HyperCard
Script Language Guide

The HyperTalk Language
Contents

Figures and Tables xiii

Preface

About This Guide xix

What’s in This Book? xix
Notation Conventions xxi
Changes Since the First Edition of This Guide xxii
Apple Developer Programs xxiii

Chapter 1

What’s New Since 2.0? 1

HyperCard System Requirements 1
HyperCard Enhancements 2
WorldScript Compatibility 3
HyperCard and Other Scripting Systems 3
Open Scripting Architecture 4
AppleScript 5
Script Attachability 6
Script Editor Enhancements 7
Button Dialog Modifications 7
New Button Features 8
Field Dialog Modifications 12
New Field Features 13
Integrated Stand-Alone Application Builder 14
Enhanced HyperTalk 15

Chapter 2

HyperTalk Basics 23

What Is HyperTalk? 23
Objects 24
Buttons and Fields 24
Cards, Backgrounds, and Stacks 25
Chapter 8 System Messages 125
Messages and Commands 125
Messages Sent to a Button 126
Messages Sent to a Field 128
Messages Sent to the Current Card 131
Message Order 138

Chapter 9 Control Structures and Keywords 141
Keywords in Message Handlers 141
Message Handler Example 144
Keywords in Function Handlers 145
Function Handler Example 149
Repeat Structure 149
Repeat Statements 150
If Structure 155
Single-Statement If Structure 155
Multiple-Statement If Structure 156
Nested If Structures 157

Chapter 10 Commands 165
Redefining Commands 165
Syntax Description Notation 166
### Chapter 11  Functions  289

- Function Calls  289
- Syntax Description Notation  290
- Function Descriptions  291

### Chapter 12  Properties  357

- Retrieving and Setting Properties  357
- Object Properties  358
  - Stack Properties  359
  - Background Properties  360
- Card Properties  361
- Field Properties  362
- Button Properties  365
- Rectangle Properties  367
- Environmental Properties  368
  - Global Properties  369
  - Painting Properties  372
- Window Properties  374
- Menu, Menu Bar, and Menu Item Properties  375
- Message Watcher and Variable Watcher Properties  376
- HyperCard Property Descriptions  377

### Appendix A  External Commands and Functions  503

- Definitions, Uses, and Examples  503
- XCMD and XFCN Resources  503
- Uses for XCMDs and XFCNs  504
- Using an XCMD or XFCN  504
  - Invoking XCMDs and XFCNs  505
  - Message-Passing Hierarchy  505
Guidelines for Writing XCMDs and XFCNs 507
Attaching an XCMD or XFCN 508
Parameter Block Data Structure 509
  Passing Parameters to XCMDs and XFCNs 510
    ParamCount 510
    Params 510
  Passing Back Results to HyperCard 510
    ReturnValue 510
    PassFlag 511
  Callbacks 511
    EntryPoint 511
    Request 511
    Result 511
    InArgs 512
    OutArgs 512
Callback Procedures and Functions 512
  HyperTalk Utilities 513
  Memory Utilities 514
  String Utilities 514
  String Conversions 515
  Field Utilities 518
  Miscellaneous Utilities 519
Creating and Disposing of External Windows 522
  Window Utilities 525
  Text Editing Utilities 529
  Script Editor Utilities 530
  Variable Watcher Support 532
  Debugger Support 533
External Windows 534
  Events in External Windows 536
    Handling Events 537
  Closing an External Window 540
Special XCmdBlock Values 540
  Message Watcher 541
  Variable Watcher 541
  Script Editor 541
  Debugger 542
Appendix B Constants 553

Appendix C Enhancing the Execution Speed of HyperCard 555

<table>
<thead>
<tr>
<th>Change Stacks as Seldom as Possible</th>
<th>556</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Variables, Not Fields, for Operations</td>
<td>556</td>
</tr>
<tr>
<td>Refer to a Remote Card Rather Than Going There</td>
<td>557</td>
</tr>
<tr>
<td>Migrate to XCMDs and XFCNs for Repetitive Tasks</td>
<td>558</td>
</tr>
<tr>
<td>Set LockScreen to True to Avoid Needless Redrawing</td>
<td>558</td>
</tr>
<tr>
<td>Set LockMessages to True During Card-to-Card Data Collection</td>
<td>558</td>
</tr>
<tr>
<td>Combine Multiple Messages</td>
<td>558</td>
</tr>
<tr>
<td>Take Unnecessary Code Out of Loops</td>
<td>559</td>
</tr>
<tr>
<td>Use In-Line Statements Rather Than Handler Calls</td>
<td>559</td>
</tr>
<tr>
<td>Do Complex Calculations Once</td>
<td>560</td>
</tr>
<tr>
<td>Watch Overuse of Variable References</td>
<td>560</td>
</tr>
<tr>
<td>Appendix D</td>
<td>Extended ASCII Table</td>
</tr>
<tr>
<td>-------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>Appendix E</td>
<td>Operator Precedence Table</td>
</tr>
<tr>
<td>Appendix F</td>
<td>HyperCard Synonyms</td>
</tr>
<tr>
<td>Appendix G</td>
<td>HyperCard Limits</td>
</tr>
<tr>
<td>Appendix H</td>
<td>HyperCard Syntax Summary</td>
</tr>
<tr>
<td></td>
<td>Syntax Description Notation</td>
</tr>
<tr>
<td>Appendix I</td>
<td>HyperTalk Vocabulary</td>
</tr>
<tr>
<td></td>
<td>Glossary</td>
</tr>
<tr>
<td></td>
<td>Index</td>
</tr>
</tbody>
</table>
Figures and Tables

Chapter 1  What's New Since 2.0?  1

| Figure 1-1 | Two applications exchanging information using the AppleScript capabilities of HyperCard 2.2 | 4 |
| Figure 1-2 | Button Info dialog box | 8 |
| Figure 1-3 | The Button Contents dialog box | 9 |
| Figure 1-4 | New button styles | 10 |
| Figure 1-5 | Oval style button (shown in Button tool with Show Name checked) | 11 |
| Figure 1-6 | Field Info dialog box | 13 |
| Figure 1-7 | List fields | 13 |
| Figure 1-8 | Building a stand-alone application from your stack | 14 |

| Table 1-1 | Enhanced HyperTalk commands | 15 |
| Table 1-2 | Enhanced HyperTalk functions | 16 |
| Table 1-3 | Enhanced HyperTalk properties | 17 |
| Table 1-4 | Enhanced HyperTalk messages | 20 |

Chapter 2  HyperTalk Basics  23

| Figure 2-1 | HyperCard objects | 24 |
| Figure 2-2 | Relationship between the location of a card and a card window | 29 |

