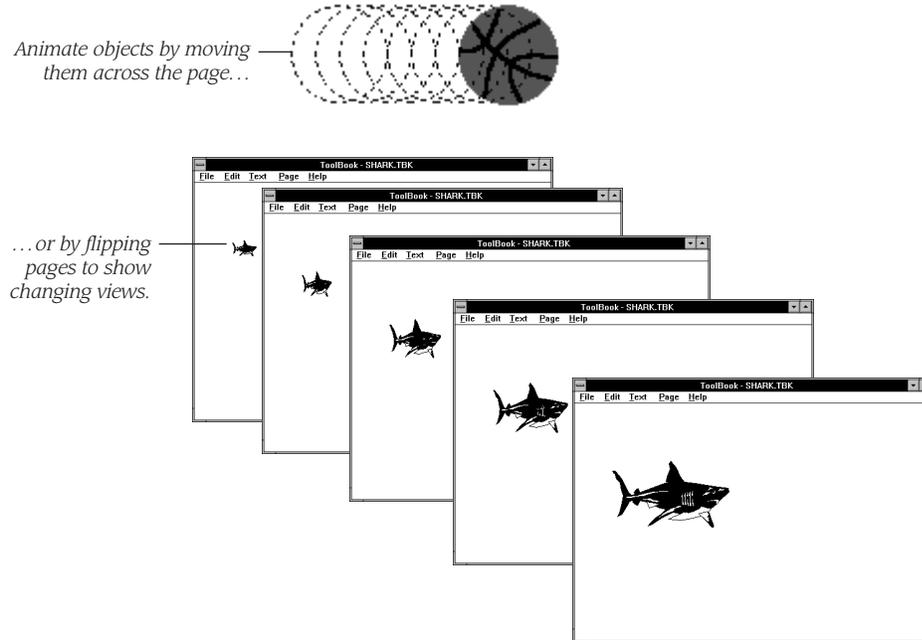
- ◆ Move an object using the *move* command.
- ◆ Resize or change the shape of an object by changing its *size* property.
- ◆ Simultaneously move and resize an object by changing its *bounds* property.
- ◆ Change the shape and position of a draw object (such as a line or polygon) by changing its *vertices* property.
- ◆ Show and hide a series of objects in a stack, each of which is a slight variation on the same image.
- ◆ Flip pages, each of which contains a slightly different view of the same image.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Figure 1 Two techniques for animating objects in ToolBook II



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Creating animation by moving objects

Use the *move* command to change the location of an object. You can move the object to specific locations on the page, or you can move it by a certain number of page units relative to its current location. If the location you indicate is outside the current window, the object will move off the screen and not be visible, but will still be available and respond to messages.

For example, the following handler for a draw object bounces the object off the bottom edge of the window. The exact value by which to adjust the bottom edge of the window depends on the shape of the object you want to bounce.

```
to handle buttonClick
  send sizeToPage                --Sets window size
  bottomEdge = (item 2 of size of this book) -700
  originalPos = item 2 of my bounds --Where object is now
  changeInPos = 100              --How much to move each time
  do
    move self by 0, changeInPos
    newBounds = my bounds
    if item 2 of newBounds > bottomEdge then
      changeInPos = -100        --Change object direction
    end if
  until item 2 of newBounds <= originalPos
end buttonClick
```



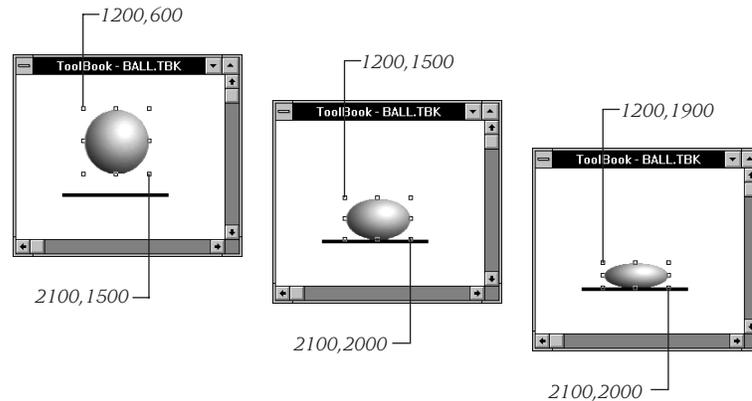
Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Changing the shape of an object

You reshape or resize an object by setting its *bounds* property, which changes the shape of its bounding box.

Figure 2 Changing the shape of an object by setting its *bounds* property



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

For example, the following script changes the bounds of an object so that the object appears squashed before bouncing up again:

```
to handle buttonClick
  vMyBounds = my bounds
  --Squash down the top
  step ctr from (item 2 of vMyBounds) to \
    (item 4 of vMyBounds) by 100
    item 2 of my bounds = ctr
  end step
  --Bounce back up
  step ctr from (item 4 of vMyBounds) to \
    (item 2 of vMyBounds) by -100
    item 2 of my bounds = ctr
  end step
end buttonClick
```

You can make an object grow or shrink by moving the location of the upper-left or lower-right corners of its bounding box. The following handler causes an object to grow equally in all directions by changing the coordinates of its bounding box by an equal amount in each direction. The handler makes the changes to a local variable and sets the *bounds* properties to the value of the variable so that all four coordinates are updated simultaneously:

```
to handle buttonClick
  step ctr from 1 to 15
    vMyBounds = my bounds
    decrement item 1 of vMyBounds by 100
    decrement item 2 of vMyBounds by 100
    increment item 3 of vMyBounds by 100
    increment item 4 of vMyBounds by 100
    my bounds = vMyBounds
  end step
end buttonClick
```



Contents

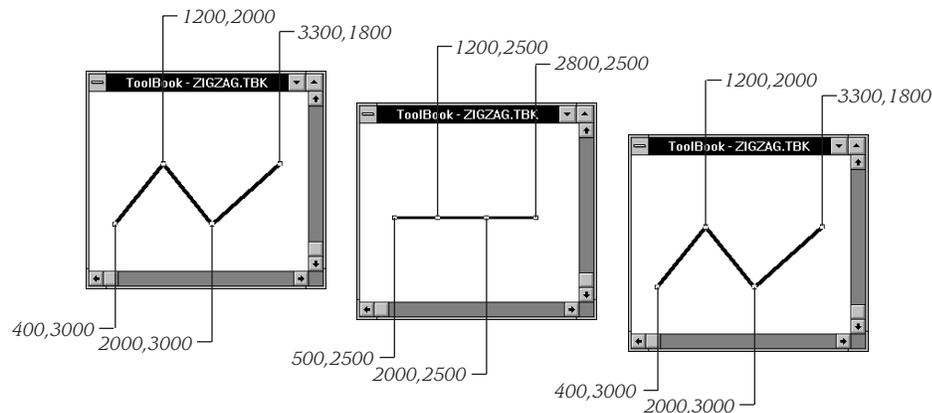
- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Animating lines and curves by changing vertices

You can animate angled and curved lines by changing their vertices. This technique is useful to create graphs or move waves along a line. For example, the following handler changes a line into a zigzag, a straight line, and then a zigzag again:

```
to handle buttonClick
    zigZag = "400,3000,1200,2000,2000,3000,3300,1800"
    straight = "500,2500,1200,2500,2000,2500,2800,2500"
    move angledLine "ZigZag" to 500,2500
    vertices of angledLine "ZigZag" = zigZag
    pause 50
    vertices of angledLine "ZigZag" = straight
    pause 50
    vertices of angledLine "ZigZag" = zigZag
end
```

Figure 3 Changing lines or curves by setting their *vertices* properties



Contents

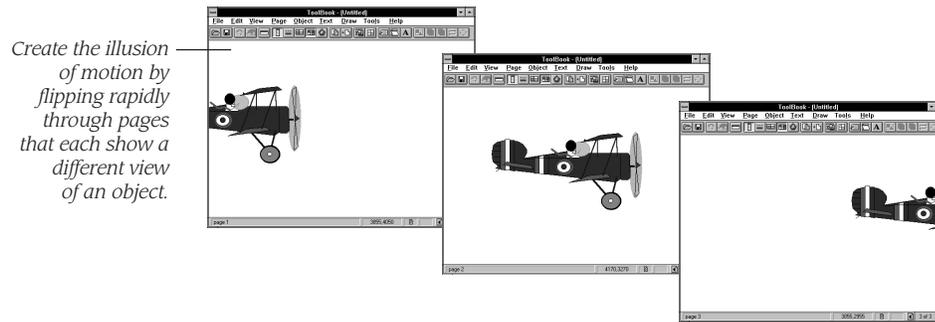
- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Tip To manipulate angled or curved lines, you must always provide the correct number of vertices. If necessary, use the function *itemCount* to determine how many points an object has.

Creating animation by flipping pages

To create animation that involves complex graphics such as bitmaps, you can place different views of the graphic on different pages and then move through the pages rapidly using the *flip* command. This is similar to the cel animation used to make cartoons.

Figure 4 Creating the illusion of motion by flipping pages



For example, this handler uses the technique illustrated above:

```
to handle buttonClick
    flip 3 pages
end
```

Tip You can use a viewer to flip pages for an animation sequence while displaying another page in the main window. This can be useful in a tutorial or demonstration application.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

To make the animation sequence run smoother, preload the pages into memory to eliminate delay while ToolBook II finds and loads the pages. To preload pages, set *lockScreen* (for a single viewer) or *sysLockScreen* (for all viewers) to *true* to prevent anything from showing on the screen, flip through the pages, and then set *lockScreen* or *sysLockScreen* back to *false*.

The following handlers illustrate this technique. The *enterBook* handler defines a system variable called *svFirstTime* that is set to *true* the first time the book is opened. When ToolBook II enters the background of the page that contains the animation, an *enterBackground* handler checks to see if this is the first time the book has been opened. If it is, the handler flips the pages.

```
to handle enterBook
  --Creates flag to check in enterBackground handler
  system svFirstTime
  svFirstTime = true
  forward
end

to handle enterBackground
  system svFirstTime
  if svFirstTime is true
    sysLockScreen = true
    flip 3 pages
    sysLockScreen = false
  end if
  forward
end enterBackground
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Optimizing animations that flip pages

To increase the speed of page-flipping animation:

- ◆ Build the sequence of images on the same background. Changing backgrounds takes extra time.
- ◆ Keep page size as small as possible so that ToolBook II has less to display and can display pages faster.
- ◆ Keep the number of objects on the pages to a minimum.
- ◆ Set the system variable *sysHistoryRecord* to *false* so that ToolBook II does not keep a record of each page visited.
- ◆ Set the system variable *sysSuspendMessages* to *true*, which avoids the overhead of *enterPage*, *enterField*, or *enterButton* messages.

If the screen seems to flicker when animated objects move, the *drawDirect* property of each object is set to *true*. To help eliminate flicker, set *drawDirect* to *false* for animated objects.

Note If you are working with 256-color display devices, bitmaps can flicker because of palette shift. For more information about palette shift, refer to the user guide that came with your ToolBook II product.



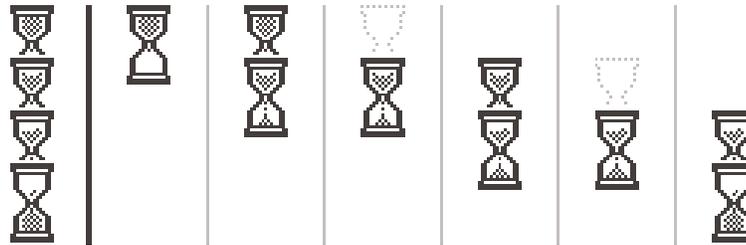
- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Creating animations by showing and hiding objects

You can animate an object by creating a series of objects that are each a variation on that object, and then showing and hiding the variations one by one.

Figure 5 Animating an object by showing and hiding a series of objects

To animate sand flowing through an hourglass...



...stack objects on top of each other,

hide all but the first object,

show the next object,

hide the previous object,

and then show and hide each pair in turn.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

For example, the following handler shows a succession of six objects by first showing one and then hiding the previous one.

```
to handle buttonClick
  step ctr from 1 to 6
    if ctr < 6 then
      show ellipse (ctr + 1)  --Shows next view
    end if
    hide ellipse ctr        --Hides the current one
    pause 20
  end step
  show ellipse 1            --Restores first view
end buttonClick
```

Creating handlers for animation

To start an animation, you must trigger it with an event. Events that you can use to trigger animation sequences include:

- ◆ **Button-click events** (*buttonClick*, *buttonUp*) Use a button-click event to set off an animation sequence that should occur under the control of the user.
- ◆ **Enter events** (*enterPage*, *enterBook*) Use an enter event to trigger an animation sequence that starts automatically when the user opens a book or navigates to a page.
- ◆ **Idle event** (*idle*) Use an idle event to create animation that runs continuously, but that can be interrupted when the user interacts with other objects on the page.
- ◆ **Timer events** (*timerStart*, *timerNotify*) Use a timer event to trigger an animation sequence that starts automatically after a specified period of time.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

When you design a handler to trigger an animation sequence, consider how the animation will run in the application. If the animation sequence is completely contained in a handler, the user will not be able to interrupt the sequence unless the handler also checks for an ESC key or a button click.

If you use an *enterPage* or *enterBackground* handler to trigger an animation sequence, the sequence will run each time the user navigates to the page. This might annoy users as they move from page to page, especially if the animation is lengthy. To prevent an animation sequence from running each time a user enters a page or background, build in conditions. For example, create a condition that runs the animation only the first time the user enters the page or background.

Linking animation to an *idle* handler

You can use an *idle* handler to run an animation sequence whenever a user is not working with an object on the current page. For example, you can have an animated corporate logo that changes as long as the user is not clicking the mouse or typing characters.

If you run animation sequences from an *idle* handler, be sure not to run the complete animation each time the handler is called, or no time will be left for other events to occur. One way to design animation for an *idle* handler is to put each step of the animation process in a single execution of the *idle* handler, and to keep the execution time of each step as short as possible. That way, ToolBook II can check between each *idle* handler execution to see if there are any pending messages, such as button clicks or menu selections.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

The handler for an animation sequence can store its current state in a user property or system variable, so the next time ToolBook II sends an *idle* message, the handler can continue from that point in the animation sequence. For example:

```
--Hides and shows an object to animate it; each time the handler is
--called, it hides and shows one view of the object
to handle idle
  system ctr
  if ctr = null then
    ctr = 1
  end if
  if ctr < 6
    show ellipse (ctr + 1)
    hide ellipse ctr
    increment ctr
  else
    ctr = 1
    show ellipse "1"
    hide ellipse "6"
  end if
  pause 10      --Delay animation for 1/10 second
  forward
end idle
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Initializing and cleaning up animation sequences

When you run an animated sequence, you should initialize and clean up the sequence properly. For example:

- ◆ Place an object at a known location before starting an animation sequence that moves it.
- ◆ Show the proper starting view of an object that has multiple views, hiding the other views.
- ◆ Set the bounds of an object to their proper size before changing them to show the object growing or shrinking.

You can handle initialization and cleanup tasks in the handler that runs the animation sequence. However, if you want an entire page to be displayed correctly when the user navigates to it, place the initialization in an *enterPage* or *enterBackground* handler. You can also use an *enterPage* handler to perform any cleanup tasks on the page you are leaving, or you can write a *leavePage* handler that restores the page to its normal state before the user moves to the next page.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Controlling animation speed

If the animation effect you are creating runs too quickly, you can slow it down using a *pause* statement between portions of the animation sequence. For example:

```
to handle buttonClick
    step ctr from 1 to pageCount of this book
        flip
        pause 5 ticks
    end step
end buttonClick
```

Note that if the delay between the stages in an animation sequence is too great, it will appear jerky.

Creating smoother animations

A number of factors control how smoothly an object moves, changes shape, or redraws. Use these guidelines:

- ◆ Large jumps in location make the motion of an object appear jerky.
- ◆ Setting the *drawDirect* property of an object to *false* reduces the flicker caused by redrawing directly on the screen.
- ◆ The simpler the object, the smoother the motion. For example, a single rectangle moves more smoothly than a group of objects.
- ◆ Transparent objects and objects with rounded corners display more slowly than other objects.
- ◆ The viewer you use to display an animation sequence should use at least one image buffer. If the animation sequence also changes objects on the background, it should use two image buffers.



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects**
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Adding drag-and-drop behavior

In ToolBook II, you can define a special effect called drag-and-drop for your applications. With drag-and-drop, you can click an object (called the source object), drag an image representing the source object to another location, and release the image on another object (called the destination object).

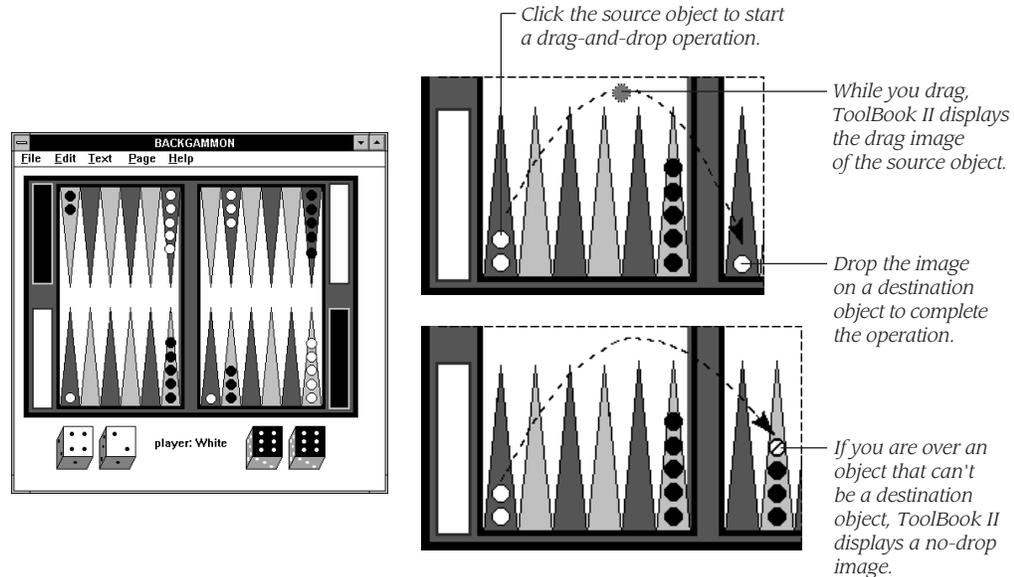
While you are dragging, a drag image (usually an icon) moves around the screen to show where the mouse pointer is. If you drag the image over an object that cannot be a destination object, a no-drop image (often a circle with a line through it) appears.

For example, in Figure 6 on the following page, the playing piece of a backgammon game is the source object and each triangle is a destination object. When a player clicks and drags a game piece, ToolBook II displays a drag image to indicate where the piece moves. When the player releases the mouse, the application moves the game piece to its new location. If the player tries to drag outside the game area, ToolBook II displays a no-drop image to indicate that the user is not allowed to drop a piece in that location.