Chapter 3  The Scripting Environment  33

| Figure 3-1 | The Objects menu | 33 |
| Figure 3-2 | Button Info dialog box | 34 |
| Figure 3-3 | Script editor window | 36 |
| Figure 3-4 | Script menu | 37 |
| Figure 3-5 | Find dialog box | 38 |
| Figure 3-6 | Replace dialog box | 39 |
| Figure 3-7 | Nested control structures | 40 |
| Figure 3-8 | The Debugger menu | 44 |
| Figure 3-9 | The Message Watcher window | 46 |
Figure 3-10 The Variable Watcher window 47
Figure 3-11 A selected variable in the Variable Watcher window 48

Table 3-1 Script editor command summary 41
Table 3-2 Debugger command summary 49

Chapter 4 Handling Messages 51

Figure 4-1 Handler that responds to message openStack 55
Figure 4-2 Message-passing hierarchy 57
Figure 4-3 Layered buttons and fields 58
Figure 4-4 Message traversing current hierarchy 59
Figure 4-5 Command sent as a message 60
Figure 4-6 The target 61
Figure 4-7 One stack added to the message-passing hierarchy 63
Figure 4-8 Two stacks added to the message-passing hierarchy 64
Figure 4-9 Removing a stack from the message-passing hierarchy 66
Figure 4-10 Static path before the go command executes 68
Figure 4-11 Dynamic path after the go command executes 69
Figure 4-12 Using the send keyword 70
Figure 4-13 Handler in a card script 74
Figure 4-14 Handler in a stack script 75
Figure 4-15 Intercepting a message 77
Figure 4-16 Parameter passing 79

Table 4-1 HyperTalk’s keywords 54

Chapter 5 Referring to Objects, Menus, and Windows 81

Figure 5-1 Card Info dialog box and descriptors for the same card 82
Figure 5-2 A pathname 90
Figure 5-3 New Stack dialog box 91
Figure 5-4 New Stack dialog card-size pop-up menu 92
Figure 5-5 A custom menu 95

Chapter 6 Values 99

Figure 6-1 Manipulating the selection 107
Appendix H  HyperCard Syntax Summary  573

Table H-1  HyperTalk command syntax  574
Table H-2  HyperTalk function syntax  581

Appendix I  HyperTalk Vocabulary  589

Table I-1  HyperTalk vocabulary  589
PREFACE

About This Guide

This book provides detailed information about HyperTalk, the scripting language of HyperCard. Even a little knowledge of HyperTalk enables you to customize buttons and other parts of HyperCard stacks for your own purposes, and you can use HyperTalk to make the stacks you create act the way you want.

While you’re using HyperCard, you can find information about HyperTalk in the HyperCard Help stack and the HyperTalk Reference stack. These stacks make use of some of HyperCard’s best features, such as multiple windows, computer-supported cross-referencing, and fast text searching.

Some of the concepts in this book, such as message handling and objects, may be new to you. Use this guide as it suits your own style of learning: you might be the kind of person who understands best by thoroughly studying the explanations, or you might be the kind who learns by skimming the material and then playing with HyperTalk—writing scripts or copying the examples and trying them out.

Reference material for beginning scriptors can be found in the HyperCard Reference and the HyperTalk Beginner’s Guide, which are available in the HyperCard software package.

What’s in This Book?

Here’s a brief description of the contents of this book:

Chapter 1, “What’s New Since 2.0?” discusses the differences between earlier versions of HyperCard and HyperCard 2.2. If you are already familiar with HyperTalk as described in the original edition of this book, you can use this chapter as a guide to new information in this edition. If you’re new to HyperTalk, however, and haven’t used the original edition, you’ll probably want to skip Chapter 1 initially and come back to it later.

Chapter 2, “HyperTalk Basics,” introduces the basic concepts of HyperTalk, showing how it is used in the HyperCard environment.
Chapter 3, “The Scripting Environment,” explains how to create and modify scripts in HyperCard objects.

Chapter 4, “Handling Messages,” describes how HyperTalk works, how it carries out actions, and how it responds to events in the HyperCard environment.

Chapter 5, “Referring to Objects, Menus, and Windows,” explains how to refer to objects—the parts of HyperCard that contain HyperTalk scripts and that respond to and initiate actions. It describes how you can use names, numbers, and ID numbers to identify and work with objects, menus, and windows.

Chapter 6, “Values,” explains the elements within HyperTalk that contain values.

Chapter 7, “Expressions,” describes HyperTalk’s operators and explains how HyperTalk evaluates expressions—the descriptions of how to get a value.

Chapter 8, “System Messages,” describes the messages that HyperCard generates in response to events (such as mouse clicks) that happen in its environment.

Chapter 9, “Control Structures and Keywords,” describes the handlers within which you write all HyperTalk scripts to enable objects to respond to messages and function calls. It also describes the control structures of HyperTalk that let you specify how and when sections of scripts execute, and it describes the keywords that you use in control structures.

Chapter 10, “Commands,” describes each of HyperTalk’s built-in commands—the action statements that make HyperCard do things.

Chapter 11, “Functions,” describes HyperTalk’s built-in functions—named values that reflect conditions in the HyperCard environment.

Chapter 12, “Properties,” describes the properties of HyperCard objects—characteristics that determine how objects look and act.

Appendix A, “External Commands and Functions,” contains general information about XCMDs and XFCNs, extensions to HyperTalk that can be written by expert programmers to increase the power of HyperCard.

Appendix B, “Constants,” describes HyperTalk’s built-in constants—named values that don’t change.
Appendix C, “Enhancing the Execution Speed of HyperCard,” provides some helpful hints for scriptors who want to increase the efficiency of HyperCard.

Appendix D, “Extended ASCII Table,” lists the decimal values of the standard Macintosh character set used by HyperCard.

Appendix E, “Operator Precedence Table,” summarizes the order in which HyperTalk performs operations when it evaluates expressions.

Appendix F, “HyperCard Synonyms,” lists the abbreviations and alternate spellings for HyperTalk terms.

Appendix G, “HyperCard Limits,” lists various minimum and maximum sizes and numbers of elements defined in HyperCard.

Appendix H, “HyperTalk Syntax Summary,” shows the syntax of HyperTalk’s command and function parameters in abbreviated form.

Appendix I, “HyperTalk Vocabulary,” lists alphabetically each of the primary HyperTalk terms that HyperCard understands, names the category it’s in, and provides a brief description of its meaning.

This book also contains a glossary of terms commonly associated with the HyperCard environment and an index to help you quickly find specific information contained in this guide.

Notation Conventions

Before you read this guide, you should know about a few typographic conventions. Words or phrases in a monospaced font like this are HyperTalk language elements or are to be typed exactly as shown. New terms are shown in boldface type when they are first introduced and defined. The glossary contains definitions of these terms and other related technical terms.

In descriptions of HyperTalk syntax for commands and other language elements, words in italic type describe general elements, not specific names—you must substitute the actual instances. (These elements are called metasymbols in this book.) Brackets ([ ]) enclose optional elements, which may be included if you need them. (Don’t type the brackets.) In some cases, optional elements change what the message does; in other cases they are helper words that have
PREFACE

no effect except to make the message more readable. The vertical bar symbol (|) indicates a choice of elements: the syntax accepts either the element to the left or the element to the right of the vertical bar. Syntax descriptions for some language elements have a particular format, which is explained at the beginning of the chapter about that language element.