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 **Using special effects**
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Figure 6 The drag-and-drop operation and a no-drop image



In ToolBook II, you can use drag-and-drop behavior to create a mouse-driven visual interface for many tasks and actions. Some uses might be to drag:

- ◆ items to a wastebasket to delete them.
- ◆ symbols onto a calendar to represent appointments, birthdays, or reminder alarms.
- ◆ game pieces around a board, as in a chess or card game.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

ToolBook II simplifies defining drag-and-drop behavior by including special properties and messages. When a user drags an object, ToolBook II automatically displays the correct image (drag or no-drop), sends notification that a drag operation has begun or ended, and identifies the source and destination objects.

During a drag-and-drop operation, ToolBook II does not actually change anything in your application—for example, it does not move an object as you drag it. You must write a handler to define what happens when the mouse button is released, such as deleting the source object, moving it, or changing its color.

You can drag any object that you can place on a page, including pictures or paint objects that you have imported. ToolBook II allows you to drag objects within a viewer or from one viewer to another. You cannot drag a page, background, or book, although pages can be destination objects.

Defining basic drag-and-drop behavior

To define drag-and-drop behavior, you must:

- ◆ identify the source object. When a user clicks the source object, ToolBook II initiates the drag-and-drop operation.
- ◆ identify at least one destination object so that there is a place to drop the source object. The destination object can be the page itself.
- ◆ write a handler for an *endDrag* message to define what happens when a user releases the drag image over a destination object.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

You can also initiate drag behavior using the OpenScript *drag* command. This gives you the flexibility to trigger a drag operation using an event other than clicking the source object, or to drag objects that cannot be clicked, such as groups. For details, see “Using the *drag* command” later in this chapter.

Identifying source and destination objects

You can identify the source object for a drag-and-drop operation by setting its *defaultAllowDrag* property to *true*. Right-click the object, choose Drag & Drop from the shortcut menu to display the Drag & Drop dialog box, and then select Allow Drag. When this property is *true*, ToolBook II initiates drag-and-drop behavior whenever the user clicks the object.

To identify a destination object, set its *defaultAllowDrop* property to *true* by checking Allow Drop in the Drag & Drop dialog box of the object. When this property is *true*, ToolBook II sends an *endDrag* message to the object when the user drops it. If the *defaultAllowDrop* property is *false*, ToolBook II displays the no-drop image when the user moves the drag image over the object, and sends no messages if the user releases the mouse button over the object.

You can also determine conditionally whether objects can be source or destination objects. For example, in a chess game, you can determine whether squares on the chessboard can be destination objects depending on what piece the player is moving. To create source and destination objects conditionally, write handlers for the *allowDrag* and *allowDrop* messages. For details, see “Specifying source and destination objects conditionally” later in this chapter.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Creating an *endDrag* handler

If the user releases the mouse button while the drag image is over a destination object, ToolBook II sends an *endDrag* message to the source object. Write a handler for this message to define the result of the drag-and-drop operation. For example, in a chess application, if the user releases the drag image over a square, an *endDrag* handler moves the game piece and defines related behavior, such as removing other game pieces from the board.

To illustrate drag-and-drop behavior, create an application that allows a user to delete objects at Reader level by dragging the objects to a wastebasket graphic. Create an object named “wastebasket” and set its *defaultAllowDrop* property to *true*. Then, create a number of additional objects and set their *defaultAllowDrag* properties to *true*. Put the following handler in the script of the page or book:

```
to handle endDrag destinationObject, loc, targetWindow
    if destinationObject is not null then
        if name of destinationObject is "wastebasket" then
            clear target      --Deletes the source object
        end if
    end if
end endDrag
```

ToolBook II passes the *uniqueName* of the destination object to the *endDrag* handler, along with the location of the drag image relative to the viewer in which the object is dropped. These two parameters allow you to identify which object receives the drop and where the drag image is when the mouse button is released. If the user releases the mouse button over an object that will not accept a drop, the *<destination object>* parameter is *null*. In an *endDrag* handler, *target* refers to the original source object.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

OpenScript elements for drag-and-drop operations

The following tables specify properties, commands, and event messages that relate to drag-and-drop behavior.

Properties

Property	Value
<i>defaultAllowDrag</i>	Specifies whether an object can be dragged
<i>dragImage</i>	Image (icon, cursor, or bitmap) displayed during a drag operation when the object under the cursor can accept a drop
<i>noDropImage</i>	Image (icon, cursor, or bitmap) displayed during a drag-and-drop operation when the object under the cursor will not accept a drop
<i>defaultAllowDrop</i>	Specifies whether an object that is the destination of a drop can accept the drop

Commands

Command	Description
<i>drag</i>	Starts drag-and-drop behavior from within a script as if the user had initiated it with the mouse. The drag can be confined within a specified rectangular area



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Event messages

Message	Description
<i>beginDrag</i>	Sent to the dragged object when a drag event begins
<i>endDrag</i>	Sent to the dragged object when the mouse button is released; signals the end of a drag event
<i>enterDrop</i>	Sent to an object if its <i>defaultAllowDrop</i> property is <i>true</i> or if an <i>allowDrop</i> handler returns <i>true</i> when the mouse cursor enters its bounds during drag mode
<i>leaveDrop</i>	Sent to a destination object when the mouse cursor leaves its bounds during drag mode
<i>objectDropped</i>	Sent to a destination object when a dragged object is released over its bounds
<i>stillOverDrop</i>	Sent to the destination object continuously during a drag event when it is positioned under the drag image; similar to <i>buttonStillDown</i>
<i>allowDrag</i>	Sent to the source object to determine whether that object can be dragged; if a <i>to get allowDrag</i> handler for this message is not found, ToolBook II checks the <i>defaultAllowDrag</i> property of the source object
<i>allowDrop</i>	Sent to the destination object to determine if the object will accept the drop; if a <i>to get allowDrop</i> handler is not found, ToolBook II checks the <i>defaultAllowDrop</i> property of the object



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Properties of objects

Object	Drag-and-drop properties
All draw objects and hotwords	<i>defaultAllowDrag, defaultAllowDrop, dragImage, noDropImage</i>
Group	<i>dragImage, noDropImage</i>
Page	<i>defaultAllowDrop</i>
Background	<i>none</i>
Book	<i>none</i>
Viewer	<i>none</i>

Specifying source and destination objects conditionally

Instead of relying on the *defaultAllowDrag* and *defaultAllowDrop* properties, you can add case-by-case control over whether an object can be dragged or will allow a drop. For example, in a game application you might prohibit a drop on a square if the player is trying to make an illegal move. To define a source object conditionally, write a *to get* handler for the *allowDrag* message. To define a destination object, write a *to get* handler for the *allowDrop* message.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

When the user clicks an object, ToolBook II sends an *allowDrag* message to the target object.

- ◆ If the *allowDrag* handler returns *true*, ToolBook II begins a drag-and-drop operation.
- ◆ If the *allowDrag* handler returns *false*, the object cannot be dragged and ToolBook II sends a *buttonClick* message.
- ◆ If there is no *allowDrag* handler in the hierarchy of the object, ToolBook II uses the value of the *defaultAllowDrag* property of the object to determine whether to begin a drag-and-drop operation.

The following example shows how to use a *to get* handler to determine whether an object can be dragged:

```
--This handler allows an object to be dragged if the user is an
--administration user, as determined elsewhere in the application;
--place this handler in the script of the source object
to get allowDrag
    system userLevel          --System variable set elsewhere
    if userLevel is "Admin"
        return true
    else
        return false
    end if
end allowDrag
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

You can also use a *to get* handler to determine whether an object will allow a drop. When the user moves the drag image over an object, ToolBook II sends an *allowDrop* message to the object. If there is a *to get* handler for the *allowDrop* message and it returns *true*, the object can be a destination object. If the *to get* handler returns *false*, the object cannot be a destination object and ToolBook II displays the no-drop image. If there is no *to get* handler for the *allowDrop* message in the hierarchy of the object, ToolBook II checks the value of the *defaultAllowDrop* property of the object. For example:

```
--This handler allows a drop if the source object is named reportList;  
--place this handler in the script of the destination object  
to get allowDrop sourceObject  
    if name of sourceObject contains "reportList" then  
        return true  
    else  
        return false  
    end if  
end allowDrop
```

Using the *drag* command

Use the *drag* command to begin a drag operation from within OpenScript. This command is the equivalent of clicking the source object—the drag image appears and the *beginDrag* message is sent to the source object. From that point the user can move the mouse to drag the source object.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

The drag operation ends when a user clicks the left mouse button—that is, when the next *buttonUp* message is sent. Use the *drag* command to initiate a drag operation for objects that cannot be clicked directly, such as groups, or to trigger the drag operation using an event other than *buttonDown*. For example:

```
--Handler that begins a drag operation when a user clicks
--the right mouse button
to handle rightButtonDown
    drag paintObject "alarmClock"
end
```

You can constrain the area in which the drag operation takes place, but only if you use the *drag* command. To constrain a drag-and-drop operation, add the keyword *within* and a list of four integers in page units that define the area. For example:

```
--Stays in upper-left corner
drag rectangle "colorBar" within "1000,1000,4000,4000"
```

Forcing an object to accept a drop (silent dragging)

Silent dragging works independently of the *allowDrop* and *defaultAllowDrop* values of the destination object. For example, you might have an application such as a clip art window from which the user can drag images. In this case, you want to allow the source object to be dropped on a destination object regardless of what the *defaultAllowDrop* property of the destination object is or what an *allowDrop* handler returns.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

During silent dragging, ToolBook II does not send *allowDrop*, *enterDrop*, or *leaveDrop* messages. ToolBook II still sends the *beginDrag* and *endDrag* messages to the source object, and always displays the drag image.

You can initiate silent dragging only by using the OpenScript *drag* command and adding the keyword *silently* to the *drag* command. For example:

```
drag picture "artClip" silently
```

For details, refer to the entries for *allowDrag*, *allowDrop*, *beginDrag*, *defaultAllowDrag*, *defaultAllowDrop*, and *enterDrop* in the OpenScript reference in online Help.

Working with mouse-event messages for drag-and-drop operations

During a drag-and-drop operation, ToolBook II sends drag-and-drop messages instead of the usual mouse or button messages. For example, when the user clicks a source object, ToolBook II sends a *beginDrag* message instead of a *buttonDown* message. The following table shows the sequence of messages sent for a drag-and-drop operation.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Order in which drag-and-drop messages are sent

Event	Message	Target
User clicks an object	<i>allowDrag</i>	Source object
User has clicked object with <i>allowDrag</i> returning <i>true</i> or <i>defaultAllowDrag</i> set to <i>true</i>	<i>beginDrag</i>	Source object
Mouse cursor enters bounds of a destination object	<i>allowDrop</i> , then <i>enterDrop</i>	Object under drag image
Mouse cursor is over a destination object	<i>stillOverDrop</i>	Object under drag image
Mouse cursor leaves destination object	<i>leaveDrop</i>	Object under drag image
Mouse button is released over destination object	<i>objectDropped</i> <i>endDrag</i>	Destination object Source object

The user can bypass drag-and-drop behavior for an object by double-clicking the object. This prevents ToolBook II from initiating a drag-and-drop operation. Instead, ToolBook II sends a *buttonDoubleClick* message, followed by the normal *buttonStillDown* messages if the user holds down the mouse button, followed by single *buttonClick* and *buttonUp* messages.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Because ToolBook II still sends *buttonStillDown*, *buttonClick*, and *buttonUp* messages when the user double-clicks the object, you do not have to change any scripts that handle these messages. A handler for *buttonClick* or *buttonUp* still runs, but only if the user double-clicks the object. The only message that is not sent for a draggable object is the *buttonDown* message.

Testing the start of a drag operation

If you need to initialize variables or properties, or perform other operations when a drag-and-drop operation begins, write a handler for the *beginDrag* message. For example, you can store the current state of the position, color, and so on of the object in system variables when the drag operation begins so you can compare them to changed values later when the *endDrag* message is sent.

Changing drag and no-drop images

You can change the images that ToolBook II displays when you drag an object or cross over an object that will not accept a drop. To use a custom drag or no-drop image, assign an imported cursor, icon, or bitmap resource to the *dragImage* and *noDropImage* properties using the Drag & Drop dialog box of the object. Or, use OpenScript statements such as these:

```
dragImage of field "list" = icon ID 101  
noDropImage of field "list" = icon ID 102
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

If you set the *dragImage* or *noDropImage* property to *null* or choose None for these settings in the Drag & Drop dialog box of the object, ToolBook II uses the default drag and no-drop images.

Figure 7 Default drag and no-drop images



Dragging between Reader and Author levels

The behavior of objects dragged between viewers at Author and Reader levels is the same. For example, you can drag clip art from one viewer at Reader level into another at Author level.

Dragging fields

As with other objects, you can define drag-and-drop behavior for fields, list boxes, and combo boxes by setting the *allowDrag* property to *true* or by writing an *allowDrag* handler. However, if you do so, you change the ToolBook II default response to the left mouse button click, which is to highlight a line in a list box, select an item in the drop-down list of a combo box, or move the insertion point in an editable field.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

To retain the ToolBook II default behavior for a left mouse button click, you can set *allowDrag* to *false* or return *false* from an *allowDrag* handler and trigger the drag behavior using a different event. In a list box, you can set *allowDrag* to *false* and use a handler for the *buttonDown* message to execute the *drag* command. When you click a line in the list box, the highlight moves and a drag event begins. For example:

```
to handle buttonDown
    drag target
end
```

You can also use a *to get allowDrag* handler to determine the state of a field and start a drag operation accordingly. For example, an *allowDrag* handler for an editable field can return *true* if there is selected text; if no text is selected, the *allowDrag* handler returns *false*, which retains the ToolBook II default editing behavior in the field.

For details about using the drag command, see “Using the *drag* command” earlier in this chapter. For details on implementing drag-and-drop behavior in fields, refer to the scripts in the Library.tbk sample application.

Dragging groups

If a user clicks a draggable object in a group, ToolBook II starts a drag operation for that individual object, not for the group. You can assign a drag image to a group, but groups do not have *defaultAllowDrag* or *defaultAllowDrop* properties, nor does ToolBook II ever send them *allowDrag* or *allowDrop* messages.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

If you want to drag a group, use the OpenScript *drag* command. You can trigger the *drag* command using any event, such as the *buttonDown* event. For example, put this handler into the script of the group to initiate dragging when the user clicks anywhere in the group:

```
to handle buttonDown
    drag self
end
```

However, if the *defaultAllowDrag* property for the component of the group that receives the *buttonDown* message is *true*, the drag-and-drop behavior of that component takes priority for the group. If you want to create a group that you can drag but that also includes independently draggable objects, put a *to get allowDrag* handler in the group script. Be sure there is not already an *allowDrag* handler in the script of the component object that is clicked. This handler can then determine whether the drag operation applies to the component or to the group. The group itself will not be the target for an *allowDrag* message, but the *allowDrag* message will pass up the hierarchy to the group.

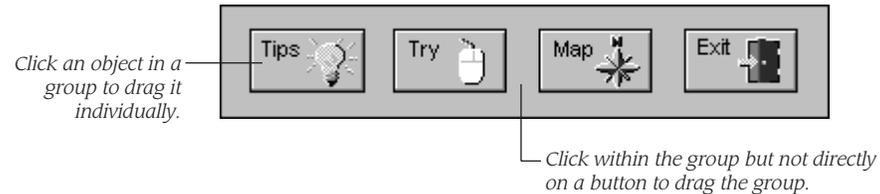


Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 **Using special effects**
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

For example, you can create a button palette by grouping a number of buttons on top of a rectangle while leaving some space between the buttons where you can click with the mouse. You can then click the buttons to get their drag behavior, or between the buttons (but still within the group) to initiate the drag behavior of the group.

Figure 8 Dragging individual objects or an entire group



Use a handler such as the following in the group script:

```
to get allowDrag
  if target contains "button" then
    return true
  else
    drag group "buttonPalette"    --Use the name you have assigned
    return true
  end if
end allowDrag
```



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Creating sound effects

You can create sound effects in ToolBook II using the *beep* command or by incorporating sound files. For example:

```
to handle buttonClick
  if pageNumber of this page < pageCount of this book then
    go next page
  else
    beep 1 --Specify number of times to beep
    request "Last page."
  end if
end buttonClick
```

Note In Windows 3.1, the *beep* command uses the sound assigned to the Default Beep event under Sound in the Control Panel. Windows plays sound files asynchronously, so the handler continues while the beep sound plays. If the sound file for the beep is long, beeping repeated times can cause the sound file to play continuously.