It doesn’t matter whether you use uppercase or lowercase letters in commands or variable names; message names that are formed from two words are shown in small letters with a capital in the middle (likeThis) merely to make them more readable.

Changes Since the First Edition of This Guide

This edition of the HyperCard Script Language Guide is different from the first edition in several ways. Of course, it has new information in it that reflects the new features of HyperCard. In addition, the page format and design are different, and to make finding information easier, a few chapters have been divided into smaller chapters and others have been reorganized.


Chapter 12, “Properties,” now has one main alphabetical list of properties, rather than having the properties grouped by the object or environment to which they can apply. For each property, the objects or environments to which it can apply are listed on the first line after the heading. Also, at the beginning of the chapter, there are tables that list properties by object or environment (button properties, field properties, painting properties, and so forth).
Apple Developer Programs

Apple’s goal is to provide developers with the resources they need to create new Apple-compatible products. Apple offers two programs: the Partners Program, for developers who intend to resell Apple-compatible products, and the Associates Program, for developers who don’t intend to resell products and for other people involved in the development of Apple-compatible products.

As an Apple Partner or Associate, you will receive monthly mailings including a newsletter, Apple II and Macintosh Technical Notes, pertinent Developer Program information, and all the latest news relating to Apple products. You will also receive Apple’s Technical Guide Book and automatic membership in APDA. You’ll have access to developer AppleLink and to Apple’s Developer Hotline for general developer information.

As an Apple Partner, you’ll be eligible for discounts on equipment, and you’ll receive technical assistance from the staff of Apple’s Developer Technical Support department.

For more information about Apple’s developer support programs, contact Apple Developer Programs at the following address:

Apple Developer Programs
Apple Computer, Inc.
20525 Mariani Avenue, M/S 75-2C
Cupertino, CA 95014
CHAPTER 1

What’s New Since 2.0?

This chapter is most useful for those who are already familiar with HyperCard because it describes the enhancements made to HyperCard since HyperCard 2.0. If you haven’t done any scripting with HyperCard before, you should start with Chapter 2, “HyperTalk Basics,” and work your way through the rest of the book. If you are already familiar with HyperCard, you can use this chapter as a guide to locations in the book that cover the new features of HyperTalk in more detail.

HyperCard 2.2 provides more power and flexibility across the entire range of Macintosh computers, starting with the Macintosh Plus. It incorporates features that improve the use of any HyperCard application on the smaller screen of the Macintosh PowerBook, Macintosh Plus, and Macintosh SE and on larger screens commonly used on the Macintosh II and other modular Macintosh computers.

Most of the new features of this HyperCard release are transparent and cannot be seen in the user interface. However, there have been several changes to the dialog boxes and their associated functioning. (For a more detailed description of the user interface, see the HyperCard Reference.) In addition, HyperCard 2.2 stacks are now globally localizable and can be scripted with scripting languages like AppleScript so that stacks can easily exchange information with other programs.

HyperCard System Requirements

HyperCard version 2.0 and later requires system software version 6.0.5 or later. Stacks created with earlier versions of HyperCard are opened as read-only in HyperCard version 2.0 or later. The earlier version stacks are write-protected until converted to the 2.0 format by choosing the Convert Stack command from the File menu.
CHAPTER 1

What's New Since 2.0?

Stacks that wish to take advantage of WorldScript features, open scripting, and the stand-alone application building capabilities of HyperCard 2.2 require system software version 7.1.

HyperCard Enhancements

HyperCard 2.2 provides several feature enhancements that make scripting easier and that enable your HyperCard stacks to be more flexible, robust, and powerful. You have always been able to use HyperTalk scripts to automate and customize your HyperCard application, but now, with Apple’s Open Scripting Architecture (OSA) extensions to HyperCard 2.2, you can use scripts to integrate HyperCard with other applications so that you can use features of other applications in your stacks by exchanging data with those applications.

Most of the features of this release are controllable from the HyperTalk script language and can be easily used from any existing HyperCard script. Almost all of the improvements of this release are transparent, but are nonetheless important in making your stacks increase the execution speed of HyperCard. The following list summarizes the new features of HyperCard 2.2.

- WorldScript compatibility, which makes it easy to produce localizable stacks
- support for Apple’s Open Scripting Architecture and communication between applications, including
  - support for the AppleScript script language and other OSA-compliant scripting languages
  - script attachability so that you can choose the scripting language you wish to use in the HyperCard environment
  - HyperTalk equivalents to some AppleScript commands
  - HyperTalk support for embedded AppleScript instructions
- support for new button styles and features, including
  - standard buttons (rounded rectangle style without a shadow); similar to those found in most dialog boxes for the Cancel button
  - default buttons (with the additional 3-pixel-wide outline); similar to the OK button found in most standard dialog boxes
  - oval buttons (transparent, to overlay circular and oval shapes); HyperCard respects the actual shape of the button when tracking mouse actions
CHAPTER 1

What’s New Since 2.0?

- pop-up menu buttons, which include a resizable title field and a menu area
- behavior conforming to the standards of *Macintosh Human Interface Guidelines* built into checkbox and radio button styles
- the family, partNumber, and enabled properties
- fields that behave as lists, including highlighting of list items when clicked
- an integrated stand-alone application builder
- miscellaneous improvements to HyperTalk made in response to developer requests; some examples are
  - mouseDoubleClick system message
  - script-controlled enabling and disabling of buttons
  - text sorting by sort keys

**WorldScript Compatibility**

Software and Stackware are more commonly distributed worldwide now than when HyperCard was created, and the Macintosh system is now easily localized. Modifications have been made throughout HyperCard 2.2 to make it sensitive to the current key script or the current font script, as appropriate, in determining whether to invoke its special-case code for handling non-Roman text characteristics.

Also, the `convert` command now works with dates and times written in any format supported by any script installed in your system. See the `convert` command in Chapter 10, “Commands.”

**HyperCard and Other Scripting Systems**

Using any scripting system that’s compliant with Apple’s Open Scripting Architecture (defined in the next section), like AppleScript, you can write scripts that extend the functionality of your stacks by integrating them with other scriptable applications, such as Claris FileMaker Pro 2.0 and Microsoft Excel 4.0. For example, you might want to store records detailing a large library of films and film clips in a database program like Claris’s FileMaker Pro, which is built to handle large amounts of data efficiently.
CHAPTER 1

What's New Since 2.0?

The stack in Figure 1-1 uses AppleScript statements to request the FileMaker Pro Films database to look up a film (Vertigo) and send all the information it contains about the film to the HyperCard stack. The HyperCard stack then adds the information to its Hitchcock films stack, which is a smaller subset of all the films in the larger database.