If your computer has a sound card and the appropriate media control interface (MCI) sound driver installed, you can play wave-audio (WAV) files. For example, you can add recorded sound to your books for extended sound effects or short voiceovers. To play wave-audio files, use the *playSound()* function:

```
to handle buttonClick
  get playSound("c:\windows\chimes.wav")
  request It
end
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

The *playSound()* function has an optional second parameter, *<wait>*, that determines whether ToolBook II waits for the sound to complete before continuing with the next command in the handler. If the value of *<wait>* is *false* (the default), ToolBook II returns immediately, and the sound plays asynchronously. For example, you can launch a voiceover that will play while the handler continues. If the value of *<wait>* is *true*, ToolBook II suspends the handler until the sound has finished.

The *playSound()* function returns *true* immediately if the wave-audio file is available; otherwise, it returns *false*. For example:

```
--Asks user which sound file to play and then plays it
ask "What sound do you want to play?"
--Returns immediately because the wait parameter is false
get playSound(It,false)
if not It
    request "File not found."
end
```

To stop the sound while it is playing, call *playSound()* and pass it *null*:

```
to handle buttonClick
    get playSound("toolbook\samples\dbeat1.wav",false)
    pause 2 seconds
    get playSound(null)          --Stops the sound file
end
```

Note The *playSound()* function works best with wave-audio files smaller than 100 KB.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange**
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Chapter 12

Using dynamic data exchange

With dynamic data exchange (DDE), you can add the capabilities of other Windows applications to ToolBook II. You can exchange information with another Windows application, or even control other applications from within ToolBook II. This chapter describes how to use DDE with ToolBook II.

CONTENTS

About dynamic data exchange	270
Using ToolBook II as a client	272
Using ToolBook II as a server	279



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

About dynamic data exchange

Dynamic data exchange (DDE) is a Windows communications protocol that you can use to exchange data between Windows applications or to execute commands in other Windows applications. DDE integrates Windows applications so that they can use each other's features.

A DDE conversation consists of two or more applications passing Windows messages back and forth. The client application initiates the conversation, and the server application responds to the requests of the client.

The client initiates a conversation with the server by addressing it by its server name. For example, the ToolBook II server name is *ToolBook*. The client can also specify a topic, which is either a general classification of data that the two applications will exchange, or a server file name. In general, a typical DDE conversation goes like this:

- 1 The client initiates a conversation.
- 2 The server responds.
- 3 The client and server exchange data or execute commands. For example:
 - ◆ The client requests data from the server and the server sends data.
 - ◆ The client sends data to the server without the server requesting it.
 - ◆ The client requests that the server execute a command.
- 4 The conversation is terminated by either the client or server.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 **Using dynamic data exchange**
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Both applications must support DDE to exchange information, and both must be open before the exchange can begin. In a DDE conversation, the same application can be both client and server. For example, one instance of ToolBook II can talk to another instance of ToolBook II.

If you are using DDE to return data, each exchange between a client and server sends and receives a single piece of information. For example, you can use DDE to remotely set the text of a field, but not to set the text of a whole collection of fields at once.

For details about DDE, refer to the Microsoft Windows SDK. For details about the server and topic names of an application, refer to the documentation for the application.

Using dynamic data exchange with ToolBook II

In ToolBook II, DDE is built into a set of OpenScript messages: *closeRemote*, *executeRemote*, *getRemote*, *keepRemote*, *remoteCommand*, *remoteGet*, *remoteSet*, and *setRemote*. To use DDE, you need a copy of the program you want to exchange data with, and you need to know what DDE syntax the program expects.



For details about OpenScript DDE commands and messages, refer to their entries in the OpenScript reference in online Help.

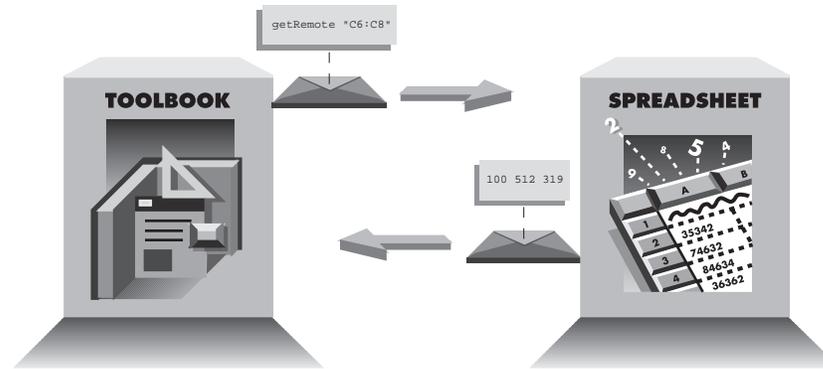
You can use the OpenScript DDE commands to exchange text data with other DDE-enabled applications or other ToolBook II applications. Figure 1 on the following page shows how OpenScript commands are automatically translated into DDE messages.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Figure 1 A DDE conversation with ToolBook II as a client



Note Some applications can establish permanent data links using WM_DDE_ADVISE and WM_DDE_UNADVISE messages. These DDE messages have no effect in ToolBook II because ToolBook II cannot establish permanent data links.

Using ToolBook II as a client

When ToolBook II is the DDE client, you use the *getRemote*, *setRemote*, and *executeRemote* commands to exchange data with and execute commands in the server application. For example, you can use Microsoft Excel for mathematical functions and data maintenance, and access that data from ToolBook II. You can also use DDE to make changes to the Excel spreadsheet or execute Excel commands from within ToolBook II.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

In general, to use ToolBook II as a DDE client:

- 1 Open the server application with the *run* command.
- 2 Exchange data with and execute commands in the server application using the *getRemote*, *setRemote*, and *executeRemote* commands.
- 3 Determine the success of each DDE request by checking the value of the *sysError* property.
- 4 If necessary, keep the DDE channel open with the *keepRemote* command, or explicitly close it with the *closeRemote* command.

The following sections go into more detail about each step.

Opening the server application

To open the server application, use the OpenScript *run* command, specifying the application and file name. Include the full path of the application if it is not in your DOS path. If you do not want to see the application on the screen when you run it, use the keyword *minimized* with the *run* command. In the same handler, call the *yieldApp()* function from *Tbwin.dll* to ensure that the application is initialized and ready to receive DDE commands. For example:

```
--Put this handler in the book script
to handle enterBook
    linkDLL "tbwin.dll"
        INT yieldApp( )
    end linkDLL
    run "excel.exe budget.xls" minimized
    get yieldApp( )
    ... --Statements here to handle errors if Excel is not running
    ... --Statements here to handle the DDE conversation
forward
end enterBook
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs



For details about `Tbwin.dll` and the `yieldApp()` function, refer to “DLL reference” in the OpenScript reference in online Help.

If the server application is already running, executing a `run` command will usually start a second instance of the application. To check whether the application is already running, send it a DDE command and examine the value of `sysError`. For details, see “Checking for errors” later in this chapter.

Exchanging data and executing commands in the server application

You use the `getRemote`, `setRemote`, and `executeRemote` commands to exchange data with and execute commands in a DDE server application.

To request an item from a server application, use `getRemote` and specify the item, server application, and topic. For example:

```
--Requests Excel data stored in a range named Regional_Totals
--Excel returns tab-delimited text in the Windows CF_TEXT format
to handle buttonClick
    getRemote "Regional_Totals" application "excel" topic "regions.xls"
    put It into text of field "summary"
end
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

To set an item in a server application, use *setRemote* and specify the value to give the item, the server application, and the server topic. For example:

```
--ToolBook II is DDE "middle man" between WinWord and Excel  
to handle buttonClick  
  getRemote "StockPrices" application "excel" topic "prtfolio.xls"  
  put CRLF & text of field "old prices" after It  
  --In the next statement, "prices" is a WinWord bookmark  
  setRemote "prices" to It application "winword" topic "annlrept.doc"  
end
```

To execute commands in the server application, use *executeRemote* and specify the commands, the server application, and the server topic. For example:

```
--Size and move the Excel window with Excel macro commands  
--The square brackets are required by Excel  
executeRemote "[app.restore( )][app.size(481,190)][app.move(1,172)]" \  
  application "excel" topic "system"
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Checking DDE commands before sending them

Because DDE commands often involve long strings that can be difficult to get right, you can build the DDE command out of shorter, more manageable pieces, preview it with the *request* command, and then run it with the *execute* command. For example:

```
--Builds a DDE command out of individual pieces
to handle buttonClick
  ddeCommand = "executeRemote"    --DDE command to be run
  ddeParameters = null           --Parameter string for the command
  put "[app.restore( )]" after ddeParameters
  put "[app.size(481,190)]" after ddeParameters
  put "[app.move(1,172)]" after ddeParameters
  ddeTopic = "application excel topic system"
  ddeExecute = ddeCommand && ddeParameters && ddeTopic
  request ddeExecute             --Previews the command
  execute ddeExecute             --Runs the command
  request sysError
end
```

Checking for errors

After executing a DDE command, immediately check the value of *sysError* to determine whether the command was successful. If the first item of *sysError* contains something other than OK, then the DDE command failed. Most failed commands are caused by incorrect syntax for the server application in the *<command>* parameter. The possible *sysError* values resulting from a DDE command are:

- ◆ *OK* The application responded to the request. (For *getRemote* and *setRemote*, the requested data is put into *It*.)
- ◆ *Failed: Busy* The application responded but was occupied and failed to perform the requested action.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

- ◆ *Failed: Denied* The application responded but could not or would not satisfy the request.
- ◆ *Failed: Memory Error* ToolBook II did not have enough global or local memory to generate the request or accept a response.
- ◆ *Failed: Interrupted* The application responded, but the connection was broken before the application acknowledged the command.
- ◆ *Failed: No Server* No application responded to the request.

You should include statements in your scripts to respond to errors based on the value of *sysError*. For example:

```
--This handler checks to see whether a ToolBook II application
--is running by testing its response to a DDE query
to get isRunning bookName          --Is the requested book running?
  getRemote "this book" application "toolbook" topic bookName
  if item 1 of sysError is "OK"     --Check sysError
    return true
  else
    return false                   --Failed to respond
  end
end
```

Keeping the DDE channel open

By default, a DDE channel remains open until ToolBook II is no longer executing a handler or command; when the handler is finished, ToolBook II automatically closes the DDE channel. Each time you run the *getRemote*, *setRemote*, and *executeRemote* commands, ToolBook II:

- ◆ initializes the server application.
- ◆ gets data, sets a data value, or executes a command in the server.
- ◆ terminates the DDE communication.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

To keep the DDE channel open after ToolBook II has executed all handlers or commands, use the *keepRemote* and *closeRemote* commands. If you use the *keepRemote* command, you eliminate the need to initialize the server application and terminate the DDE conversation, making the scripts more efficient. The *keepRemote* command can go anywhere in a handler that runs DDE commands. When you finish with the server application, use the *closeRemote* command to end the DDE communication.

For example, the following script keeps a DDE conversation open while the user remains on the page and closes it when the user navigates to another page. The content of the status field is updated whenever a script executes *send updateStatus*.

```
--Put these handlers in the page script
to handle enterPage
    keepRemote application "ToolBook" topic "myapp.tbk"
    send updateStatus --Initializes status field
    forward
end

to handle updateStatus
    getRemote "text of field ""user name"" application \
        "ToolBook" topic "myapp.tbk"
    if item 1 of sysError is "OK"
        text of field "Status" = It
    else
        request "Could not establish DDE link."
    end if
end updateStatus

to handle leavePage
    closeRemote application "ToolBook" topic "myapp.tbk"
    forward
end
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Tips for using DDE with ToolBook II

When working with DDE in ToolBook II, remember that:

- ◆ DDE conversations are associated with an instance, not a book. If you have the same book open in two instances of ToolBook II, only the most recently active instance will respond to a DDE request.
- ◆ An open but inactive DDE conversation does not consume local memory.
- ◆ The maximum number of DDE conversations possible is limited by system resources. If system resources are exhausted, an out-of-memory error occurs.

Using ToolBook II as a server

ToolBook II responds as a DDE server to any Windows application that sends the Windows DDE messages WM_DDE_REQUEST, WM_DDE_POKE, or WM_DDE_EXECUTE. (If these DDE requests originate in another instance of ToolBook II, they are sent by the *getRemote*, *setRemote*, and *executeRemote* commands.)

For example, if you are running ToolBook II and have the book Sample.tbk open, you can send this command from a second instance of ToolBook II:

```
getRemote "text of field Index of page 1" application \  
"ToolBook" topic "sample.tbk"
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

The first instance of ToolBook II—the one currently running the Sample.tbk book—will respond as if you had run this command from the Command window in that instance:

```
get text of field "Index" of page 1
```

When ToolBook II is acting as a DDE server, it responds to the WM_DDE_REQUEST message as if the message is an OpenScript *get* command. Similarly, ToolBook II responds to the WM_DDE_POKE message with a *set* command, and to the WM_DDE_EXECUTE message with an *execute* command. In all cases, ToolBook II runs DDE messages as if they have been run in the Command window, so the context of the commands is the current page.

Controlling the ToolBook II response to DDE requests

If you want to control how a particular ToolBook II book responds to DDE requests, you can create *remoteGet*, *remoteSet*, or *remoteCommand* handlers. These handlers are called when another Windows application sends WM_DDE_REQUEST, WM_DDE_POKE, or WM_DDE_EXECUTE messages, respectively. Using these handlers allows you to restrict access to a book that is accessed using the DDE channel or to create custom DDE-only behavior in a book. For example:

```
--Prevents another application from changing  
--a value with the DDE WM_DDE_POKE message  
to handle remoteSet  
    request "You cannot change values using DDE." \  
        & CRLF & "Book:" && name of this book  
end
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Handlers that respond to DDE-event messages must forward the messages so that ToolBook II can execute the command, or ToolBook II returns *Failed: Denied* to the client. Alternatively, you can include a *respondRemote* statement to define a custom response to the DDE request. For example, the following handlers turn off the book as a DDE server altogether:

```
--Place in a book script
to handle remoteGet
    respondRemote "Failed: Denied"
end

to handle remoteSet
    respondRemote "Failed: Denied"
end

to handle remoteCommand
    respondRemote "Failed: Denied"
end
```

Note You should use only one *respondRemote* command in each handler, because ToolBook II ignores any other *respondRemote* commands after the first one in that handler.

Handlers for OpenScript DDE-event messages are usually placed in the book script so that the response to a DDE message is the same regardless of which page is displayed.

Extending DDE-event handlers

The DDE-event messages *remoteGet*, *remoteSet*, and *remoteExecute* include a *<command>* parameter that contains the text of the command sent by the DDE client.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

For example:

```
--This handler displays the text of a WM_DDE_REQUEST command  
to handle remoteGet command  
    request command  
    forward  
end
```

You can use the *<command>* parameter to receive additional information that was passed with the DDE event messages. For example, you can create a *remoteGet* handler that returns not just the value of a property, but the value of a system variable as well. The client sends a request such as this:

```
getRemote "system myVar" application "ToolBook" topic "myapp.tbk"
```

The *remoteGet* handler in the server evaluates the request, looking for the word *system* as the first word of the request string. If the first word is *system*, the second word is the name of the variable to return. The handler uses *execute* to declare the system variable, and then *evaluate* to get its value:

```
to handle remoteGet command  
    if word 1 of command is "system"      --Request for system variable  
        execute command                  --Declares variable  
        It = evaluate(word 2 of command)  --Sets it to the value of the  
                                          --system variable  
        respondRemote "OK"              --Sets sysError value  
    else  
        forward  
    end if  
end
```



For details about using the *execute* and *evaluate* commands, refer to their entries in the OpenScript reference in online Help, and see Chapter 8, "Writing common scripts."



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries**
- 14 Debugging OpenScript programs

Chapter 13

Using dynamic-link libraries

Dynamic-link libraries (DLLs) contain functions to which you can link an application at run time to perform particular tasks not supported in OpenScript. Windows itself is composed of DLLs, which can be called from OpenScript to perform low-level tasks and interact with other applications.

This chapter describes how to use DLLs to extend the functionality available to your application. This chapter also explains how to connect to the Windows messaging system from OpenScript. In addition, if you are writing DLLs for ToolBook II, the last topic in this chapter discusses special features you can use to make your DLLs more powerful.

CONTENTS

About DLLs	284
Linking and declaring DLL functions	287
Allocating Windows memory	295
Calling Windows API functions	300
Translating Windows messages in ToolBook II	300
Writing DLLs to use with ToolBook II	309



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

About DLLs

You can extend the power of ToolBook II by linking your applications to dynamic-link libraries (DLLs). DLLs are executable files containing functions that can be linked dynamically (on demand) by an application (*linkDLL*), and then called to perform useful tasks, such as getting file and device information from Windows. You need to know the function names and their parameter types. You should also know as much as possible about what the functions do.

The following DLLs are included with ToolBook II and can be called from your application:

- ◆ **Tbwin.dll** Windows-related functions for ToolBook II to get system resources, access Win.ini, and so on
- ◆ **Tbdos.dll** Utility functions for manipulating DOS files from OpenScript
- ◆ **Tbdb3.dll** dBASE III support for ToolBook II
- ◆ **Tbpdx.dll** Paradox database support for ToolBook II
- ◆ **Tbdlg.dll** Common dialog box functions
- ◆ **Tbfile32.dll** 32-bit file and common dialog box functions

Windows itself is composed of DLLs. The following DLL modules provide the application programming interface (API) used by all Windows applications to interact with the system. You can access these API functions from OpenScript as well:

- ◆ **KERNEL** Windows operating system basic functions, such as memory allocation, file I/O, and resource management
- ◆ **USER** Windows user interface functions
- ◆ **GDI** Windows graphic display functions



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Many suppliers of development tools provide their products as DLLs that you can call from your ToolBook II scripts. You can also write your own DLLs to enhance your applications using C, C++, Pascal, or another language that supports creating DLLs. For details about how to call the DLLs supplied with another product, refer to the documentation that came with the product.

DLLs and ToolBook II

Using a DLL in ToolBook II is a two-step process:

- 1 Link the DLL to make it available to your application.
- 2 Call functions in the DLL from your script.

The *linkDLL* control structure tells ToolBook II which library you want to link to, which functions you want to access in the library, and the data types for their parameters and return values. For example:

```
--This DLL contains routines for manipulating DOS files
linkDLL "tbdos.dll"
  --Declares a function that returns an integer data type
  INT moveFile(STRING,STRING)
  --Parameters passed to function are strings
  INT setCurrentDirectory(STRING)
end
```

For details, see “Specifying data types for DLL functions” later in this chapter.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

After you have linked and declared a DLL function, you call it from a script using the same syntax as OpenScript functions. For example, the following script uses the *moveFile()* function to move the file *Train.tbk* to another directory and rename it:

```
get moveFile("c:\toolbook\train.tbk","c:\archive\train1.tbk")
if It is not 1
    request "Error moving file:" && It
end
```

The integer returned by this particular function represents an error code. A value of 1 indicates success; any other number indicates an error. This strategy is common for indicating errors in DLL calls. It is good programming practice to always check the return values of DLL calls for errors.



For a complete list of the error codes used by the *moveFile()* function, refer to “DLL reference” in the OpenScript reference in online Help.

Because they always return a value, DLL functions are called the same way as OpenScript functions, even though some DLL functions may look more like commands. For example, the following handler calls the *setCurrentDirectory()* function:

```
to handle buttonClick
    get setCurrentDirectory("c:\windows")
    if It is not 1
        request "Error setting directory:" && It
    end if
end buttonClick
```

Both of the previous examples call the DLL functions with the *get* command, which places the return value into *It*. You can also put the return value into a variable or property as shown below:

```
errorCode = setCurrentDirectory("c:\windows")
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

The sections that follow provide more information about how to use DLLs.

Linking and declaring DLL functions

The *linkDLL* control structure uses this syntax:

```
linkDLL <dll name>  
  <return type> [<alias name>=<function name>  
  ↳ ([<parameter type list>])  
end [linkDLL]
```

The *<dll name>* parameter is the name of the DLL file; for example, *Tbwin.dll*. If the file extension of the DLL is *.exe*, you can omit the extension, but you must specify other file name extensions, such as *.dll* or *.drv*.

ToolBook II puts the name of linked DLLs into a list in the *sysLinkedDLLs* system property. Each item in the list is the full path name of a linked DLL. You can use the list in the *sysLinkedDLLs* property to verify that a DLL is linked:

```
to handle buttonClick  
  dllFileName = "tbdos.dll"  
  if dllFileName is not in sysLinkedDLLs then  
    linkDLL dllFileName  
    STRING getCurrentDrive( )  
    end linkDLL  
  end if  
  ... --Further statements here  
end buttonClick
```

Note You can also use the *DLLfunctions()* function to return a list of functions linked in a DLL.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

To link Windows libraries, specify the module name rather than the file name to ensure that your scripts perform properly on systems where these modules can have different file names. Module names are KERNEL, USER, GDI, and DISPLAY (not Krnl386.exe, User.exe, Gdi.exe, or Wn800.driv). For example, to link to memory allocation functions in Windows, you would use this *linkDLL* statement:

```
linkDLL "kernel"
```

Aliasing function names

When you link to a DLL, you usually specify the function as it is named in the DLL. Sometimes, however, you might need to create an alias for a function. Typical situations in which you might want to use an alias are:

- ◆ when the DLL function name is the same as a ToolBook II keyword.
- ◆ when the name of the function is not descriptive.
- ◆ in the rare case when you need to simultaneously link the same function with two different sets of data type declarations.