The HyperCard stack also contains a script that examines the information sent back from the database to determine if a QuickTime clip of that movie exists. The HyperCard films stack then takes the information sent back from the Trailer cell in the films stack to look up the film clip and play it.

By working with another application, this HyperCard stack has extended what both applications can accomplish.

Open Scripting Architecture

Apple System software’s Open Scripting Architecture (OSA) provides a standard mechanism that allows users to control multiple applications with scripts written in a variety of scripting languages.

Figure 1-1  Two applications exchanging information using the AppleScript capabilities of HyperCard 2.2

Films ‘R Us

The Films of Alfred Hitchcock

Vertigo

The Birds
Blackmail
Champagne
Dial M For Murder
Dowhill
Easy Virtue
Family Plot
The Fiancée’s Wife
Foreign Correspondent
Frenzy
G Continent
Siamiaa Ison
June and the Paycock
The Lady Vanishes
Lifeboat
The Lodger
The Man Who Knew Too Much
The Man Who Knew Too Much
The Manxman
Marnie
CHAPTER 1

What’s New Since 2.0?

Each scripting language that utilizes OSA has a corresponding scripting component within its own application. When an application’s scripting component executes a script, it performs the actions described in the script, much like HyperCard executes HyperTalk scripts.

HyperCard 2.2 users can now use its AppleScript extensions to communicate with applications and objects that are outside of HyperCard. This process is enabled by sending and receiving OSA-defined messages called Apple events.

Applications typically use Apple events to request services and information from other applications or to provide services and information in response to such requests.

AppleScript

The AppleScript component of the OSA, which implements the AppleScript scripting language, is the implementation provided by Apple that allows applications to exchange information and data.

AppleScript has a number of features that set it apart from other scripting systems:

- The AppleScript language makes it easy to refer to data within applications. Scripts refer to objects that closely correspond to familiar objects in applications. For example, a script can refer to paragraph, word, and character objects in a word-processing document and to row, column, and cell objects in a spreadsheet.

- You can script many different applications. Although there are applications that include built-in scripting or macro languages, most of these languages work for only one application. In contrast, you can use AppleScript to control any of the applications that support it. You don’t have to learn a new language for each application.

- You can write scripts that control applications on more than one computer. A single script can control any number of applications, and the applications can be on any computer within a given network.

- AppleScript supports multiple dialects. These additional dialects can use words from another human language, such as Japanese, and have a syntax that resembles a specific human language or programming language. You can convert a script from one dialect to another without changing what happens when you run the script.
Comparing AppleScript and HyperTalk

AppleScript and HyperTalk are fairly similar. They both work by sending messages to objects within their systems. The major difference is that the system encompassing HyperTalk is the HyperCard application, whereas the system for the AppleScript language is the Macintosh system software. Essentially, this means that whereas HyperTalk instructions can be understood only by HyperCard, any application could potentially understand and act on a set of AppleScript instructions.

AppleScript works by sending messages, called *Apple events*, from scripts in one application to an object. An example could be an application within the environment of your Macintosh. HyperTalk works by sending messages, called *system messages, commands, and functions*, to HyperCard objects, like stacks, cards, backgrounds, buttons, and fields.

HyperCard 2.2 supports AppleScript by implementing a set of related Apple events called *Apple event suites*, as defined in the *Apple Event Registry*, including the Required suite, the Core suite, and the Text suite. It adds to these suites the HyperCard suite, an extension to the Apple events understood by AppleScript that exposes the functionality of HyperTalk to external scripting systems. The HyperCard suite includes AppleScript equivalents for most of the commands, functions, and properties defined in this book.

You can learn more about the AppleScript language and its tools from the *AppleScript Language Guide*.

Script Attachability

In HyperCard 2.2, scripts of HyperCard objects can be written either in HyperTalk or in a language defined by any external scripting system, such as AppleScript, that implements the optional attachability interface defined by the Open Scripting Architecture.

When HyperCard 2.2 passes a message to an object, it checks to see whether the script attached to that object is a HyperTalk script or a script belonging to an external scripting system, such as AppleScript.

If the script is a HyperTalk script, HyperCard uses its built-in mechanism for invoking HyperTalk message handlers. However, if the script is from an external scripting system like AppleScript, then HyperCard translates the message into an Apple event and uses a system software extension to invoke the relevant message handler.
A script attached to a HyperCard object receives HyperCard system messages according to its position in the inheritance hierarchy, regardless of its language. For example, a card script can handle the standard openCard message whether it’s written in HyperTalk, AppleScript, or UserTalk.

Scripts can also pass messages to other scripts without regard to their language. When HyperCard receives an OSA-defined event, it translates the event to a message and sends it along the current message path.

**Script Editor Enhancements**

The script editor in HyperCard 2.2 is capable of editing text-based scripts belonging to any OSA-compliant scripting system, such as AppleScript. The script editor now has a pop-up menu that lists the available scripting systems, including HyperTalk, and lets you select the scripting system you want to use for the currently displayed script. You can also use the global property scriptingLanguage to set the scripting language you want to use for a stack in a startUp script. See the scriptingLanguage property in Chapter 12, “Properties.”

**Button Dialog Modifications**

The Button Info dialog box contains these new items:

- a Preview area displaying a picture of the button with its currently selected attributes
- static text displaying the button’s part number
- a checkbox for setting the button’s enabled property
- a pop-up menu for selecting a number for the button’s family property
- a Contents button for editing the contents of the button
- a Text Style button for setting the text font, size, and style of the button’s name (and contents for pop-up buttons)
- a pop-up menu for selecting the button’s style property (this replaces the set of radio buttons for button styles used by previous versions of HyperCard)
- a text-entry area, displayed only for pop-up buttons, for setting the button’s titleWidth property
CHAPTER 1

What's New Since 2.0?

In addition, the Button Info dialog box is now a movable modal dialog box and can be dragged to new positions by the user. HyperCard 2.2 remembers the positions of movable modal dialog boxes while running; it does not remember their positions after you quit the program. Figure 1-6 shows the new Button Info dialog box.

Figure 1-2   Button Info dialog box

![Button Info dialog box](image.png)

New Button Features

In HyperCard 2.2, buttons are now valid HyperTalk containers; thus, they can contain data. The contents of buttons can be edited in a dialog box that’s accessible from the Button Info dialog box (see the Contents button in Figure 1-6). A button on the Clipboard can be pasted along with its contents by holding down the Shift key (the same as pasting a field and its contents). In HyperTalk, button expressions are now valid; they evaluate to the contents of the specified button (the same as field expressions evaluating to the specified field’s contents).
The contents of a pop-up button have a special purpose. You use them to create the menu that pops up when the user clicks the pop-up button. Each line within the contents of the button becomes a menu item in the menu. Figure 1-3 shows the Button Contents dialog box containing the list of menu items for the SodaPopup button in Figure 1-6.

Scripts can determine the number and text of the pop-up menu item that is currently selected by using the selectedLine and selectedText functions, described in Chapter 11, “Functions.” You can use the contents of other types of buttons for any purpose.

Support for new styles of buttons in HyperCard 2.2 makes it easier to create stacks that comply with the Macintosh Human Interface Guidelines. For instance, you can now easily create the standard default buttons that appear in many Macintosh dialog boxes. HyperCard 2.2 also makes it easier to create radio
buttons and checkboxes that behave according to *Macintosh Human Interface Guidelines* standards without requiring a lot of scripting to get the standard behavior. Some of these buttons are shown in Figure 1-4.

![New button styles](image)

In Figure 1-4, the Cancel button, shown in the standard button style; the OK button, shown in the default button style; and the Test Popup button, shown in the pop-up button style, illustrate three of the new styles.

As shown in Figure 1-4, pop-up buttons have both a title area and a menu area. The title of the button is drawn to either the right or the left of the menu area, depending on the script system of the button’s font. If the script system is a left-to-right script system, the title is drawn on the left; otherwise, it is drawn on the right.

When the user resizes the pop-up button by dragging it from one of its corners, the width of the title area remains fixed. The user can drag the dividing line between these areas to widen one and narrow the other. You also can change the title area’s width either by changing the Title Width value in the Button Info dialog box, or by setting the titleWidth property of the button from a script.

The default button style does not automatically provide the behavior a user expects from clicking a default button. You can provide that behavior by including a `returnKey` handler and an `enterKey` handler in the card, background, or stack script. Here is an example:

```on returnKey
    if the selection is empty then pressDefaultBtn
    else pass returnKey
end returnKey

on enterKey
    if the selection is empty then pressDefaultBtn
    else pass enterKey
end enterKey```

HyperCard Enhancements
CHAPTER 1

What's New Since 2.0?

on pressDefaultBtn
  -- click the default button
  put the number of buttons into btnCount
  repeat with i = 1 to btnCount
    if the style of button i is default then
      click at the loc of button i
      exit pressDefaultBtn
    end if
  end repeat
end pressDefaultBtn

Also new is the oval style button, which is transparent. In the Button tool, both the rectangular and oval frames are visible, as shown in Figure 1-5.

Figure 1-5  Oval style button (shown in Button tool with Show Name checked)

The transparency of oval buttons makes them useful for overlaying oval or circular shapes. HyperCard respects this oval shape when tracking the mouse above it for highlighting or for sending messages. For example, mouseWithin messages aren’t sent until the pointer enters the oval, and mouse clicks must be within the oval to count as clicks within the button. The shape of an oval button is defined by its rect property. Oval buttons whose rect property is a square are, not surprisingly, circular.

There are other less visible changes that have been made to buttons. For instance, there are five new button properties:

- The family property, which you can use to group buttons, makes it easy to get the behavior Macintosh users expect from a group of radio buttons.

- The partNumber property, which was invented specifically to give you read/write access to a property that represents the ordering of buttons and fields within their backgrounds and cards. The partNumber property of a
CHAPTER 1

What’s New Since 2.0?

button or field represents the ordinal position of the button or field among the objects of both kinds—buttons and fields—of the same card or background.

- The enabled property determines whether the button appears and behaves in an enabled or a disabled state. When a button is enabled, it appears and behaves normally. When it is disabled, the button is dimmed and it ignores mouse clicks—it neither highlights nor receives mouse messages.

- The scriptingLanguage property determines the scripting system of the button’s script. For example, you can use the scriptingLanguage property to set a button to accept scripts written in AppleScript.

- The titleWidth property determines the width of the title area of a pop-up button. You can change the title area’s width by changing the Title Width value in the Button Info dialog box, or by setting the titleWidth property of the button from a script.

You can read more about these properties in Chapter 12, “Properties.”

Field Dialog Modifications

The Field Info dialog box contains these new items:

- a Preview area displaying a picture of the field with its currently selected attributes
- static text displaying the field’s part number
- checkboxes for setting the field’s autoHilite and multipleLines properties (described in the section “New Field Features”)
- a pop-up menu for selecting the field’s style property (this replaces the set of radio buttons for field styles used by previous versions of HyperCard)

In addition, the Field Info dialog box is now a movable modal dialog box and can be dragged to new positions by the user. HyperCard 2.2 remembers the positions of movable modal dialog boxes while running; it does not remember their positions after you quit the program. Figure 1-6 shows the new Field Info dialog box.
NEW FIELD FEATURES

HyperCard 2.2 lets you create fields that behave as lists. When the lockText and autoHilite properties of a field are set to true, the field responds to mouse clicks by highlighting the line that was clicked. If the multipleLines property is true, the user can hold down the Shift key and click again to extend the range of highlighted lines. Scripts can determine the range of lines that are currently selected in a list field by using the selectedLine function, and they can get the text of the selected lines by using the selectedText function, described in Chapter 11, “Functions.” Figure 1-7 shows two list field examples.
CHAPTER 1

What's New Since 2.0?

The scrolling list field allows the user to click an item in the list to select it or to scroll through the list to peruse its contents without selecting anything. You can read more about the standard behavior of scrolling lists in the Macintosh Human Interface Guidelines.

Like buttons, fields also have a partNumber property and a scriptingLanguage property. See the discussion in the previous section for more information. You can read more about these properties in Chapter 12, “Properties.”

Integrated Stand-Alone Application Builder

HyperCard 2.2 enables you to build a stand-alone application without the use of specialized external tools. This capability requires system software version 7.0 or later.

The standard file dialog box, shown in Figure 1-8, appears when you choose Save a Copy from the File menu. This dialog box contains a pop-up menu from which you can choose a file format. The choices in the pop-up menu are, at a minimum, Stack and Application, representing the standard stack and application formats, respectively. Third-party developers can add more file format choices by creating additional file translation modules for use with HyperCard 2.2, such as a stack-to-ScriptX translator or a stack-to-text translator.

Figure 1-8 Building a stand-alone application from your stack
CHAPTER 1

What’s New Since 2.0?

Enhanced HyperTalk

The HyperTalk vocabulary has been enlarged to accommodate the new features within HyperCard. Table 1-1 describes the enhanced HyperTalk commands. Table 1-1 also includes the enhanced keyword do. Table 1-2 describes the enhanced HyperTalk functions. Table 1-3 describes the enhanced HyperTalk properties. Note that the information in the “Definition” column of Tables 1-2 and 1-3 describes only new functionality. Table 1-4 describes the enhanced HyperTalk messages. See the appropriate chapters for syntax, definitions, and examples of items in these tables.