For example, you must use an alias when a DLL function name conflicts with the name of an OpenScript command or function. The following example establishes *winOpenFile* as the alias for the *openFile()* DLL function in KERNEL, because *openFile* is an OpenScript command:

```
linkDLL "kernel"          --Do not specify .exe  
    INT winOpenFile = openFile(STRING, POINTER, WORD)  
end
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Some Windows DLLs export functions by a number, called an ordinal, instead of a name. You can create an alias to refer to the export ordinal reference of a function when declaring the function in a *linkDLL* statement. The following example establishes the alias *localFunction* for a function based on its ordinal reference of 7:

```
linkDLL "local.dll"  
    WORD localFunction = 7(WORD, INT)  
end
```

Where to link a DLL

You can link a DLL at any time before you call it. Typically, your *enterApplication* or *linkSysbook* handler will contain the *linkDLL* statements so that the functions are available when the book is opened. Each link to a function takes both time and memory; therefore, declare only the functions you need from a DLL.

If you link a DLL more than once, the new function declaration replaces any previous declarations for that function, unless they have different aliases. The only impact on your application when you relink a DLL is the time it takes to execute.

Tip When you debug an application, you will sometimes find it necessary to relink the DLLs or alter their declarations. To make this task easier, group your *linkDLL* structures in their own handler:

```
to handle linkAppDLLs  
    ...    --Link functions here  
end
```

Then, put a *send linkAppDLLs* statement in your *enterApplication* handler.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Specifying data types for DLL functions

You must specify the number and type of arguments for each function in your *linkDLL* control structure.



For a list of the functions available in the DLLs supplied with ToolBook II and information on how to declare them, refer to “DLL reference” in the OpenScript reference in online Help.

DLLs are most commonly documented by their C data types. The following tables provide ToolBook II translations for common data types in DLLs.

Data types for DLL function parameters

Windows C data type	<i>linkDLL</i> parameter type	<i>linkDLL</i> return type
int	INT	INT
BOOL	INT	INT
void †	<none>	INT
unsigned char, BYTE	BYTE	BYTE
UINT, unsigned int, WORD	WORD	WORD
HANDLE, HWND, HBMP, HDC, and so on	WORD	WORD
long	LONG	LONG
unsigned long	DWORD	DWORD
FLOAT	FLOAT	FLOAT
DOUBLE	DOUBLE	DOUBLE

† The return type of a void C function requires the INT return type in OpenScript. This is because when you call a function in OpenScript, you use the *get* command, which requires a return value. The void type in C actually returns an INT as well.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Note Some of the declarations for function return values in the table on the previous page differ from the declarations for parameters. ToolBook II cannot verify that the information passed to and from DLLs is of the correct data type, and so incorrect declarations can result in unpredictable behavior or cause Windows to crash. If you encounter problems using a DLL, verify your function declarations as a first debugging step.

String data types

Windows C data type	linkDLL parameter type	linkDLL return type
VOID FAR *, CHAR FAR *, LPSTR, LPCSTR, or POINTER (Using the ToolBook II string return protocol) [†]	STRING	POINTER STRING

[†] For details, see “Strings as return values” later in this chapter.

Boolean values

The BOOL data type indicates an integer that should have the value of 0 to indicate false or 1 to indicate true. Declare BOOL functions and parameters as INT and use values of 1 or 0 (not OpenScript *true* or *false*).



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

The *void* data type

The *void* data type in C means two different things. As a return type, it means that the function returns an integer, which is ignored. As a parameter type, it means that no parameters should be passed. For example:

```
--The function declared in C
void updateEntry(void)

--The same function declared in your linkDLL statement
int updateEntry( )
```

Note Do not confuse the *void* data type with declarations such as *void FAR **, which are pointers.

Passing strings as parameters

Working with the string data type in your DLL function, you can pass constant strings, declare string buffers, and return string values to ToolBook II. The following sections provide details.

Constant strings

If a DLL function requires string data for input only—that is, it does not change the value of the string—you declare the parameter as `STRING` and pass either a ToolBook II constant (a string in quotes) or a ToolBook II property or variable. The `moveFile()` and `setCurrentDirectory()` examples earlier in this chapter pass constant strings.

When passing a string parameter from ToolBook II, you can pass an empty string to a DLL by using a pair of quotation marks (" "). Because all strings are passed as long pointers, ToolBook II passes the quotation marks as a long pointer to a null string, not a null pointer. However, if you pass a value of *null*, ToolBook II will give a *null* pointer to the DLL.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

String buffers

If the function you are linking to modifies a string or returns data in a buffer, declare the data type as *POINTER* rather than *STRING* and pass a pointer to Windows memory. The following example uses the *getMemBlock()* and *freeMemBlock()* functions from *Library.tbk* to allocate a buffer to receive the name of the Windows directory. The program deliberately allocates more memory than the maximum string length specified to allow for the *NULL* character that C uses to terminate all strings. The *pointerString()* function is used to convert the Windows buffer to a ToolBook II variable.

```
to handle buttonClick
  linkDLL "kernel"
    --POINTER is a buffer to receive a string
    --WORD is the maximum length for that buffer
    WORD getWindowsDirectory(POINTER,WORD)
  end linkDLL
  linkDLL "tbwin.dll"
    pointer getMemBlock (DWORD)
    word freeMemBlock (POINTER)
  end
  lpDir = getMemBlock(120)
  get getWindowsDirectory(lpDir,119)
  if (It is not 0)  --Checks for error
    dirName = pointerString(0,lpDir)
    request "Win directory:" && dirName
  else
    request "Failed to get directory."
  end if
  get freeMemBlock(lpDir)
end buttonClick
```

For details about using Windows memory, see “Allocating Windows memory” later in this chapter.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Strings as return values

There are two formats for returning string information to ToolBook II from a DLL. You use one format with functions that return a LPSTR (long pointer to a string), and another with functions that return a value declared as a string.

Note This discussion applies only to the return value of the DLL, not to the data types for the parameter.

If a function returns a LPSTR or a CHAR FAR *, declare it as returning a POINTER and use the OpenScript *pointerString()* function to convert the value to a ToolBook II data type. You can declare this type of function as shown in the following example:

```
linkDLL "sample.dll"  
    POINTER funcGivesLPSTR(  
end
```

Then, call the DLL function and use the OpenScript *pointerString()* function to convert the global pointer to a ToolBook II string, as in the following example:

```
get funcGivesLPSTR(  
vStringVar = pointerString(0,It)
```

You use a different format with functions that return a value as a string. For return values, ToolBook II reserves the STRING data type as a special protocol that includes both the string data and error information to be placed in *sysError*.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

DLLs you specifically design to work with ToolBook II use functions with return values declared as strings. For example:

```
linkDLL "tbfile.dll"  
    STRING getCurrentDrive( )  
end
```

For technical details about this protocol, see “Writing DLLs to use with ToolBook II” later in this chapter.

Allocating Windows memory

If your DLL requires you to pass arguments by reference—that is, pass a pointer to them—or to pass buffers, you need to allocate Windows memory and pass the pointer to that memory.

Tbwin.dll contains *getMemBlock()*, which allocates memory for use, and *freeMemBlock()*, which frees memory you have finished using. You should always free memory buffers when you are finished with them so that the memory is available for other uses. However, if you want to maintain a buffer over a period of time, you can store the pointer in a system variable:

```
to handle enterBook  
    link getMemBlock() and freeMemBlock()  
    system bufferPointer  
    bufferPointer = getMemBlock(1024)  
    forward  
end  
  
to handle leaveBook  
    system bufferPointer  
    get freeMemBlock(bufferPointer)  
    forward  
end
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Using the scripts on the previous page, any handler that declares *bufferPointer* as a system variable can use that memory. To get and set values within such a buffer, you use the functions described in the next section.

Accessing memory with *pointer<type>()* functions

OpenScript includes a set of *pointer<type>()* functions to copy data between the ToolBook II variable and the Windows global memory. Each of the following *pointer<type>()* functions corresponds to a data type; you can use them to either set or get data within a buffer.

<i>pointerByte()</i>	<i>pointerFloat()</i>	<i>pointerPointer()</i>
<i>pointerDouble()</i>	<i>pointerInt()</i>	<i>pointerString()</i>
<i>pointerDword()</i>	<i>pointerLong()</i>	<i>pointerWord()</i>

The general syntax of the *pointer<type>* function is

```
pointer<data type>(<offset>,<pointer>[,<set value>])
```

Where:

- ◆ *<data type>* is the portion of the function name that identifies the type of the data to be copied by the function. The function you choose depends on the type of data you expect the function to return. The data type name corresponds to C data types, as listed in the table on the following page.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Data types for values passed by reference

Data type	Description
BYTE	8-bit unsigned value (unsigned char)
INT	16-bit signed value
WORD	16-bit unsigned value (unsigned int)
LONG	32-bit signed value
DWORD	32-bit unsigned value (unsigned long)
FLOAT	32-bit Microsoft-format floating point value
DOUBLE	64-bit IEEE†-format floating point value
STRING	NULL-terminated array of 8-bit unsigned values interpreted as characters
POINTER	32-bit FAR pointer represented by a two-item list of unsigned integers: <i><segment></i> , <i><offset></i> ; can be a pointer to any data type

† Institute of Electrical and Electronics Engineers

◆ *<offset>* is the number of bytes offset into the buffer pointed to by *<pointer>*.

Note If you attempt to set a value that has an *<offset>* outside the buffer, you might generate a general protection fault.

◆ *<pointer>* is the Windows memory block you wish to access. This parameter is treated as a void pointer—that is, it can be a pointer to any valid memory location without regard to the data type of its contents. Pointers are represented in OpenScript by a list of comma-separated, unsigned integers: *<segment>*,*<offset>*. The *<pointer>* parameter can point to memory objects with more than 64 KB of data.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

- ◆ `<set value>` is a value of a type that matches `<data type>`. If no `<set value>` is supplied, ToolBook II returns the current value. If a value is supplied, the function fills in the memory location pointed to by the `<pointer>` plus `<offset>` with `<set value>`. This parameter is optional.

A *pointer*`<type>()` example

A common use for OpenScript *pointer*`<type>()` functions is to manipulate database records. The following example shows C declarations for two data retrieval functions in an imaginary database DLL called MyBase.dll. Each function takes three parameters: a record number (*lRecord*), a field number (*lField*), and a data value of type *long* passed as a pointer. The return value, an integer, is an error code:

```
int getData(long lRecord, int nField, long far *lpValue);  
int setData(long lRecord, int nField, long far *lpValue);
```

In OpenScript, declare these functions in this way:

```
linkDLL "mybase.dll"  
  int getData(long, int, pointer)  
  int setData(long, int, pointer)  
end
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

To get or set a value from this database, first use a pointer allocation function to copy the data from memory to a ToolBook II variable, and then manipulate it as you would any other variable:

```
--Sets up variables for what data to retrieve
vRecord = 10                --Get the 10th record
vField = 4                  --Get the 4th field in record
vPointer = getMemBlock(4)   --Longs are 4 bytes
--Gets a value from the database, and returns its memory location in
--vPointer; the return value of the function is an error code
get getData(vRecord, vField, vPointer)
--Copies data from memory to OpenScript using pointerLong( )
set vLongValue to pointerLong(0,vPointer)
```

The ToolBook II variable *vLongValue* now holds the value from the database record. To set data using *pointerLong()*, change the value of the OpenScript variable, and then put the data back in the database using the *setData()* DLL function:

```
increment vLongValue        --Changes the data
--Copies value to the memory buffer
get pointerLong(0, vPointer, vLongValue)
get setData(vRecord, vField, vPointer) --Writes it to database
get freeMemBlock(vPointer)  --Frees memory
```



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Calling Windows API functions

You can link the Windows modules KERNEL, GDI, and USER to call their functions just as you do other DLLs.

For details about Windows API function names and the parameters they take, refer to:

- ◆ the Library.tbk and Win31wh.hlp files included with ToolBook II.
- ◆ the Winconst.hlp file, also included with ToolBook II (provides definitions for constants and declarations for APIs).
- ◆ the *Guide to Programming*, which is included in the Microsoft Windows SDK.
- ◆ any of the wide variety of books on Windows API programming available commercially.

Translating Windows messages in ToolBook II

This section contains information that assumes you have some knowledge of the Windows messaging system.

Important Translating some messages can cause applications to stop responding to user actions. Be sure to save your work frequently when experimenting with message translation, and test your scripts thoroughly.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Like ToolBook II, Windows is an event-driven messaging environment; when a user performs an action, Windows sends a message to the current window. When you use ToolBook II, Windows sends the message to ToolBook II, and ToolBook II in turn converts this Windows message into the OpenScript message that you are familiar with. For example, the Windows message WM_LBUTTONDOWN becomes *buttonDown* in ToolBook II, and WM_CHAR becomes *keyChar*.

For the most part, you can handle a Windows message using its OpenScript equivalent. However, you might want to handle Windows messages that have no ToolBook II equivalent. You can use *translateWindowMessage* in your scripts to receive and respond directly to Windows messages.

The *translateWindowMessage* command causes ToolBook II to translate the Windows message into an OpenScript message. Your script can then respond to almost any Windows messages for any running application, even non-ToolBook II applications.

When you translate a message, you substitute your handler for the ToolBook II default response to the message. For example, if you translate the Windows message WM_LBUTTONDOWN, your handler is called whenever a user clicks the left mouse button down. Your handler determines the response to the message.

As with any message, you can forward Windows messages up the object hierarchy. If the message reaches the ToolBook II system, it triggers the ToolBook II default response to the message. If you want to handle the message before ToolBook II does, you forward the message after processing it. If you want ToolBook II to respond first, you forward the message, and then process it. If you want to substitute your response altogether for the ToolBook II response, you can process the message without forwarding it. Some Windows messages (such as WM_MOUSEACTIVATE) require a return value, and *translateWindowMessage* supports this as well.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

As a simple example, the following script translates the Windows message WM_WININICHANGE (message number 0x001A) into a ToolBook II user-defined message, *winIniChanged*. Translating this Windows message allows you to write scripts that are notified when there is any change to the Win.ini file:

```
translateWindowMessage for sysWindowHandle
  --0x001A = WM_WININICHANGE
  --Indicates that Win.ini has been changed
  on 0x001A send winIniChanged
end
```

The general steps for working with Windows messages are:

- 1 Establish Windows-to-OpenScript message translation with the *translateWindowMessage* control structure.
- 2 Handle the OpenScript message with a *to handle* or *to get* handler.
- 3 Cancel message translation with the *untranslateWindowMessage* or *untranslateAllWindowsMessages* command.

Establishing translation of Windows messages

The *translateWindowMessage* control structure establishes the Windows-to-OpenScript message translation. For messages that send a ToolBook II message, the syntax is:

```
translateWindowMessage [for <winHandle>]
  on <winMsg> send <tbkMsg> [to <tbkObj>]
end [translateWindowMessage]
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

The `<winHandle>` parameter is the handle of the window receiving the message. For ToolBook II applications, it is either the *clientHandle* of a viewer, which receives most user input and redraw messages, or the *windowHandle* of a viewer, which receives application messages. For non-ToolBook II applications, you need to supply the appropriate handle. (For details, refer to the *findWindow()* API example in Library.tbk.) The default value is the ToolBook II main window, *sysWindowHandle*.

The `<winMsg>` parameter is the message number for the Windows message to be translated. You can find the hexadecimal value for each Windows message in the Winconst.hlp file, which is included with ToolBook II. The Win31wh.hlp file supplied with ToolBook II contains a description of each Windows message.

Note ToolBook II reserves the message numbers between 1124 and 1224 (WM_USER+100 to WM_USER+200) for its internal messages. Do not redefine or intercept these messages; doing so can cause system errors and data corruption.

The `send <tbkMsg> [to <tbkObj>]` statement indicates the OpenScript message to send and an optional target object. If no object is specified, the current page is the target.

For messages that return a value, the syntax is:

```
translateWindowMessage [for <winHandle>]
    on <winMsg> get tbkMsg [of <tbkObj>] return <dllType>
end [translateWindowMessage]
```

This syntax closely resembles the syntax you use to send a ToolBook II message. The difference is that instead of a message handler, you can specify an object property to be returned or a *to get()* handler to be called. The `<dllType>` parameter specifies the C data type ToolBook II must use to return this value. This value is the same as for a DLL function of the same type, usually an INT.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Guidelines for translating Windows messages

When translating Windows messages, keep in mind the following:

- ◆ If no target object is specified for the translated ToolBook II message, the ToolBook II message is sent to the current page.
- ◆ If the target object is destroyed after a translated message has been defined for that object, creating a new object with the same name will not restore the translation.
- ◆ Message translation stops if a translated message is being sent to a book and the book is saved with a new name.
- ◆ Utilities such as Spy.exe (supplied with the Microsoft Windows SDK) display all messages sent to a window, which can be useful for determining what messages to translate.

Note Each message can have only one current translation. For example, you cannot translate WM_LBUTTONDOWN more than once for the same window. Only the most recent translation works.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Handling translated messages

Your application must have a handler to respond to the translated message. When ToolBook II sends the message, it provides parameters that match those found at the Windows system level. The meaning of these parameters is different for each Windows message, and, in many cases, your handler can ignore this information. Refer to the following table for descriptions of the parameters.