Table 1-1  Enhanced HyperTalk commands

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>answer program promptText [of type processType]</td>
</tr>
<tr>
<td>close [docPathname with</td>
</tr>
<tr>
<td>convert [chunk of] container literal [from format][and format][to format][and format]</td>
</tr>
<tr>
<td>delete part</td>
</tr>
<tr>
<td>disable menu</td>
</tr>
<tr>
<td>disable menuItem of menu</td>
</tr>
<tr>
<td>disable button</td>
</tr>
<tr>
<td>do expression as scriptingLanguage</td>
</tr>
<tr>
<td>doMenu itemName [,menuName][without dialog][with modifierKey [,modifierKey]]</td>
</tr>
<tr>
<td>enable button</td>
</tr>
<tr>
<td>find [international] text [in field][of marked cards]</td>
</tr>
<tr>
<td>find chars [international] text [in field][of marked cards]</td>
</tr>
<tr>
<td>find string [international] text [in field][of marked cards]</td>
</tr>
<tr>
<td>find whole [international] text [in field][of marked cards]</td>
</tr>
<tr>
<td>find word [international] text [in field][of marked cards]</td>
</tr>
<tr>
<td>lock error dialogs</td>
</tr>
<tr>
<td>open [fileName with] application</td>
</tr>
<tr>
<td>picture [fileName,fileType,windowStyle,visible,depth,floatingLayer]</td>
</tr>
</tbody>
</table>

HyperCard Enhancements 15

continued
CHAPTER 1

What’s New Since 2.0?

Table 1-1 Enhanced HyperTalk commands (continued)

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>put text into button</td>
</tr>
<tr>
<td>read from file fileName [at [-]start]</td>
</tr>
<tr>
<td>for numberOfChars until char</td>
</tr>
<tr>
<td>reply expression [with keyword expression]</td>
</tr>
<tr>
<td>reply error expression</td>
</tr>
<tr>
<td>request expression from program</td>
</tr>
<tr>
<td>request appleEvent class</td>
</tr>
<tr>
<td>select line number [to number] of field</td>
</tr>
<tr>
<td>select line number of button</td>
</tr>
<tr>
<td>sort [lines</td>
</tr>
<tr>
<td>unlock error dialogs</td>
</tr>
<tr>
<td>visual [effect] push left</td>
</tr>
<tr>
<td>write text to file fileName [at [-]start</td>
</tr>
</tbody>
</table>

Table 1-2 Enhanced HyperTalk functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>destination</td>
<td>Returns the full pathname of the destination stack when HyperCard is in the process of going to another stack.</td>
</tr>
<tr>
<td>diskSpace</td>
<td>Returns the amount of free space on any mounted volume.</td>
</tr>
<tr>
<td>number</td>
<td>Returns the number of menu items in a specified menu or the number of parts of a card or background. (These are new features of the number function. For a complete description of the number function, see Chapter 11, “Functions.”)</td>
</tr>
<tr>
<td>programs</td>
<td>Returns a return-delimited list of all the System 7–friendly processes currently running on your machine.</td>
</tr>
</tbody>
</table>

continued
## CHAPTER 1

What's New Since 2.0?

### Table 1-2  Enhanced HyperTalk functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>selectedButton</td>
<td>Returns the name of the button that is highlighted in a family of buttons.</td>
</tr>
<tr>
<td>selectedLine</td>
<td>Returns the line number of the selected line (or lines) in a list field or pop-up style button.</td>
</tr>
<tr>
<td>selectedText</td>
<td>Returns the text of the selected line (or lines) in a list field or pop-up style button.</td>
</tr>
<tr>
<td>sum</td>
<td>Returns the sum of a list of comma-delimited numbers.</td>
</tr>
<tr>
<td>systemVersion</td>
<td>Returns a decimal string that represents the running version of system software.</td>
</tr>
</tbody>
</table>

### Table 1-3  Enhanced HyperTalk properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>Global; determines where HyperCard is running—that is, the full path, including network zone and machine name. This property works only in System 7.</td>
</tr>
<tr>
<td>autoHilite</td>
<td>Fields; defines a list field if the lockText property is also set to true.</td>
</tr>
<tr>
<td>bottom</td>
<td>Menu bar; determines the value of item four of the rectangle property (left, top, right, bottom) when applied to the menu bar.</td>
</tr>
<tr>
<td>bottomRight</td>
<td>Menu bar; determines the value of items three and four of the rectangle property (left, top, right, bottom) when applied to the menu bar.</td>
</tr>
<tr>
<td>dialingTime</td>
<td>Global; determines how long HyperCard waits before closing the serial connection to a modem after dialing.</td>
</tr>
<tr>
<td>dialingVolume</td>
<td>Global; sets the volume of the touch tones generated through the Macintosh speaker by the dial command.</td>
</tr>
</tbody>
</table>

continued
CHAPTER 1

What's New Since 2.0?

Table 1-3  Enhanced HyperTalk properties (continued)

<table>
<thead>
<tr>
<th>Property</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>enabled</td>
<td>Buttons; determines or changes whether the specified button appears and behaves in an enabled or a disabled state.</td>
</tr>
<tr>
<td>environment</td>
<td>Global; determines the environment of the currently running HyperCard application; returns development if it is the fully enabled development version and returns player if the HyperCard player is running.</td>
</tr>
<tr>
<td>family</td>
<td>Buttons; groups two or more buttons together into a family specified by the numbers 1 to 15, inclusive. If the button is part of a button family, clicking one of the buttons highlights it and unhighlights the rest of the buttons in that family.</td>
</tr>
<tr>
<td>height</td>
<td>Menu bar; determines the vertical distance, in pixels, occupied by the rectangle of the menu bar.</td>
</tr>
<tr>
<td>hilite</td>
<td>Buttons; determines whether the specified button is highlighted (displayed in inverse video). If the hilite property of a button in a family is set to true, the hilite property of the other buttons is automatically set to false.</td>
</tr>
<tr>
<td>ID</td>
<td>Windows, menus, and applications; determines the permanent ID number of a window or menu and determines the application signature of an application.</td>
</tr>
<tr>
<td>itemDelimiter</td>
<td>Global; determines what delimiter is used to separate a list of items. HyperCard resets the delimiter to its default, the comma, when the computer is idle.</td>
</tr>
<tr>
<td>left</td>
<td>Buttons, fields, and windows; determines the value of item one of the rectangle property (left, top, right, bottom) when applied to the specified object or window.</td>
</tr>
</tbody>
</table>

continued
CHAPTER 1

What's New Since 2.0?

Table 1-3  Enhanced HyperTalk properties (continued)

<table>
<thead>
<tr>
<th>Property</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>lockErrorDialogs</td>
<td>Global; determines or changes whether HyperCard displays an error dialog box in response to an error while executing a script.</td>
</tr>
<tr>
<td>lockText</td>
<td>Fields; defines a list field if the autoHilite property is also set to true.</td>
</tr>
<tr>
<td>name</td>
<td>Windows and HyperCard itself; determines the name of the specified object.</td>
</tr>
<tr>
<td>[english] name</td>
<td>Menus and menu items; the adjective english lets your code test for the names of menus and menu items.</td>
</tr>
<tr>
<td>number</td>
<td>Windows; determines the number within the window layer of any window on your screen.</td>
</tr>
<tr>
<td>owner of card</td>
<td>Cards; returns the name or ID of the background shared by this card.</td>
</tr>
<tr>
<td>owner of window</td>
<td>Windows; returns the name of the entity that created the window; this could be HyperCard or the name of an XCMD like Picture, for example.</td>
</tr>
<tr>
<td>partNumber</td>
<td>Buttons and fields; determines or changes the number that represents the ordering of the buttons and fields within their enclosing card or background. Setting this property can have the effect of either bringing the object closer or moving it farther (behind) other buttons and fields.</td>
</tr>
<tr>
<td>rect[angle]</td>