Windows parameters passed with translated messages

Parameter	Description
<i>hwnd</i>	Handle of the window that received the Windows message
<i>wmsg</i>	Numeric value for the Windows message that is to be translated into a ToolBook II message
<i>wp</i>	< <i>wParam</i> > parameter for the Windows message
<i>lplo</i>	Low-order word of < <i>lParam</i> > parameter for the Windows message
<i>lphi</i>	High-order word of < <i>lParam</i> > parameter for the Windows message
<i>lParam</i>	< <i>lParam</i> > parameter of the Windows message expressed as a pointer



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

For example, you can translate the Windows message WM_WININICHANGED into the ToolBook II message *winIniChanged*. Then, you can write a handler to check the string pointed to by the *lParam* of the *winIniChanged* message, finding out which section of Win.ini was modified. The following example shows a pointer to the name of the changed section of the Win.ini file.

```
--Establishes Windows message translation
to handle enterBook
    translateWindowMessage for sysWindowHandle
        --Indicates that Win.ini has been changed; 0x001A = WM_WININICHANGE
        on 0x001A send winIniChanged
    end translateWindowMessage
    forward
end enterbook

--Handles the message sent by a translated Windows message
to handle winIniChanged hwnd, wmsg, wp, lpl, lphi, lParam
    --Returns name of changed section in Win.ini
    set vSection = pointerString(0,lParam)
    if vSection is "MyAppName"
        --If section matches this app, sends a reset to the application
        send ResetPreferences
    end if
    --Otherwise ignores it
    forward
end winIniChanged
```



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Returning a value

The example below translates WM_QUERYOPEN (Windows message 0x0013) to return a user property or the result of a *to get* function. Windows expects a Boolean value: zero to prevent a minimized instance from being activated, or a nonzero number to allow the instance to be activated. In the example, translation occurs after the normal handling of the message so that the value supplied by OpenScript overrides the value returned by the application.

```
--anyHandle is a local variable containing a window handle
translateWindowMessage for anyHandle
  --0x0013 = WM_QUERYOPEN, which determines if minimized application
  --will allow itself to be opened
  on 0x0013 get openIcon of this book return INT
end
```

The words *return INT* in this example indicate that an INT data type is required for this message. After the translation above is established, when ToolBook II receives a WM_QUERYOPEN message, it looks for either of the following:

- ◆ A user-defined function that defines the response and return value for the WM_QUERYOPEN message. For example:

```
to get openIcon of this book hwnd, wmsg, wp, lplo, lphi, lParam
  system handleToLock
  if hwnd is handleToLock
    return 0
  else
    return 1
  end if
end openIcon
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

This function must be placed in the script of the book or higher in the hierarchy because the translation statement specified *of this book*.

- ◆ A user property of the book. Because the WM_QUERYOPEN message in Windows returns an integer value, the user property in ToolBook II must likewise contain an integer value. For example:

```
openIcon of this book = 1
```

Tip You can control whether a minimized ToolBook II application can be opened by setting the *state* property of a viewer to *minimized* or *lockedMinimized*.

Canceling the translation of Windows messages

You can cancel message translation by doing any of the following:

- ◆ Executing the *untranslateWindowMessage* command for a specific Windows message
- ◆ Executing the *untranslateAllWindowsMessages* command for a particular window handle
- ◆ Exiting ToolBook II
- ◆ Closing the window for which the message is being translated (all translations are canceled after the window receives the WM_NCDESTROY message)
- ◆ Deleting the target object of the *translateWindowMessage* command



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Writing DLLs to use with ToolBook II

Writing a DLL to use with ToolBook II is much like writing a DLL to use with any other application. However, ToolBook II provides some special features that can make your DLLs more powerful and efficient if you plan to use them with ToolBook II.

Note The information in this section assumes you have some knowledge of the C programming language and are familiar with writing DLLs. Technical Support can help you with linking to DLLs, as described earlier in this chapter, but not with the programming tasks needed to create DLLs.

Declaring strings

ToolBook II passes the address of a string to a DLL as a FAR pointer into ToolBook II memory space. To protect against accidentally damaging ToolBook II data, it is a good idea to declare string parameters in your DLL as LPCSTR (long pointer to a constant string) rather than LPSTR (long pointer to a string) unless you intend to modify them; doing so causes the compiler to flag any statement that attempts to modify the value of the string. You also cannot assume that any string will remain in a fixed location, so you should not pass the address of a ToolBook II string through the Windows messaging system.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Returning strings for ToolBook II to use

ToolBook II supports a string return type that provides extra error handling and simplifies memory management. If you are writing a DLL for ToolBook II, you can use this type to simplify the OpenScript code that calls your function. The DLL function is declared in the *linkDLL* statement as returning a STRING, as shown below:

```
linkDLL "tbdos.dll"  
    STRING getDriveList( )  
end
```

This tells ToolBook II to expect an error code for *sysError*. If there is no error, the DLL returns a string that is automatically placed in the ToolBook II variable. The calling OpenScript statements might look like this:

```
clear sysError  
driveList = getDriveList( )  
if sysError is null  
    request "Available drives:" && driveList  
else  
    request "Error Getting Drives:" && sysError  
end
```

Inside the DLL, the function returns a DWORD where the HIWORD contains a value for *sysError* and *sysErrorNumber* and the LOWORD contains the handle to a string. If the HIWORD is NULL (zero), ToolBook II locks the string handle in the LOWORD, retrieves its value, and unlocks and frees the handle. If the function needs to return an empty string, the LOWORD must still contain a handle for a null-terminated string of length zero. To indicate an error, the DLL sets the HIWORD to any value other than NULL; and ToolBook II ignores the LOWORD, returns a null string to OpenScript, and sets *sysErrorNumber* to the value stored in the HIWORD.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

You can also return a LPSTR to ToolBook II by specifying a return type of POINTER. However, the OpenScript code must retrieve the string using the *pointerString()* function and unlock and free the memory. Using the STRING return type avoids these steps and provides clear error handling.

The following example shows a DLL function that creates a DWORD for a ToolBook II STRING return type. A new copy of the input string is allocated so that the caller frees its memory normally. This function unlocks the new memory, but ToolBook II is responsible for freeing it.

```
#define MEMORY_ERROR -1
#define INVALIDPARAM_ERROR -2
DWORD_export FAR PASCAL OpenStrRet(LPSTR lpszReturnString,
    int err_value)
{
    LPSTR lpBuffer;
    HANDLE hBuffer;
    DWORD dwResult = (DWORD) MAKELONG(NULL, MEMORY_ERROR);
    int sLength;

    //if err_value is already set, lpszRetrunString is a null pointer,
    //or lpszReturnString is a pointer to a 0 length string set the
    //return value to the current error value.
    if (err_value || !(lpszReturnString) ||
        !(sLength = strlen(lpszReturnString)))
    {
        dwResult = (DWORD)MAKELONG(NULL, err_value);
    }
    else
    {
```

(continued)



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

```
//if memory can be allocated, lock it and do the copy.
hBuffer = GlobalAlloc(GMEM_ZEROINIT|GMEM_MOVEABLE,
    (DWORD) sLength + 1);
if ( hBuffer != 0 )
{
    //if memory cannot be locked, free the buffer and exit.
    lpBuffer = GlobalLock(hBuffer);

    if (!lpBuffer)
    {
        GlobalFree(hBuffer);
    }
    else
    {
        // lstrcpy returns NULL on an error.
        if (!lstrcpy(lpBuffer, lpszReturnString))
        {
            GlobalUnlock(hBuffer);
            GlobalFree(hBuffer);
        }
        else
        {
            GlobalUnlock(hBuffer);
            dwResult = (DWORD) MAKELONG(hBuffer, err_value);
        }
    }
}
return (dwResult);
}
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

The following example converts the buffer returned by the Windows *getPrivateProfileString()* function to a comma-delimited list of section names for easy access in OpenScript. This function uses the *openStrRet()* function defined earlier.

```
#define MAX_INI_BUFFER 1024
DWORD _export FAR PASCAL getINIEntries(LPSTR lpszPathAndFile,
    LPSTR lpszSection)
{
    HANDLE hBuffer;
    LPSTR lpBuffer = NULL,
        lpParsePtr;
    DWORD dwResult = openStrRet(NULL, MEMORY_ERROR);

    if (!lpszPathAndFile || !*lpszPathAndFile || !lpszSection ||
        !*lpszSection)
    {
        dwResult = openStrRet(NULL, INVALIDPARAM_ERROR);
    }
    else
    {
        hBuffer = GlobalAlloc(GMEM_ZEROINIT|GMEM_MOVEABLE,
            MAX_INI_BUFFER);

        //if allocate worked then lock it and do the copy.
        if (hBuffer != 0)
        {
            //if it cannot be locked, free the buffer and exit.
            if ((lpBuffer = GlobalLock(hBuffer))
                { //if any bytes are copied, turn null separators
                // to commas.
                if (GetPrivateProfileString(lpszSection, NULL, "",
                    lpBuffer, MAX_INI_BUFFER, lpszPathAndFile) != 0)
```

(continued)



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

```
{ //move lpParsePtr through buffer looking for the
//double null terminator, converting single nulls to
//commas to make a list.
lpParsePtr = lpBuffer;
while ( !(lpParsePtr[0] == '\0' && lpParsePtr[1] ==
'\0') )
{
if (lpParsePtr[0] == '\0')
{
lpParsePtr[0] = ',';
}
++lpParsePtr;
}
dwResult = openStrRet(lpBuffer, NULL);
}
GlobalUnlock(hBuffer);
}
GlobalFree(hBuffer);
}
}
return(dwResult);
}
```

Calling OpenScript from a DLL

Your DLL can access OpenScript commands and expressions directly by sending special messages. You can use this callback protocol to get information from the ToolBook II application calling your DLL, or your DLL can execute OpenScript statements within the context of that application. For example, a DLL that does an extended search can report its progress by executing a *send ReportInfo* message, or you can have a hardware device continually query the position of a control knob object.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

To accomplish this task, you use two special Windows messages. These are:

- ◆ **TBM_EXECUTE** Sent to the instance of ToolBook II to execute an OpenScript statement within that application
- ◆ **TBM_EVALUATE** Sent to an instance of ToolBook II to get data or evaluate an OpenScript expression

TBM_EVALUATE and TBM_EXECUTE correspond to the OpenScript commands *evaluate* and *execute* and are registered by ToolBook II at startup. To register the messages in your DLL, use the Windows API function *registerWindowMessage()*, which returns the same message number assigned to ToolBook II by Windows. For example, your DLL might include two global variables, *wExecute* and *wEvaluate*, which you would initialize with these lines:

```
wExecute = RegisterWindowMessage("TBM_EXECUTE");  
wEvaluate = RegisterWindowMessage("TBM_EVALUATE");
```

Because you want these actions to happen synchronously, use the *sendMessage()* function rather than the *postMessage()* function to execute these commands. The return value of *sendMessage()* is TRUE if the statements execute or evaluate successfully and FALSE if there is an error. The syntax for *sendMessage()* is:

```
sendMessage(hWnd, wMsg, wParam, (LONG)lParam)
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

The table below shows how these arguments are interpreted for the TBM_EXECUTE message.

Arguments for *sendMessage()* when used with TBM_EXECUTE

Parameter	Description
<i>hWnd</i>	Specifies the window handle that is to receive the message by passing the ToolBook II <i>sysWindowHandle</i> as a parameter to a DLL initialization function
<i>wMsg</i>	The registered Windows message
<i>wParam</i>	TRUE (nonzero) if <i>sysSuspend</i> is set to <i>false</i> during command execution
<i>lParam</i>	(LPSTR) Points to a zero-terminated string containing the OpenScript statements to be executed (as with all <i>execute</i> strings, multiple lines can be separated with semicolons)



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

The table below shows the arguments that the TBM_EVALUATE messages takes.

Arguments for *sendMessage()* when used with TBM_EVALUATE

Parameter	Description
<i>wMsg</i>	The registered Windows message
<i>wParam</i>	TRUE (nonzero) if <i>sysSuspend</i> is set to <i>false</i> during command execution
<i>lParam</i>	FAR pointer to an evaluation buffer containing the expression to be evaluated and a buffer to receive the result

The structure for the evaluation buffer of the TBM_EVALUATE message is defined as:

```
typedef struct tagTBMEval {  
    LPSTR  lpExpression;  
    WORD  wRetType;  
    LPVOID lpRetVal;  
    WORD  nRetValLength;  
} TBMEVAL;
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Where:

- ◆ *lpExpression* defines a null-terminated string containing the OpenScript expression to evaluate.
- ◆ *wRetType* identifies the type of the expected return value, as shown in the table below.

Return value types for *wRetType*

Return value	Data type	Return value	Data type
0	CHAR	5	DWORD
1	BYTE	6	FLOAT
2	INT	7	DOUBLE
3	WORD	8	POINTER
4	LONG	9	LPSTR (zero-terminated)

- ◆ *lpRetValue* is a FAR pointer to a memory block where the return value is stored. If the return type is STRING and *nRetValueLength* is zero, *lpRetValue* is not used.
- ◆ *nRetValueLength* gives the length of the return value memory block if *wRetType* is STRING.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

The DLL function in the following example is written in C using both the *execute* and *evaluate* forms of the Windows messages. It first determines the number of pages in a book in an instance identified by the window handle *winHandleMain*, and then constructs a command that goes to each page.

```
#include <windows.h>
typedef struct tagTBMEval {
    LPSTR    lpExpression;
    WORD     wRetType;
    LPSTR    lpRetValue;
    WORD     nRetValueLength;
} TBMEVAL;
#define EVALTYPE_WORD 3
#define COMMAND "step i from 1 to %d; go to page i; end"
void stepPages (HWND hwndMain) {
    WORD wExecute = RegisterWindowMessage("TBM_EXECUTE");
    WORD wEvaluate = RegisterWindowMessage("TBM_EVALUATE");
    WORD wPages;
    TBMEVAL Eval;
    char Command[64];
    Eval.lpExpression = "pageCount of this book";
    Eval.wRetType = EVALTYPE_WORD;
    Eval.lpRetValue = (LPVOID)&wPages;
    if (!SendMessage(hwndMain,wEvaluate,TRUE,(LONG)(LPVOID)&Eval))
        return;
    wsprintf(Command,COMMAND,wPages);
    SendMessage(hwndMain,wExecute,TRUE,(LONG)(LPSTR)Command);
}
```



How to use this online guide

Contents

- 1** Finding the information you need
- 2** Understanding OpenScript programming
- 3** Learning OpenScript basics
- 4** Using the scripting tools
- 5** Handling messages, events, objects, and properties
- 6** Creating an OpenScript statement
- 7** Using variables, values, and text
- 8** Writing common scripts
- 9** Handling user events
- 10** Managing data
- 11** Using special effects
- 12** Using dynamic data exchange
- 13** Using dynamic-link libraries
- 14** Debugging OpenScript programs



Contents

- 1** Finding the information you need
- 2** Understanding OpenScript programming
- 3** Learning OpenScript basics
- 4** Using the scripting tools
- 5** Handling messages, events, objects, and properties
- 6** Creating an OpenScript statement
- 7** Using variables, values, and text
- 8** Writing common scripts
- 9** Handling user events
- 10** Managing data
- 11** Using special effects
- 12** Using dynamic data exchange
- 13** Using dynamic-link libraries
- 14** Debugging OpenScript programs

Chapter 14

Debugging OpenScript programs

You debug a script to find and correct errors that prevent it from running, cause undesirable or unpredictable behavior, or produce incorrect results. Use the tools and the techniques presented in this chapter to build more reliable applications.

CONTENTS

About debugging a ToolBook II script	322
Responding to the Execution Suspended message	328
Using the ToolBook Debugger	330
Using scripts for debugging	342



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

About debugging a ToolBook II script

Debugging involves checking the syntax of OpenScript statements, testing statements, determining the source of errors that suspend script execution, and tracing the history of calls made in a script. Each technique described in this chapter helps you identify a specific type of error.

Debugging your application is an ongoing development task. You can reduce errors and make debugging easier by avoiding common errors in scripts, and by designing your scripts for easy testing. Because all parts of your application relate to each other, fixing one error may reveal another. Each time you correct a script or change the application, you need to test it.

Types of scripting errors

Three types of scripting errors can occur in your ToolBook II applications:

- ◆ Compile errors
- ◆ Run-time errors
- ◆ Logic errors

Compile errors occur in the syntax of an OpenScript statement and do not allow ToolBook II to compile the script. These errors are detected when you save a script and ToolBook II compiles the OpenScript code. The compiler highlights the error. If you ignore the error and carry on, your script will not run. Sometimes the compiler cannot find the error (because of a missing *end* statement, for example), and it stops at the end of the questionable handler or script.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Run-time errors occur while script is running. They are typically caused by unexpected values or actions, such as errors in the logic of a script statement. For example, a run-time error occurs if you attempt to add a number to a string. When run-time errors occur, ToolBook II suspends the script and displays the Execution Suspended message.

Using *sysSuspend* and *sysErrorNumber*, you can override the display of the ToolBook II error message and provide your own custom error handling. To do this, set the *sysSuspend* variable to *false*. After an OpenScript command executes, the system variable *sysErrorNumber* will hold an error number value greater than zero if an error occurred. Note that not checking for errors can lead to serious problems; therefore, you should turn the display of the ToolBook II error message off around only the code you intend to check yourself.

Logic errors occur when your application does not work in the way you intended, even though the syntax of the handler is correct. For example, every statement of a handler might operate perfectly, but not produce the expected results because it does not include a *forward* command.

Preventing errors

By planning, checking, and testing your scripts carefully, you can significantly reduce the number of errors in your scripts.

Plan the application carefully

Before you create an application, carefully plan how each part of it will function. Consider making a sketch of the application that shows the various objects, including the names of objects, the properties they will use, and the purpose of the handlers in their scripts. Also, consider the best placement of objects and scripts in the ToolBook II object hierarchy.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

During the planning process, you will begin to recognize where you need to define local and system variables and where you might need to create your own properties, functions, and messages.

As you plan your application, consider:

- ◆ the purpose of each object and script.
- ◆ the effect a script will have on other objects in the application.
- ◆ the properties available to each object.
- ◆ the built-in messages that ToolBook II sends and how they move up the object hierarchy.
- ◆ the variables used in a script with their data types and initial values.
- ◆ the source data that a script will use.
- ◆ the desired data that a script will produce.
- ◆ the steps required to turn the source data into the final data.

You should not duplicate handlers unnecessarily. For example, if two buttons have scripts that calculate the same value, such as *grossMargin*, it is usually more efficient to write a single *to get* handler and call it when needed, rather than write two separate handlers.

Unlike single-thread programming languages, ToolBook II is event-driven, and messages are passed up an object hierarchy. The location of a script in the hierarchy can affect the reaction of the application to the corresponding message. Handlers higher in the hierarchy can affect other handlers.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Watch for handlers that respond to the same message at different points in the object hierarchy. For example, you can place an *enterPage* handler that executes correctly in the script of the background. However, unless you forward the message from that handler, a second *enterPage* handler in the script of the book will not execute.

Design handlers for easy testing

To make handlers easier to test:

- ◆ indent control structures appropriately so your scripts are easier to read.
- ◆ plan each handler so that it performs a single task or just a few tasks. For example, one handler might get data from an Excel spreadsheet and send a message to a second handler that formats the data. A third handler might display the appropriate number of objects to graph the data.

Test handlers and scripts as you write them

You should perform incremental testing of your scripts and application. In a script or in the Command window, write statements to test and show the results of running a handler. When areas of an application do not work correctly, you can isolate the problem by breaking the scripts into sections to test for errors.

When you are creating an application, keep in mind that the systems of those who will use it might be different than your own. If a script contains more than one handler, test each handler when you finish writing it. It is easier to isolate an error by testing one handler at a time than by testing a script that contains several handlers.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Read a script carefully after you create it

When you create a script in the script editor, print it and read it carefully for spelling and typographical errors. Although ToolBook II can check the syntax of a script, it cannot verify whether you spell a variable, message, or function name correctly and consistently every time it occurs in an application. The same problem arises when you create user-defined functions, properties, and messages. ToolBook II accepts any name you provide by assuming that the name already exists in another script, or that you are creating a new name.

Also, be sure to declare and initialize system variables before referring to them in a script. Be careful not to declare a system variable as a local variable.

Check the script for logic errors

Logic errors cause incorrect or unexpected results or undesirable behavior in an application. For example, an expression might not evaluate to the correct result, or an infinite or recursive loop will prevent ToolBook II from exiting a control structure.

To check for logic errors, test statements in the Command window or step through the execution of the script on paper. To check an expression, perform the calculation manually and check your results against the results of the script. Be especially careful if you use data types to declare variables; use the correct types for the data you are working with. For example, if you assign a real number to a variable typed as an *int*, ToolBook II will truncate the decimal portion of the number.

To avoid errors such as infinite or recursive loops, make certain that each control structure can actually be exited before you attempt to run the script. If you get stuck in a loop use SHIFT+SHIFT to break to the ToolBook Debugger.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Check the syntax before compiling the script

As you write a script in the script editor, periodically choose Check Syntax from the File menu or click the Check Syntax button on the script editor toolbar. However, even when the syntax of a script is correct, it might still contain other types of errors that appear at run time.

Test the value of the *sysError* property

As ToolBook II executes a script, some commands send an error message to the *sysError* property when certain errors occur. For example, if a handler requests that ToolBook II import a file that does not exist, ToolBook II sets the *sysError* property to *No such file*.

You can get the value of the *sysError* property from a script. The following statement gets the value of *sysError* and displays it in a field:

```
text of field "Error" = sysError
```

For details about the *sysError* property, see “Checking for results and errors” in Chapter 8, “Writing common scripts.”

Test the script at Reader level

The only sure way to check a script for errors is to switch to Reader level and test it as a user. For example, to test a *mouseEnter* handler for a field, switch to Reader level and move the mouse pointer over the field.

If you run a script at Reader level and ToolBook II displays the Execution Suspended message, a run-time or logic error has occurred. ToolBook II cannot continue executing the script until you correct the error. Even if the Execution Suspended message does not appear when you execute a script, you will want to verify that the script behaves as expected and returns the correct results.



- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

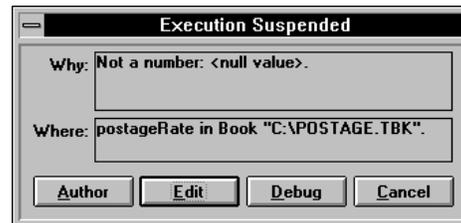
Responding to the Execution Suspended message

ToolBook II displays the Execution Suspended message when it encounters a run-time error. Typical execution errors include:

- ◆ referring to a nonexistent object.
- ◆ setting a property that cannot be set.
- ◆ setting a property to an invalid value.
- ◆ setting a typed variable to an inappropriate value.
- ◆ specifying a parameter that is outside the valid range of a function.

The Execution Suspended message dialog box contains details about the error in its Why and Where boxes. The Why box displays the reason that ToolBook II suspended script execution. The Where box shows the handler identifier for the message and the object that was handling it when ToolBook II suspended execution.

Figure 1 An error causing ToolBook II to suspend execution



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

When ToolBook II displays the Execution Suspended message, you can respond as shown in the table below.

Responses in the Execution Suspended message box

Click	To do this
Author button	Suspend further execution of the script and switch to Author level
Edit button	Suspend further execution of the script and display the script editor; the script term that caused the error is highlighted
Debug button	Display the ToolBook Debugger with the error highlighted in the script box (for details about the ToolBook Debugger, see the next section)
Cancel button	Suspend the execution of the current handler but complete the operation that triggered the suspended handler

Note To prevent ToolBook II from displaying the Execution Suspended message when it encounters an error, set *sysSuspend* to *false*. You can then check the value of *sysError* after commands that set it to determine the result of the command. However, do not leave *sysSuspend* set to *false* any longer than absolutely necessary. Subtle errors multiply when you disable ToolBook II error trapping and do not replace it with something else.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Using the ToolBook Debugger

To locate a problem with a script, use the ToolBook Debugger to view and change variables, and to trace or methodically step through the execution of a script. Use the ToolBook Debugger to:

- ◆ determine where you are in the script.
- ◆ edit the script to change how it executes.
- ◆ set breakpoints so that you can halt the script as it is evaluating expressions or running commands.
- ◆ trace the sequence of calls to handlers.
- ◆ execute a single statement or expression.
- ◆ examine and change the values of variables and arrays.
- ◆ halt execution of a script and return to Author level.

You can open the ToolBook Debugger in the following ways:



- ◆ In the script editor, choose Debug from the File menu, click the Debug button on the toolbar, or press CTRL+D.
- ◆ Set a breakpoint so that the ToolBook Debugger is called automatically when the script is running. For details, see “Setting, using, and removing breakpoints” later in this chapter.
- ◆ Send the *debugScript* message. For details, refer to the entry for *debugScript* in the OpenScript reference in online Help.
- ◆ Click the Debug button in the Execution Suspended message box.
- ◆ Press SHIFT+SHIFT while a script is running.
- ◆ Press CTRL and click the Debug button on the main window toolbar.

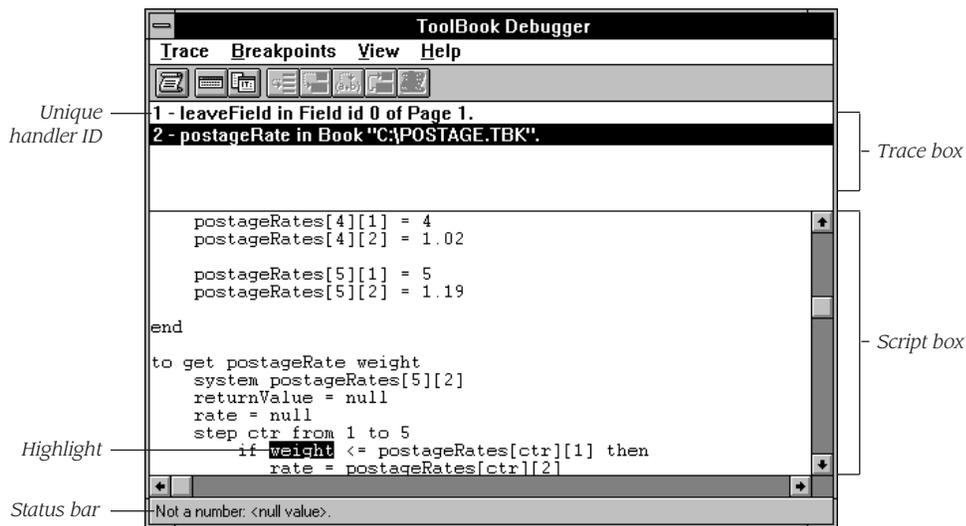


Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Note If you open the ToolBook Debugger from the script editor, you cannot trace script execution or display the Variables window. To trace script execution or to display the Variables window, you must first set a breakpoint in the ToolBook Debugger, and then exit the debugger and run the script. For details, see “Setting, using, and removing breakpoints” later in this chapter.

Figure 2 The ToolBook Debugger window



The ToolBook Debugger contains three areas that provide different types of information: the trace box, the script box, and the status bar.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

The trace box displays an identifier for each handler currently running and highlights the handler that suspended execution in the script box. If there is more than one handler running, the trace box contains a call stack, or history, of all the handlers that are running. In many cases, there is only one handler in the call stack. However, if the handler that caused the ToolBook Debugger to appear was called from another handler, or if it is a user-defined function, the call stack includes the calling handler as well.

The script box displays the script of the object that contains the currently selected handler and highlights the term at which execution stopped. When you double-click a handler identifier in the trace box, ToolBook II displays the associated script in the script box. If the Variables window is also displayed, double-clicking the unique handler identifier automatically updates the values for any displayed variables.

The ToolBook Debugger status bar contains a short description of why execution was suspended (breakpoint encountered, run-time error occurred, or SHIFT+SHIFT pressed). If you invoked the ToolBook Debugger from the Execution Suspended message box, the status bar displays the contents of the Why box from the Execution Suspended message.

Setting, using, and removing breakpoints

A breakpoint is an indicator, or marker, that you place in a script to tell ToolBook II to stop executing a script at the specified point and display the ToolBook Debugger. Breakpoints are useful for forcing a break in the execution of a script to evaluate the script up to that point. Every time the ToolBook Debugger appears, you can trace the execution of the script, check the values in the variables of the handler, and edit the script before allowing it to continue running.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Use the following methods to set and remove breakpoints:

- ◆ To set a breakpoint, open the ToolBook Debugger and click the term you want to use as a breakpoint in the script box. ToolBook II overstrikes the term with slashes (/). When the script runs next, ToolBook II halts execution when it reaches the term you have marked.
- ◆ To remove a breakpoint, open the ToolBook Debugger and click the term from which you want to remove the breakpoint in the script box. The slashes disappear and the term is no longer a breakpoint.
- ◆ To remove all breakpoints from a script, choose Clear Script Breakpoints from the Breakpoints menu.
- ◆ To remove breakpoints for all scripts in the application (even if they are not shown in the trace box), choose Clear All Breakpoints from the Breakpoints menu.

A breakpoint is valid only in the current instance in which the ToolBook Debugger is running and cannot be saved with the script. If you set a breakpoint in one instance of ToolBook II and execute the script in another instance, the executing script will not have the breakpoint. When you edit the script of an object, ToolBook II automatically removes all the breakpoints from that script. ToolBook II removes all breakpoints from all scripts in a book when you close the book.

To isolate just a portion of a script to find an error, set a breakpoint on the last term known to execute correctly before the error occurred, and then trace execution forward from the breakpoint.

Tip You can set breakpoints in scripts of unopened books using the *debugScript* message. This is useful if you want to debug the startup sequence of an application.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Tracing script execution

The ToolBook Debugger allows you to trace or step through statements and expressions in a script. When ToolBook II traces execution, it highlights each term in the script box as it is executed. By tracing the execution of a script, you can see:

- ◆ the order in which statements execute.
- ◆ how variable values change.
- ◆ how ToolBook II evaluates expressions.
- ◆ how one handler calls another handler.

By default, ToolBook II highlights and displays the handler that was running when execution stopped, but you can select any handler listed in the trace box to display it. The Trace menu in the Toolbook Debugger includes the options for tracing execution shown in the table below.

Options for tracing script execution

Trace menu command	Shortcut	Description
Trace Statement	CTRL+S	Traces execution of a statement and pauses after tracing that statement (Trace Statement is the equivalent of the Step Over command in other debugging products.)
Trace Call	CTRL+C	Traces a call; when a handler calls another handler, pauses at the start of the called handler (Trace Call is the equivalent of the Step Into command in other debugging products and is useful for debugging subroutines.)

(continued)



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Options for tracing script execution, *continued*

Trace menu command	Shortcut	Description
Trace Expression	CTRL+E	Traces execution of a statement incrementally so you can verify an evaluation order of an expression
Trace Return	CTRL+R	Finishes the execution of the called handler, and then pauses at the next statement in the original calling handler (Trace Return is the equivalent of the Until Return command in other debugging products.)
Continue Execution	CTRL+G	Closes the Toolbook Debugger and continues executing the script until the next breakpoint is encountered (Continue Execution is the equivalent of the Run command in other debugging products.)

Displaying and modifying variables

Incorrect results in a script are often caused by incorrect variable values. If you use an incorrect value more than once in an application, or if other variables are based on it, that value causes other values to be incorrect as well.

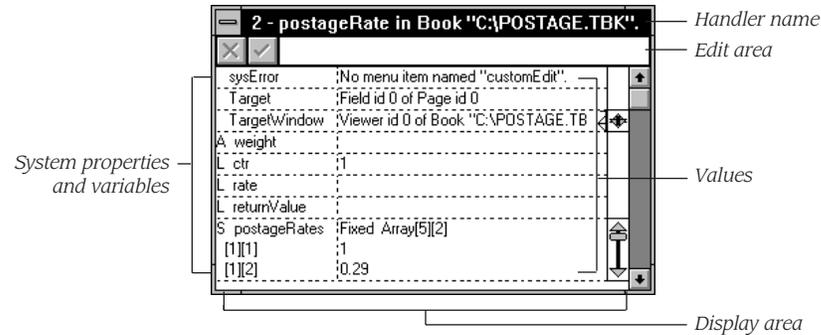
To display or modify the value of variables, display the Variables window while tracing script execution in the ToolBook Debugger. The Variables window shows the types, names, and values of all variables; the arguments referenced in the currently executing handler; and the values of the properties *sysError*, *target*, and *targetWindow*.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Figure 3 The Variables window



► **To open the Variables window:**

- 1 Open the ToolBook Debugger.
- 2 Choose Variables from the View menu, or press CTRL+F3.

The Variables window displays the variables for the handler that is selected in the trace box. To display the variables for a different handler, click its handler identifier in the trace box.

Viewing variables

The display area of the Variables window shows you the name, scope, and value of all the variables in the current handler.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Figure 4 The Variables window display area

Selected system properties	sysError	'No menu item named "customEdit".'
	Target	'Field id 0 of Page id 0'
	TargetWindow	'Viewer id 0 of Book "C:\POSTAGE.TB"'
A=Argument	weight	
L=Local variable	L ctr	1
S=System variable	L rate	
	L returnValue	
	S postageRates	Fixed Array(5)(2)
Array elements	[1][1]	1
	[1][2]	0.29

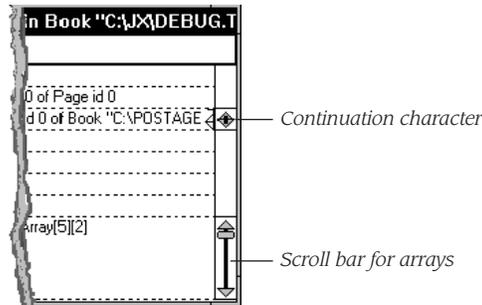
If the value of a variable is too long to fit into the display area, ToolBook II displays a continuation character to indicate this. If the variable is an array, ToolBook II displays a small scroll bar that allows you to scroll through the elements of the array. To show more of the variable, resize the Variables window, or drag the vertical bars between parts of the display area to make more room. You can also double-click the value of the variable to display a separate window for that value.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Figure 5 ToolBook II marks long values and arrays with special symbols



Changing the values of variables

You can use the Variables window to change the value of a variable. Your change is effective immediately for the current handler. For example, you can change the value of a variable if ToolBook II suspends the current handler because the value of the variable is *null* or because a statement in the current handler attempts to assign an invalid value to the variable.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

► **To change the value of a variable:**

- 1 In the Variables window, click the value of the variable you want to change. If the variable is an array, use the scroll bar for that array to display the value you want to change and then click the value.

ToolBook II displays the current value of the variable in the edit area of the Variables window.



- 2 Enter a new value and click the checkmark to save the change, or click the X to cancel the change.



Displaying and editing system variables

To see and edit a list of all the system variables declared in the current instance of the book, choose System Variables from the View menu. ToolBook II displays a window similar to the Variables window, but which shows only the current system variables. You can view and edit the values of system variables using the same techniques you use in the Variables window. For details, see “Displaying and modifying variables” earlier in this chapter.

Displaying and editing user properties

To see and edit the user properties of the object whose script is running, choose User Properties from the View menu in the Toolbook Debugger. ToolBook II displays a window similar to the Variables window, listing the user properties for the object and their current values. You can view and edit the values in this window using the same techniques you use in the Variables window. For details, see “Displaying and modifying variables” earlier in this chapter.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

The User Properties window displays the user properties for the object whose handler is selected in the trace box. To display the user properties for a different object, click its handler identifier in the trace box.

Checking the values of properties

You can check the value of a property by choosing Command from the View menu and testing the value there. You cannot directly view the value of a property of an object in the Toolbook Debugger.

Using the Command window with the ToolBook Debugger

You can display the Command window while in the ToolBook Debugger by choosing Command from the View menu or by clicking the Command Window button on the ToolBook Debugger toolbar. The Command window is very helpful for getting and setting:

- ◆ system properties, such as *activeWindowHandle*, *focus*, *focusWindow*, *printerConditions*, *selection*, *sysErrorNumber*, and *sysTimeFormat*.
- ◆ object properties, such as *header of this book*, *name of focus*, *name of this page*.
- ◆ user properties.

You cannot use the Command window to:

- ◆ change local variables.
- ◆ get the value of *target* (*target* is always the current page for the Command window).
- ◆ set *It* in the context of the handler being debugged (*It* is local to the handler being debugged).



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

While using the Command window in the ToolBook Debugger, keep in mind the following:

- ◆ Changes to system variables are updated after the next statement is executed, not immediately.
- ◆ To change a system variable, you must declare it in the context of the Command window. That means `svLevel = 4`, where `svLevel` is a system variable, does nothing to the system variable. However, `system svLevel; svLevel = 4` does change the value of the system variable.

Important Proceed carefully while using the Command window in the ToolBook Debugger as you can perform tasks that are potentially destabilizing. For example, deleting the object whose script is being debugged can cause errors and possibly change values in unexpected ways.

Debugging a Command window script

To debug a Command window script, place it into a handler in the script of an object and debug it from there. The ToolBook Debugger is unavailable because the text in the Command window appears as one line of script to ToolBook II. The debugger cannot break this combined line into its components. Pressing RETURN or ENTER for the Command window executes the command `execute commandWindow`.



- 1** Finding the information you need
- 2** Understanding OpenScript programming
- 3** Learning OpenScript basics
- 4** Using the scripting tools
- 5** Handling messages, events, objects, and properties
- 6** Creating an OpenScript statement
- 7** Using variables, values, and text
- 8** Writing common scripts
- 9** Handling user events
- 10** Managing data
- 11** Using special effects
- 12** Using dynamic data exchange
- 13** Using dynamic-link libraries
- 14** Debugging OpenScript programs

Using scripts for debugging

Instead of using the ToolBook Debugger to trace the execution of scripts, you can use OpenScript commands and functions to perform similar actions. For example, you can include statements in your scripts that display the values of variables in the Command window or status bar, or that write error information to a file.

Using OpenScript commands to debug scripts gives you more precise control over which values and statements you are debugging than the ToolBook Debugger does. In addition, you can use OpenScript commands to trace execution and display values without stopping and calling the debugger; this is particularly useful when you are processing continuous user-interface actions.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

The OpenScript commands listed in the following table are useful for debugging.

OpenScript commands for debugging

OpenScript term	Use to	Example
<i>request</i>	Display values	<pre>request userProperties of \ field "notes" request It request my fillColor</pre>
<i>beep</i>	Alert user to a problem	<pre>if xPos < 0 then beep 2 request "You have reached" & \ "the left margin." end</pre>
<i>pause</i>	Allow time to call the ToolBook Debugger	<pre>if xPos < 0 then request "Error condition" & \ "detected." pause 3 seconds end</pre>
<i>sysCursor</i>	Alert user to current status	<pre>to handle searchBook sysCursor = 4 ... --Further statements here sysCursor = 1 end</pre>



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Creating a function for debugging

When you are testing an application and need a quick look at whether it is executing correctly, it can be helpful to create a user-defined function that displays values in the Command window or field. For example:

```
--Displays values in the Command window  
to get debugCommandWindow fObjRef, fData  
    put "Debug in " & fObjRef & ":" & fData  
    return null  
end
```

```
--Displays values in a field  
to get debugFieldDisplay fObjRef, fData  
    text of field "debug"="Debug in " & fObjRef & ":" & fData  
    return null  
end
```

You can then build a debugging function call into your scripts to display values:

```
step i from 1 to 1000  
    vPos = myFunc(i)  
    --Additional statement calls the function  
    get debugCommandWindow(self, i && vPos)  
end
```

When you want to disable debugging, simply remove or rename the function and your scripts will run normally.

Note Before releasing a ToolBook II application as a final product, remove the calls to user-defined debugging functions to achieve maximum performance.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Using the status bar for debugging

You can use the status bar to display information while a script is running. Using the status bar is convenient if you do not want to display the Command window in the middle of the screen. However, the status bar is limited to displaying about 60 characters, so it might not be useful if you want to display longer strings of text.

To use the status bar, make sure that it appears when the script is running. You can display information in the status bar by setting its *caption* property. The following example illustrates a handler that displays the unique name of the object under the mouse pointer while the mouse button is pressed:

```
to handle buttonStillDown loc
  readerStatusBar of this viewer = true
  system vCurrentObject
  objPoint = objectFromPoint(loc)
  if objPoint is not vCurrentObject then
    caption of statusBar = objPoint
    vCurrentObject = objPoint
  end if
end buttonStillDown
```

You can use the status bar to display data in the same way you use the Command window. For example, you can display debugging information in the status bar instead of the Command window:

```
to get debugCommandWindow fObjRef, fData
  in viewer mainWindow
    caption of statusBar = "Debug in " & fObjRef & ": " & fData
  end
  return null
end
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Using a trace file to record errors

You might not always be able to monitor the execution of your script, but you can track problems that occur in your application while it is being used elsewhere. To keep track of areas in the application that need debugging, write your traces to a file. For details about tracing, see “Tracing script execution” earlier in this chapter.

Writing traces to a file is similar to displaying them in the Command window, except that the messages that would appear in the ToolBook Debugger trace box are written to a file that you can examine later. To create a trace file, first open or create a text file to store the error information. A good place to create this file is in an *enterBook* handler in the book script or in a system book script. For example:

```
to handle enterBook
    system svDebugFile
    svDebugFile = "debug.txt"
    clear sysError
    openFile svDebugFile
    if sysError is not null
        clear sysError
        createFile svDebugFile
    end if
    closeFile svDebugFile
end enterBook
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

You can write to this file at any time to record an activity while the book is running. One easy way to do this is to write a user-defined function that updates the file. You can call this function as part of any script, and have your script pass it the error information that you want to record. The following example contains the script for a function to update the trace file.

```
--This handler creates entries in a trace file; place this
--user-defined function in the script of the book you are debugging
--or in a system book
to get debugTraceFile fErrText
    system svDebugFile
    openFile svDebugFile
    vLine = CRLF & sysDate && sysTime && fErrText
    writeFile vLine to svDebugFile
    closeFile svDebugFile
    return null
end
```

You can call the *debugTraceFile* function as you called the Command window in the example illustrated under “Creating a function for debugging” earlier in this chapter. For example:

```
to handle buttonClick
    clear sysError
    fileName = "\data.txt"
    openFile fileName
    if sysError is not null then
        --If error detected, records the error in the trace file
        get debugTraceFile( "Error opening" && fileName && \
            "from buttonClick handler of" && self )
    end if
end buttonClick
```



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Note You should not use a trace file after debugging an application. If you do, you might build up large files, and the file manipulation will slow application performance considerably. You can use the Tb70.sbk search-and-replace utility to remove the *debugTraceFile* calls from all of your scripts.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Index

Symbols

- operator 106
& operator 111
&& operator 111
* operator 106
+ operator 106
.dll file extension 287
.drv file extension 287
.exe file extension 287
.sbk file extension 173
/ operator 106
< operator 107
<= operator 107
<> operator 108
<command> parameter 183
<command> parameter 281
<destination object> parameter 253
<dll name> parameter 287
<dllType> parameter 303
<isCtrl> parameter 202
<isShift> parameter 202
<keyCode> parameter 202
 values for 205–206
<location> parameter 196
<wait> parameter 268
<winHandle> parameter 303
<winMsg> parameter 303
= operator 37, 107
> operator 107
>= operator 108
^ operator 106

A

Access keys, defining 210
after string specifier 144
Aliasing function names 288–289
Allow Drag option 252
allowDrag message 255
allowDrop message 255
and operator 108
Animating objects 234–248
Animations
 controlling speed 248
 creating by moving objects 236
 creating by showing/hiding 243
 creating handlers for 244
 creating smoother 248
 initializing 247
 optimizing 242
ansiToChar() function 146
API functions 284
 calling 300
Applications
 about initializing 181–182
 planning 323–325
 saving changes to 170–172
argCount reserved variable 122
argList reserved variable 122
argument operator 114
Arithmetic operators 106
Array parameters 101



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Arrays
 about 127
 and text 130
 and text operators 130
 assigning as user properties 134
 declaring 129, 131
 fixed vs. dynamic 128
 getting dimensions 132
 indicating in called handler 133
 passing as parameters 132
 passing by reference 134
 setting and getting values 129
 sizing 131
as clause 217
ask command 40
Assignment operator (=) 37
Asymetrix (click2learn) FTP site 13
Asymetrix (click2learn) Web site 13
Author and Reader levels, dragging
 between 263
Author button 329
Author level, initializing 185–186
author message 186

B

background data type 124
BASIC, compared to OpenScript 20
beep command 267, 343
before string specifier 144
beginDrag message 255
Binary value 139
bitAnd operator 113
bitNot operator 113
bitOr operator 113
bitShiftLeft operator 113
bitShiftRight operator 113
Bitwise operators 112

bitXor operator 113
book data type 124
Books
 adding a system book to 174–176
 prompting users to save 172
 saving when closed 171–172
BOOL data type 290, 291
Boolean values 137, 291
bounds property (main window) 187
bounds property (object) 237
break statement 97–98
break to system command,
 and execute statements 167
break to system statement 98
Breakpoints 332–333
 defined 332
 setting and removing 333
Built-in messages
 changing or disabling 71
 defined 62
 sending 62
 tips for placing 67–69
 types 63–64
Button-click events 244
buttonClick message 27, 191
buttonDoubleClick message 191
buttonDown message 191
Buttons
 and searches 229
 defining access key for 210
buttonStillDown message 191, 192
buttonUp message 191
by reference special term 134
BYTE data type 290, 297
 return value for *wRetType* 318



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

C

- C data types, ToolBook II translations for 290
- C language, compared to OpenScript 20
- Called handler 73, 134
 - indicating an array 133
- Calling handler 73, 134
- Calling OpenScript from a DLL 314
- Cancel button 329
- caption* property 42, 210
- caretLocation* property 198
- CHAR data type, return value for *wRetType* 318
- CHAR FAR * data type 291, 294
- char(s)* operator 111
- character(s)* operator 111
- Characters
 - checking user entries 220
 - defining custom behavior 210
- charCount()* function 146
- charToAnsi()* function 146
- closeFile* command 223
- closeRemote* command 278
- Color constants, referring to 151
- color* data type 124
- Colors, referring to 149
- colorTray* system object 75
- Combo boxes, and searches 228
- Comma
 - deleting from number 216
 - using as delimiter 217
- Command option 340
- Command window
 - and the ToolBook Debugger 340–341
 - debugging scripts 59
 - displaying 55
 - history area 58
 - opening 28
 - overview 18
 - running scripts 55
 - testing statements in 57
 - using to display information 40
 - using to enter scripts 27
 - using variables in 56
 - working in 54
- Command window script, debugging 341
- Commands
 - overview 90
 - vs. messages 91
- commandWindow* system object 75
- Compile errors 322
- Conditions
 - commands for establishing 92
 - looping and testing 94–95
- conditions* command 92
- Constant strings, passing 292
- Constants. *See also* Keystroke constants
 - defined 105, 140
 - types of 140
- Containers, defined 105
- contains* operator 108
- Continue Execution command 335
- continue* statement 99



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

- Control structures
 - creating 93
 - defined 92
 - interrupting 97
 - nesting 97
 - types of 92–93
 - using 92–99
- createFile* command 223, 224
- CRLF constant 144
- Cursor. *See* Mouse pointer and Insertion point
- Curves, animating 239
- D**
- Data
 - formatting 214–218
 - formatting for calculating 215–217
 - formatting for comparisons 217–218
 - formatting for output 214
 - validating 218–222
- Data types
 - and literal values 135
 - choosing 126
 - converting to strings 143
 - declaring 74
 - declaring for parameters 127
 - declaring for system variables 126
 - defined 123
 - described 124
 - for DLL function parameters 290
 - for values passed by reference 297
 - values for variables with 125
- date* data type 124, 215
- Dates. *See also* Data
 - checking user entries 219

- DDE
 - about 270–271
 - messages for 271
 - tips for using 279
 - using with ToolBook II 271–272
- DDE channel, keeping open 277–278
- DDE client, ToolBook II as 272–279
- DDE commands
 - checking before sending 276
 - checking for errors 276–277
 - sysError* values for 276–277
- DDE conversations 279
- DDE requests
 - controlling the response to 280
 - defining custom response to 281
- DDE server application
 - exchanging data 274
 - executing commands 275
 - opening 273
- DDE-event handlers, extending 281–282
- DDE-event messages 64
 - placing handlers for 281
- Debug button 329
- Debug command 53
- Debugger. *See* ToolBook Debugger
- Debugging
 - creating functions for 344
 - using scripts for 342
 - using the status bar for 345
- Debugging Command window
 - scripts 341
- Debugging scripts, about 322–327
- debugScript* message 330
- debugTraceFile* function 347
- defaultAllowDrag* property 252, 254
- defaultAllowDrop* property 252, 254



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

- Destination object
 - conditional 256–258
 - defined 249
- Development tools 17
- Dialog boxes, creating 42
- dimensions* function 132
- displayAspect()* function 138
- div* operator 106
- DLL calls, checking return values 286
- DLL function names, aliasing 288–289
- DLL functions
 - and VGA screens 187
 - linking and declaring 287–294
 - linking in a system book 179
 - specifying data types for 290–292
- DLL modules 284
- DLLfunctions()* function 287
- DLLs
 - about 284–285
 - calling OpenScript from 314
 - commands for linking 93
 - declaring string parameters 309
 - included with ToolBook II 284
 - linking to 97
 - process for calling 285–287
 - returning strings 310–314
 - tips for placing and linking 289
 - verifying links 287
 - writing 309–319
- do/until* command 92
- DOUBLE data type 290, 297
 - return value for *wRetType* 318
- Drag & Drop dialog box 252
- drag* command 254
 - using 258–259
- Drag images, changing 262–263
- Drag operation, testing 262
- Drag-and-drop behavior
 - about 249–251
 - adding 249–266
 - commands 254
 - constraining 259
 - defining 251
 - defining for fields and boxes 263–264
 - event messages 255
 - forcing 259–260
 - properties 254
 - uses for 250
- Drag-and-drop message 64
- Drag-and-drop messages, order sent 261
- Dragging groups 265
- Dragging objects in groups 264
- dragImage* property 254
- DWORD data type 297
 - return value for *wRetType* 318
- dword* data type 124
- Dynamic arrays 128
- Dynamic data exchange. *See* DDE
- Dynamic-link libraries. *See* DLLs and DLL

E

- Edit button 329
- edit script of <object>* statement 46, 47
- endDrag* handler 253
- endDrag* message 253, 255
- Enter events 244
- Enter-event messages 156
 - and system books 178
 - for initialization handlers 182–184
- Enter/leave-event message 63
- enterApplication* handler, streamlining 185



Contents

- 1** Finding the information you need
 - 2** Understanding OpenScript programming
 - 3** Learning OpenScript basics
 - 4** Using the scripting tools
 - 5** Handling messages, events, objects, and properties
 - 6** Creating an OpenScript statement
 - 7** Using variables, values, and text
 - 8** Writing common scripts
 - 9** Handling user events
 - 10** Managing data
 - 11** Using special effects
 - 12** Using dynamic data exchange
 - 13** Using dynamic-link libraries
 - 14** Debugging OpenScript programs
- enterApplication* message
 - using 183–184
 - vs. *enterBook* message 184*enterBackground* message 156
- enterBook* handler, streamlining 66
- enterBook* message, using 184
- enterButton* message 156
- enterComboBox* message 156
- enterDrop* message 255
- enterField* message 156
- enterPage* message 156
- enterSystem* message, using 182–183
- Error handling, providing your own 323
- Errors
 - and DDE commands 276
 - and execute statements 167
 - and *leaveField* handlers 222
 - checking for in scripts 163
 - compile 322
 - handling in scripts 164
 - hiding from users 221
 - informing users of 221
 - logic 323
 - preventing 323–327
 - preventing common 59
 - run-time 323
 - scope of values 164
 - scripting 322–323
 - using a trace file to record 346–348
 - validating user entries 220
- evaluate function 168
- Event-focus properties 76
- Event-messages, for drag-and-drop behavior 255
- Events for triggering animation 244
- execute commandWindow* command 341
- execute* statement 166–167
- executeRemote* command 272–279

- Execution Suspended message
 - displaying 59
 - responding to 328–329
 - suppressing 164–165
- Execution Suspended message dialog box 328
- Explicit references to objects 35
- Explicit statements, and navigation 161
- Expressions. *See also* Operators
 - and string operators 142
 - creating with operators 104–116
 - defined 104
 - evaluating 168
 - expressions with operators, defined 105
 - expressions with strings, defined 105

F

- Failed: Busy *sysError* value 276
- Failed: Denied *sysError* value 277, 281
- Failed: Interrupted *sysError* value 277
- Failed: Memory Error *sysError* value 277
- Failed: No Server *sysError* value 277
- field* data type 124
- Files
 - commands for reading/writing 223
 - creating new 224
 - moving to set positions 227
 - reading from 224–225
 - writing to 226
- Fixed arrays 128
- flip* command 160, 240
- FLOAT data type 290, 297
 - return value for *wRetType* 318
- flushMessageQueue()* function 212
- format* command 214
- forward* command 31, 69
- forward to system* statement 68
- from* clause 214



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

FTP site, click2learn.com 13
Functions. *See also* User-defined functions
 about 100
 creating for debugging 344
 defined 105
 used with strings 146
fxDissolve command 162
fxWipe command 162
fxZoom command 162

G

GDI DLL module 284
General system properties 76
get command 38
getRemote command 272–279
graphic data type 124
Groups
 dragging 265
 dragging objects in 264

H

HANDLE data type 290
Handlers 84. *See also* Notify handlers and arrays 128
 and keyboard-event messages 203
 and parameters 39
 calling vs. called 70
 creating for animations 244
 creating libraries of 173–180
 creating simple 26
 defined 25
 designing for easy testing 325
 in system books 175
 placing for built-in messages 67
 types of 38
 writing for initializing 182–184
 writing for system books 178–179

HBMP data type 290
HDC data type 290
Help, accessing for ToolBook II 13
Hexadecimal value 139
HLS values, specifying 150
Hotwords, using to display pages 155
HWND data type 290
hWnd parameter 316
hwnd Windows parameter 305

I

Idle event 244
idle handler, and animations 245–246
if statement 217
if/then/else command 92
Implicit references to objects 35
Implicit statements, and navigation 161
in <viewer> command 93
Information
 displaying for current page 41
 prompting users for 40
Initialization, writing handlers for 182–184
Initializing applications, about 181–182
Initializing Reader and Author levels 185–186
Insertion point
 determining location 198–199
 moving with script editor 48
 setting location 198–199
Inst60.ini 177
INT data type 297
 return value for *wRetType* 318
int data type 124, 290
International system properties 76
into string specifier 144
is in operator 109
is not in operator 109
is not operator 108



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

is operator 107
isObject function 219
isType function 219
isType() function 125
It reserved variable 122
item keyword 147
item operator 111
itemCount function 240
itemCount() function 146
items operator 111

K

keepRemote command 278
KERNEL DLL module 284
keyA, *keyB* constant 141
Keyboard activity, emulating 208–209
Keyboard events
 about 202–203
 handling 201–212
Keyboard-event messages 63
 and *<keyCode>* parameter 205–206
 choosing which to handle 204
 key sequence sent 207
 placing handlers 203
 sending from scripts 208–209
keyChar message 202
keyDown message 202
keyF1 constant 141
keyHome constant 141
keyLeftButton constant 141
keyMnemonic message 210
keyNumLock constant 141
keyNumpad1 constant 141
keyState() function 141, 200, 208
Keystroke constants. *See also* Constants
 about 141
 examples of 141

Keystrokes
 capturing 32
 checking for 208
 discarding pending 212
keyTab constant 141
keyUp message 202
keyUpArrow constant 141

L

layer data type 124
Leave-event messages 156
 and system books 178
leaveBackground message 156
leaveButton message 156
leaveComboBox message 156
leaveDrop message 255
leaveField message 156
leavePage message 156
Left mouse button, determining
 state 200
Library.tbk 264
lineEndsPalette system object 75
linePalette system object 75
Lines, animating 239
linkDLL command 93
linkDLL control structure 285, 287
linkDLL statements, tips for placing
 and linking 289
Linking to DLLs 97
Linking to DLLs. *See also* DLLs
Links, verifying for DLLs 287
linkSysBook notification message 176
List boxes, and searches 228
Lists
 about 147
 defined 147
 producing and manipulating 143



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

- Literal values
 - assigning to variables 135
 - defined 105, 134
 - working with 134–139
 - local* statement 120
 - Local variables
 - creating 120
 - using 120
 - locateOnly* statement 230
 - lockScreen* property 156, 241
 - Logic errors 323
 - logical* data type 124
 - Logical operators 107–109
 - Logical values 137
 - LONG data type 297
 - return value for *wRetType* 318
 - long* data type 124, 290
 - Looping, commands for 92
 - Loops, and execute statements 167
 - lowercase()* function 146
 - lParam* parameter 316, 317
 - lParam* Windows parameter 305
 - LPCSTR data type 291
 - lpExpression* FAR pointer 318
 - lphi* Windows parameter 305
 - lplo* Windows parameter 305
 - lpRetVal* FAR pointer 318
 - LPSTR data type 291, 294
 - return value for *wRetType* 318
- M**
- Main window
 - changing size and position 187
 - fitting in page 187
 - Media control interface driver 267
 - Menu accelerator key 203
 - Menu, defining access key for 210
 - Menu item, defining access key for 210

- Menu-event message 63
- Message names, sending as
 - variables 70
- Messages. *See also* Built-in messages, User-defined messages, and Windows messages
 - about 62–74
 - and keyboard events 202–212
 - and the object hierarchy 30
 - commands for handling 92
 - forwarding 31
 - handling translated 305–306
 - leave and enter event 156
 - sending from scripts 69
 - sending variable contents 169
 - translating Windows 300–308
 - trapping 71–72
 - using parameters with 72
 - vs. commands 91
- Microsoft Excel 272
- Mnemonic access characters.
 - See* Access keys
- mod* operator 106
- Mouse button, determining state 200
- Mouse clicks, special behavior
 - for 199–200
- Mouse events
 - about 190–201
 - discarding pending 212
 - handling 190–201
- Mouse pointer
 - changing the shape of 200–201
 - determining objects under 196
 - determining position of 195–196
- Mouse-event messages 63, 190
 - handling 191–192
 - parameters for 193
- Mouse-pointer position, as user input 195



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

mouseEnter message 191, 197
mouseLeave message 191, 197
mousePosition window property 195
move command 138, 236
moveFile() function 286
Multiline statements, creating 57
my special term 36

N

Navigation
 and last page 154
 and *sysHistory* 158
 creating flexible 155
 designing 154
 restricting 159, 161
noDropImage property 254
not operator 109
Notification message 64
Notify handlers
 defined 84
 guidelines for using 87–88
 initializing objects 157
 keywords for 88
 types of 86
notifyAfter handler 38, 86
notifyAfterMessages property 88
notifyBefore handler 38, 86
notifyBeforeMessages property 88
notifyObjects property 88
nRetValLength word 318
Numbers. *See* Data
Numeric values
 prefixes for specifying 139
 specifying 136

O

object data type 124
Object hierarchy
 and system books 174
 defined 28
 overview 28–33
Object references
 implicit vs. explicit 35
 to current object 36
objectDropped message 255
objectFromPoint() function 197
Objects. *See also* System objects
 and scripts 25
 animating 234–248
 changing shape to animate 237
 defined 24
 defining behavior 67–68
 drag-and-drop properties 256
 explicit references to 35
 handling mouse events 190–201
 implicit references to 35
 initializing 157
 moving to animate 236
 naming 59
 referring to in scripts 34–36
 setting script properties 170
 specifying bounds 138
 specifying location 138
 vs. system objects 75
Octal value 139
offset() function 146
OK *sysError* value 276
Online Help, accessing for
 ToolBook II 13
openFile command 223



Contents

- 1** Finding the information you need
- 2** Understanding OpenScript programming
- 3** Learning OpenScript basics
- 4** Using the scripting tools
- 5** Handling messages, events, objects, and properties
- 6** Creating an OpenScript statement
- 7** Using variables, values, and text
- 8** Writing common scripts
- 9** Handling user events
- 10** Managing data
- 11** Using special effects
- 12** Using dynamic data exchange
- 13** Using dynamic-link libraries
- 14** Debugging OpenScript programs

- OpenScript
 - about 16
 - calling from a DLL 314
 - compared to other languages 20
 - development tools 17
 - finding information on 10
 - fundamental concepts 23–42
- OpenScript commands, using to Debug scripts 342–343
- Operations, data types for 126
- Operator precedence
 - changing 115
 - examples of 116
- Operators
 - creating expressions with 104–116
 - for arithmetic expressions 106
 - for bitwise operations 113
 - for logical comparisons 107–109
 - for string evaluations 111
 - precedence of 115–116
 - string 142
 - types of 106–114
- or operator 108
- P**
 - page* data type 124
 - Page size, changing 187
 - Page units, specifying 138
 - Pages
 - flipping automatically 160
 - flipping to create animations 240
 - initializing 155–157
 - navigating to 154–161
 - preloading into memory 161, 241
 - searching 228–230
 - searching without changing 230
 - sorting 231
 - Palette shift 242
 - Parameter variables, placing and separating 72
 - Parameters. *See also* Windows parameters
 - and handlers 39
 - called vs. calling handlers 73
 - declaring data types for 74, 127
 - for mouse-event messages 193
 - naming variables for 73
 - passing as arrays 132
 - passing strings as 292–294
 - referring to 74
 - using 100
 - Parent object, defined 30
 - Pascal, compared to OpenScript 20
 - Passing strings as parameters 292–294
 - patternPalette* system object 75
 - pause* command 160, 343
 - pause* statement 248
 - Pixels, specifying 138
 - playSound()* function 267–268
 - point* data type 124
 - POINTER 293
 - POINTER data type 291, 297
 - return value for *wRetType* 318
 - pointer<type>()* example 298–299
 - pointer<type>()* functions
 - accessing memory with 296–299
 - syntax for 296
 - pointerByte()* function 296
 - pointerDouble()* function 296
 - pointerDword()* function 296
 - pointerFloat()* function 296
 - pointerInt()* function 296
 - pointerLong()* function 296
 - pointerPointer()* function 296



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

pointerString() function 294, 296
pointerWord() function 296
polygonPalette system object 75
pop command 148
position property (main window) 187
Preventing errors 323
print command 96
print eject command 96
Printer system properties 76
Properties, *See also* User properties
 checking the values of 340
 commands for creating 92
 creating using scripts 79
 getting values of 37
 setting values of 37
push command 148
put command 37, 55
put statement 40

Q

Query message 64

R

Reader and Author levels, dragging
 between 263
Reader level
 adding scripts at 166
 displaying shortcut menus 194
 initializing 185–186
reader message 185–186
readFile command 223, 224
real data type 124
registerWindowMessage()
 API function 315
remoteCommand handler 280–282
remoteGet handler 280–282
remoteSet handler 280–282

request command 41, 343
respondRemote statement 281
return statement 102
RGB values, specifying 149
Right mouse button, determining
 state 200
rightButtonDoubleClick message 191
rightButtonDown message 191, 194
rightButtonUp message 191
round() function 136
Run-time errors, Execution Suspended
 message 328

S

save as command 171
save command 170
Save current changes? message 172
saveOnClose property 171
 values for 172
Saving changes explicitly 170
Saving changes to applications 170–172
Saving when closing books 171
Script box 332
Script editor
 moving the insertion point 48
 opening 46
 overview 18
 saving scripts 52
 selecting text 49
 toolbar buttons 45
 using 44
 working with text 50
Script editor font, changing 52
Script editor status bar, hiding and
 showing 52
Script editor toolbar, hiding and
 showing 52



Contents

- 1** Finding the information you need
 - 2** Understanding OpenScript programming
 - 3** Learning OpenScript basics
 - 4** Using the scripting tools
 - 5** Handling messages, events, objects, and properties
 - 6** Creating an OpenScript statement
 - 7** Using variables, values, and text
 - 8** Writing common scripts
 - 9** Handling user events
 - 10** Managing data
 - 11** Using special effects
 - 12** Using dynamic data exchange
 - 13** Using dynamic-link libraries
 - 14** Debugging OpenScript programs
- Script editor windows
 - about 46
 - moving between 46Script execution, tracing 334–335
script property 25
Scripting, about 24
Scripting errors, types of 322–323
Scripts
 - about debugging 322–327
 - adding at Reader level 166
 - and keyboard-event messages 208–209
 - checking for errors 163
 - debugging in the Command window 59, 341
 - editing multiple 46
 - guidelines for writing 47
 - handling errors in 164
 - modifying at Reader level 47
 - preventing errors 59, 323–327
 - referring to objects 34–36
 - running in Command window 55
 - saving 52
 - sending messages from 69
 - using for debugging 342
 - using to create properties 79
 - writing 24
 - writing common 153*search again* statement 229
search command 228
search for searchText statement 229
Search/replace utility 348
Searches, speeding up 229
Searching pages 228–230
Searching without changing pages 230
seconds format 216
seekFile command 223, 227

- select* command 145
- selectedText* property 145
- selectedTextlines* property 146
- selectedTextState* property 145
- selection* special term 36
- self* special term 36
- Self-contained objects, creating 84–88
- send <tbkMsg> [to <tbkObj>]* statement 303
- send back* command 159
- send* command 62, 69
- send ReportInfo* message 314
- sendMessage()* function, syntax for 315
- sendNotifyAfter* command 88
- sendNotifyBefore* command 88
- set* command 37
- setCurrentDirectory()* function 286
- setMenuItemName()* function 210
- setMenuName()* function 210
- setRemote* command 272–279
- Shared scripts, and script editor windows 46
- Shortcut menus, displaying at Reader level 194
- show commandWindow* statement 55
- silently* keyword 260
- size* property (book) 187
- sizeToPage* message 187
- skipNavigation* property 161
 - and the transition command 162
- sort* command 231
- Sort Pages command 231
- Sorting pages 231
- Sound effects, creating 267
- Source object
 - conditional 256–257
 - defined 249



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Spooling, commands for 93, 96
stack data type 124
Stack. *See* Lists
Standard message 64
start spooler command 93, 97
Startup properties, setting 188
Startup system properties 76
startupBook property 188
startupReaderRightClick property 188
Statements
 checking for errors 163
 checking for keystrokes 208
 commands for creating 90
 constructing at run time 166–167
Status bar
 displaying keys in 202
 displaying messages in 196
 in the ToolBook Debugger 332
 setting captions for 42
 using for debugging 345
statusBar system object 75
statusBox system object 75
statusControls system object 75
statusIndicators system object 75
step command 92
stillOverDrop message 255
STRING 292
String buffers, passing 293
STRING data type 297
string data type 124
String data types for DLL function parameters 291
String operators 110
 using in expressions 142
String specifiers, described 144
String values, specifying 137

Strings
 about 142
 as numbers in expressions 143
 as return values 294
 converting from other data types 143
 declaring in DLLs 309
 identifying a range in 143
 inserting/replacing text in 144
 passing as parameters 292–294
 producing lists 143
 returning in DLLs 310–314
 used with functions 146
svFirstTime variable 241
Syntax, checking for scripts 53
sysBooks property 174
sysChangesDB property 172
sysCurrency character 215
sysCursor command 343
sysCursor property 200
sysDate property 215
sysDateFormat property 215
sysDecimal character 215
sysError property, validating user entries 220
sysError values, for DDE
 commands 276–277
sysError variable 163–165
sysErrorNumber variable 163–165
sysHistory property 158
sysHistoryRecord property 159
sysLinkedDLLs system property 287
sysLockScreen property 156, 241
sysPageUnitsPerPixel property 138
sysReaderRightClick property 194
sysSuspend property 59
sysSuspend variable 165



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

sysSystemVariables property 121

System books

- adding to current book 174–176
- creating 173
- defined 173
- defining default 177
- loading when starting 176
- performance tips 179–180
- removing 178
- storing handlers in 173–180
- writing handlers for 178–179

System object type names 75

System objects. *See also* Objects 75

System properties

- and default behavior 75
- default values 188
- types of 76

system statement 121

System variables

- creating 121
- declaring data types 126
- determining declared 121
- displaying and editing 339
- using 120

sysThousand character 215

sysTime property 215

sysTimeFormat property 215

T

target property 68, 69, 81

target special term 36

Target window

- commands for setting 93
- setting 96

Tbdb3.dll 284

Tbdlg.dll 284

Tbdos.dll 181, 187, 284

Tbfile32.dll 284

TBM_EVALUATE message 315

- arguments for *sendMessage()* 317

TBM_EVALUATE, structure for evaluation

- buffer 317–318

TBM_EXECUTE message 315

- arguments for *sendMessage()* 316

Tbpx.dll 284

Tbwin.dll 138, 273, 284

Text

- and arrays 130
- and the script editor 49–51
- in strings 144
- selecting in fields 145
- selecting in list boxes 145

Text operators, and arrays 130

textFromPoint() function 146, 199

textline operator 112, 144, 199

textlineCount() function 146

Textlines, working with 144

this special term 36

time data type 124, 215

Time. *See also* Data

Timer events 244

to get allowDrag handler 265

to get command 92

to get handler 38

- getting property values 82–84

to handle command 92

to handle handler 38

to set command 92

to set handler 38

- creating 80
- placing in the hierarchy 81

Toolbar, script editor 45

toolBar system object 75



Contents

- 1 Finding the information you need
 - 2 Understanding OpenScript programming
 - 3 Learning OpenScript basics
 - 4 Using the scripting tools
 - 5 Handling messages, events, objects, and properties
 - 6 Creating an OpenScript statement
 - 7 Using variables, values, and text
 - 8 Writing common scripts
 - 9 Handling user events
 - 10 Managing data
 - 11 Using special effects
 - 12 Using dynamic data exchange
 - 13 Using dynamic-link libraries
 - 14 Debugging OpenScript programs
- ToolBook Debugger
and the Command window 340–341
keyboard shortcuts to 326
using 330–341
- ToolBook Debugger window,
overview 19
- ToolBook II
using as a DDE client 272–279
using as a DDE server 279–282
- ToolBook II system objects. *See* System objects
- toolPalette* system object 75
- Trace box 332
- Trace Call command 334
- Trace Expression command 335
- Trace file, using to record errors 346–348
- Trace Return command 335
- Trace Statement command 334
- transition* command 162
- Transition effects 162
- Translated messages, handling 305–306
- translateWindowMessage* command 93, 301
- Translating Windows messages 300–308
canceling 308
guidelines for 304
returning a value 307–308
- transparent* property 37

U

- UINT data type 290
- unlinkSysBook* message 176
- unsigned* char data type 290
- unsigned int* data type 290
- unsigned long* data type 290
- untranslateWindowMessage* command 308
- Update Script command 53
- uppercase()* function 146
- User defined functions, about 102
- USER DLL module 284
- User entries
checking values 219–220
responding to errors 221
validating using *sysError* 220
- User properties. *See also* Properties about 77
copying 169
creating 77
deleting 78
displaying and editing 339
getting the value of 78
listing 78
- User-defined functions. *See also* Functions
and the object hierarchy 103
calling 104
commands for creating 92
naming 103
- User-defined messages. *See also* messages
defined 62
sending 65
- useWindowsColors* property 151



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

V

Values. *See also* Literal values and arrays 129
assigning to variables 119
checking user entries 219
comparing by case 110
for *saveOnClose* property 172
hex, octal, and binary 139–140
returning for translated messages 307
specifying HLS 150
specifying numeric 136
specifying RGB 149
specifying string 137

Variables. *See also* Local variables and System variables
about 118
and data types 125
and the Command window 56
assigning data types to 123
assigning values to 119
changing the values of 338–339
creating local 120
creating system 121
defined 118
displaying and modifying 335–339
in an *execute* statement 167
naming 60
passing page names 155
renaming for parameters 73
rules for naming 119
sending contents of 169
sending message names as 70
using local and system 120
using reserved 122
viewing 336

Variables window 335
display area 337–338
displayed 336
opening 336

vertices property 239

VGA screens, getting dimensions of 187

void data type 290, 292

VOID FAR * data type 291

W

Web site, click2learn.com 13

Where box 328

while command 92

Why box 328

Win.ini 306

Win31wh.hlp 303

Winconst.hlp 300, 303

Window. *See* Main window 187

Windows libraries, linking 288

Windows messages
canceling translation of 308
establishing translation 302
guidelines for translating 304
translating to OpenScript 93, 300–308

Windows parameters passed with translated messages 305

Windows system colors, using 151

winHandleMain window handle 319

WM_DDE_ADVISE Windows message 272

WM_DDE_EXECUTE Windows message 279–280

WM_DDE_POKE Windows message 279–280



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

WM_DDE_REQUEST Windows message 279–280
WM_DDE_UNADVISE Windows message 272
wMsg parameter 316, 317
wmsg Windows parameter 305
WORD data type 290, 297
 return value for *wRetType* 318
word data type 124
word operator 111
wordCount() function 146
words operator 111
wp Windows parameter 305
wParam parameter 316, 317
wRetType
 return value types for 318
wRetType word 318
writeFile command 223, 226
Writing DLLs 309–319

Y

yieldApp() function 273



Contents

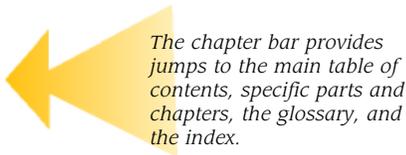
- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

How to use this online book

This online book is designed to help you access information quickly. Use the features described here to jump to the chapter, topic, or resource you need.

Along the left side of every page is the chapter bar, where you can click a chapter you want to look at and jump immediately to the first page of that chapter. At the lower right of each page are navigation buttons that move you to the next or previous page.

On the first page of each chapter is a list of the chapter's contents. Click a topic or its page number to jump directly to that topic.



CONTENTS

G.2 Navigating from the main table of contents

G.3 Navigating from the index

G.4 Using Adobe Acrobat Reader

A yellow arrow pointing up and to the right, with text describing the chapter table of contents.

The chapter table of contents provides jumps to main topics within a chapter.

A yellow arrow pointing down, with text describing navigation buttons.

Navigation buttons take you to the next or previous page.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs



The main table of contents provides jumps to parts and chapters, main topics, the glossary, and the index.

Navigating from the main table of contents

In addition to the chapter bar and the chapter table of contents, you can navigate using the main table of contents, which lists all the chapters in the book and the main topics of each chapter. To quickly jump to the main table of contents, click Contents at the top of the chapter bar.

From the main table of contents, click a chapter title or chapter topic to jump immediately to that chapter or topic.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Navigating from the index

The index lists key words and phrases in this guide. Each item is linked to the page where it appears. From the index, click a word or phrase to jump to the location of the word or phrase.

INDEX

A

[Adobe Acrobat Reader](#) G.4

B

[Button, Find](#) G.6
[Buttons, navigation](#) G.4

C

[Chapter bar](#) G.1

F

[Find button](#) G.6
[Find dialog box](#) G.6
[Forms](#) G.6

N

[Navigating](#) G.2, G.3
[Navigation buttons](#) G.1

P

[Print](#) G.6

R

[Reader Online Guide](#) G.4, G.6

S

[Search](#) G.6
[Setup](#) G.4

T

[Text selection](#) G.5
[Toolbar, Acrobat](#) G.4
[Tools](#)

- [page sizing](#) G.6
- [Text selection](#) G.5
- [Zoom in](#) G.5
- [Zoom out](#) G.5

W

[Web site](#) G.6

Z

[Zoom-in tool](#) G.5
[Zoom-out tool](#) G.5



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

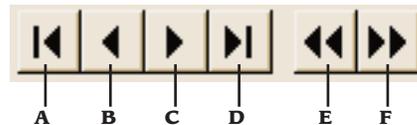
Using Adobe Acrobat Reader

Adobe® Acrobat® Reader is a software program that allows you to view this online guide. Acrobat Reader is installed on your computer during Setup. It comes with a Reader Online Guide, available from the Help menu, which explains how to use the program. The Acrobat toolbar provides some helpful features, described in the next section.

The Acrobat toolbar

In addition to the navigation features in this guide, you will find useful navigation tools on the Acrobat toolbar.

Navigation buttons



- A First page
- B Previous page
- C Next page
- D Last page
- E Previous view
- F Next view

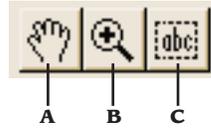
The First, Previous, Next, and Last buttons allow you to move quickly within the online book. Additionally, Acrobat allows you to move to the previous page visited and back again using the Previous view and Next view buttons. This type of navigation is convenient when jumping between chapters or nonsequentially within a chapter.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Zoom and text tools



- A Hand tool
- B Zoom-in tool
- C Text selection tool

Acrobat also comes with tools you can use to zoom in and out of a page, and to move about a page. Use the Hand tool to move around the page by clicking and dragging. The Zoom-in tool magnifies the page with each click. Select the Zoom-in tool and hold down the CTRL key to show the Zoom-out tool.

Another useful tool is the Text selection tool. Use this feature to select a block of text. Then, copy the text to another program.

Here's a tip for those of you with tired eyes.

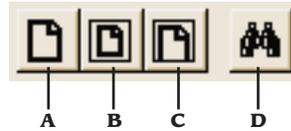
Grab the Zoom-in tool and select this paragraph (as shown). This will magnify the page to the specified area. Use the scroll bars or the Hand tool to move around as you read.



Contents

- 1 Finding the information you need
- 2 Understanding OpenScript programming
- 3 Learning OpenScript basics
- 4 Using the scripting tools
- 5 Handling messages, events, objects, and properties
- 6 Creating an OpenScript statement
- 7 Using variables, values, and text
- 8 Writing common scripts
- 9 Handling user events
- 10 Managing data
- 11 Using special effects
- 12 Using dynamic data exchange
- 13 Using dynamic-link libraries
- 14 Debugging OpenScript programs

Page-sizing tools



- A Zoom 100%
- B Fit page
- C Fit width
- D Find

Using the page-sizing tools, you can quickly change the size of the page you are viewing. This feature comes in handy when you want to restore the view of the whole page after using the Zoom-in tool. This guide is designed to be viewed at the Fit page size. At this size, you can see all of the navigation tools available to you.

The Find button opens the Find dialog box, where you can enter a word or phrase to search for.

Additional Acrobat features

Using features in Adobe Acrobat, you can do many other things, such as print the document, fill in forms, jump to a Web site, and open other files. For more information on these and other features, see the Reader Online Guide, available from the Help menu.