<td>Menu bar; reports the size of the menubar rectangle. This is a read-only property.</td>
</tr>
<tr>
<td>right</td>
<td>Menu bar; determines the value of item three of the rectangle property (left, top, right, bottom) when applied to the menu bar.</td>
</tr>
<tr>
<td>scriptingLanguage</td>
<td>Objects that can have a script and HyperCard itself; the scripting system used for the scripts of objects.</td>
</tr>
</tbody>
</table>
CHAPTER 1

What's New Since 2.0?

Table 1-3  Enhanced HyperTalk properties (continued)

<table>
<thead>
<tr>
<th>Property</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>style of button</td>
<td>Buttons; new button styles are standard, default, oval, and popup.</td>
</tr>
<tr>
<td>titleWidth</td>
<td>Pop-up buttons; determines or changes the width of the title field for a pop-up button.</td>
</tr>
<tr>
<td>top</td>
<td>Menu bar; determines the value of item two of the rectangle property (left, top, right, bottom) when applied to the menu bar.</td>
</tr>
<tr>
<td>topLeft</td>
<td>Menu bar; determines the value of items one and two of the rectangle property (left, top, right, bottom) when applied to the menu bar.</td>
</tr>
<tr>
<td>visible</td>
<td>Menu bar; determines or changes whether the menu bar is shown or hidden on the screen.</td>
</tr>
<tr>
<td>width</td>
<td>Menu bar; determines the horizontal distance, in pixels, occupied by the rectangle of the menu bar.</td>
</tr>
</tbody>
</table>

Table 1-4  Enhanced HyperTalk messages

<table>
<thead>
<tr>
<th>Message</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>appleEvent class, id, sender</td>
<td>Sent to the current card when an Apple event is received.</td>
</tr>
<tr>
<td>closePalette paletteWindowName, paletteWindowID</td>
<td>Sent to the current card when a palette that was opened with the palette command is closed.</td>
</tr>
<tr>
<td>closePicture pictureWindowName, pictureWindowID</td>
<td>Sent to the current card when a window that was created with the picture command is closed.</td>
</tr>
<tr>
<td>errorDialog</td>
<td>Sent to the current card when a script execution error occurs and lockErrorDialogs is set to true.</td>
</tr>
</tbody>
</table>

continued
## What's New Since 2.0?

### HyperCard Enhancements

<table>
<thead>
<tr>
<th>Message</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>mouseDoubleClick</td>
<td>Sent to a button, field, or card after a second mouse click is released, when all of the following conditions are true:</td>
</tr>
<tr>
<td></td>
<td>• The second click is within the double-click time interval set in the Mouse control panel.</td>
</tr>
<tr>
<td></td>
<td>• The second click is at a location within 4 pixels of the first click.</td>
</tr>
<tr>
<td></td>
<td>• The second click is within the same object as the first click.</td>
</tr>
<tr>
<td>openPalette</td>
<td>Sent to the current card when a palette is opened with the <code>palette</code> command.</td>
</tr>
<tr>
<td>paletteWindowName,</td>
<td></td>
</tr>
<tr>
<td>paletteWindowID</td>
<td></td>
</tr>
<tr>
<td>openPicture</td>
<td>Sent to the current card when a window is created with the <code>picture</code> command.</td>
</tr>
<tr>
<td>pictureWindowName,</td>
<td></td>
</tr>
<tr>
<td>pictureWindowID</td>
<td></td>
</tr>
</tbody>
</table>
This chapter explains HyperTalk’s place in the HyperCard system and describes some of HyperTalk’s characteristics.

Most concepts are discussed only briefly in this chapter, with more detailed discussion left for later chapters.

What Is HyperTalk?

HyperTalk is the scripting language of the HyperCard environment. It allows you to perform actions on HyperCard objects: buttons, fields, cards, backgrounds, and stacks as well as other elements of HyperCard, such as menus and windows.

You use HyperTalk to send messages to and from HyperCard objects. You generate a message by (among other means) clicking the mouse, opening a card, typing a statement into the Message box, or choosing a menu item. Or you can generate a message by sending an Apple event message from a program—or process—outside HyperCard.

How a given object responds to a particular message depends on the object’s script, or in the case of menu items, the menu message for that menu item. Most HyperCard scripts are written in HyperTalk, though version 2.2 of HyperCard makes AppleScript another scripting option.
CHAPTER 2

HyperTalk Basics

Objects

There are five kinds of objects in HyperCard: buttons, fields, cards, backgrounds, and stacks. (See Figure 2-1.)

Buttons and Fields

Buttons are action objects or “hot spots” on the screen. For example, clicking a button with the Browse tool can take you to the next card in a stack. Clicking a pop-up button lets you choose an action from a menu.

Figure 2-1  HyperCard objects
CHAPTER 2

HyperTalk Basics

Fields contain editable text. The Browse tool pointing hand changes to an I-beam when it’s over an unlocked field. (The card or background might also contain Paint text characters. Such characters are not editable once they are placed; they become part of the picture on the card or background.)

Cards, Backgrounds, and Stacks

The basic unit of information is the card: when you look at the screen of a Macintosh computer running HyperCard, what you see foremost is a card. Cards are viewed through card windows. Cards can be larger than the card window and can be scrolled in the card window with the Scroll command in the Go menu. The background is where you place elements that you want a group of cards to have in common. Each card has one background. The card overlays the background; both are the same size. What you see in the card window belongs to the card or to the background. Both the card and background can contain buttons, fields, and pictures. Cards are grouped in stacks; each stack is a Macintosh file. Each stack can have multiple backgrounds.

The card that is currently displayed, the background associated with it, and the stack they are in are termed the current card, background, and stack. The concept of being current doesn’t apply to buttons or fields. Chapter 5, “Referring to Objects, Menus, and Windows,” contains details about referring to objects.

Messages

HyperCard objects interact with each other, with the user, with HyperCard, and with the Macintosh environment by sending messages. Some messages are descriptions of things that happen in the environment, such as that the mouse has been clicked or a card opened: these are system messages. They are like news flashes announced to the community of objects. For example, if you press the mouse button down, HyperCard sends the message mouseDown; when you let the mouse button up, HyperCard sends the message mouseUp. Chapter 8, “System Messages,” contains more information about system messages.

Messages are sent to various objects in a particular order. For example, system messages generated by the mouse go first to the topmost button or field (if any) under the pointer on the screen. Next those messages go to the card, then to
the background, then the stack, then the Home stack, and finally to HyperCard itself. (You’ll find a detailed discussion of this hierarchical sequence in Chapter 4, “Handling Messages.”)

HyperTalk commands are also messages—orders to do some particular thing, like add two numbers or go to another card. A command, whether executed in a script or typed into the Message box, is sent as a message.

**Scripts**

Every HyperCard object has a script (although the script can be completely empty). A script is a collection of any number of handlers. A handler is a collection of HyperTalk statements; each statement ends with a return character. Any part of a statement following HyperTalk’s double-hyphen comment character (--) is ignored by HyperCard.

A handler is invoked when a particular message is received by the object whose script contains the handler. A simple handler looks like this:

```hyperTalk
on mouseUp
  go to next card
end mouseUp
```

The first line of a handler always begins with one of two words—either on or function. The last statement of a handler always begins with the keyword end. All HyperTalk statements always appear within handlers in a script.

You must place handlers in the scripts of objects that will receive the messages you want the handlers to respond to. The message-passing hierarchy, which determines where messages are sent, is described in Chapter 4, “Handling Messages.”

**Message Handlers**

A handler that begins with on is called a message handler. The example in the previous section is a message handler. This particular message handler is in the script of a button; it handles the message mouseUp, and then goes to the next card.
CHAPTER 2

HyperTalk Basics

The message to which a handler responds begins with the word following the word on. In this case, the message is `mouseUp`. When you release the mouse button while the Browse tool is inside a button’s rectangle on the screen, HyperCard sends the message `mouseUp` to the button. HyperCard looks in the button’s script for a handler matching the message `mouseUp`. If it finds a match, it executes the HyperTalk statements between `on mouseUp` and `end mouseUp—in this case, `go to next card`.

Function Handlers

In addition to message handlers, scripts can contain user-defined function handlers. Function handlers begin with the word `function` instead of the word `on`; the name of the function they handle is the second word. A function handler looks like this:

```
function day
    return first item of the long date
end day
```

This function handler responds to a HyperTalk statement containing the function’s name followed by parentheses—a function call. Here’s an example:

```
put day() into message box
```

The function call is `day()`—the rest of the line and the function call together form a statement. When the function call is made, HyperCard looks for the matching function handler. If it finds one, it executes the lines between `function day` and `end day`. The value derived from the expression `first item of the long date` is returned to the `put` statement in place of `day()`. In the example, the value returned by the function (Friday, for example) is put into the Message box.

Function calls use the same message-passing hierarchy as messages; it’s described in Chapter 4, “Handling Messages.” Message and function handler structures are described in detail in Chapter 9, “Control Structures and Keywords.”
Windows are another HyperCard element that you can control with the HyperTalk language. Windows share many properties with objects but are not HyperCard objects because they do not have scripts or respond to messages. (Scripts and messages are described earlier in this chapter; see the section “Properties” in Chapter 6, “Values,” for an introduction to HyperCard properties.)

HyperCard windows include card windows, the FatBits window, the Tools palette, the Patterns palette, the Scroll window, the Message box window, and external windows.

HyperCard’s built-in external windows include the script editor, the Message Watcher, the Variable Watcher, and the Navigator palette. User-defined external windows include windows created with the picture command and windows created with external commands. (The picture command is described in Chapter 10, “Commands,” and external commands, which control external windows, are described in Appendix A, “External Commands and Functions.”)

Card Windows

You view the cards in a HyperCard stack through a card window. Card windows have a title bar, a zoom box, and if there is more than one stack open at the same time, a close box. A stack’s card window can be the same size as or smaller than the current card size. You can reposition card windows by dragging them or setting HyperTalk properties.

HyperTalk allows you to control the following features of a stack’s card window:

- the current size of the card window
- the current location of the card window
- the type of title shown in the title bar of the card window
- the current position or scroll of the card within the card window

Card windows are resized automatically when the card is resized from either the Stack Info dialog box or with HyperTalk. You can resize card windows...
independently of the card. Card windows are located within the global coordinates of the screen. Cards are located within the card window's local coordinates. The representation of the relationship between the location of the card window and a larger card is shown in Figure 2-2. You can scroll the contents of a card that is larger than the current card window with the

**Figure 2-2** Relationship between the location of a card and a card window

![Diagram showing the relationship between card window and card coordinates]
HyperTalk Basics

**Scroll property.** See the `location`, `rectangle`, and `scroll` properties in Chapter 12, “Properties,” for more information about moving and sizing card windows and cards.

To find out how to refer to HyperCard card windows with the HyperTalk language, see Chapter 5, “Referring to Objects, Menus, and Windows.”

**HyperCard’s Built-in External Windows**

HyperCard has several built-in external windows. They are the script editor windows, the Message Watcher window, and the Variable Watcher window. You can control the following properties of these windows with HyperTalk:

- the current font and style of text to use in the script editor window
- the current location of the Message Watcher window
- the current location of the Variable Watcher window
- the current size of the Variable Watcher window
- the positions of the horizontal and vertical bars of the Variable Watcher window

For more information about the script editor window, the Message Watcher window, and the Variable Watcher window, see Chapter 3, “The Scripting Environment.”

**Menus**

You can also control the behavior of HyperCard menus with HyperTalk commands. Menus are not, however, HyperCard objects either, because like windows, they do not have a script and can’t respond to messages. (Messages and scripts are described earlier in this chapter.)
Menus are containers. They contain menu items. Menu items can send messages to HyperCard objects with menu messages. If a menu item has a menu message, the menu message is sent to the current card or specified object when the menu item is chosen.

HyperTalk allows you to perform the following actions on menus:

- create new menus in the HyperCard menu bar with the `create menu` command
- add menu items to menus with the `put` command
- change the behavior of menu items by modifying a menu item’s menu message with the `menuMsg` property
- change the style of menu item text with the `textStyle` property
- assign Command-key equivalents to menu items with the `commandChar` property
- assign checkmarks for menu items (to show they are chosen) with the `checkMark` and `markChar` properties
- determine the name of menus or menu items with the `name` property
- disable menus and menu items with the `disable` command
- enable menus and menu items with the `enable` command and `enabled` property
- delete menu items with the `delete` command (except in the Font, Tools, and Patterns menus)
- delete entire menus with the `delete` command (once deleted, they’re gone, unless you recreate them)
- reset the HyperCard menu bar with the `reset menubar` command (once reset, all custom menus created with the `create menu` command are deleted)

To find out how to refer to HyperCard menus and menu items with the HyperTalk language, see Chapter 5, “Referring to Objects, Menus, and Windows.”
Chapter Summary

Here is a summary of the material covered in this chapter:

■ HyperTalk controls the properties of HyperCard objects: buttons, fields, cards, backgrounds, and stacks.

■ HyperTalk also controls the properties of HyperCard windows and HyperCard menus, although these elements do not have scripts as objects do.

■ HyperCard objects interact by sending and receiving messages.

■ How an object responds to a message is specified by its script, which is written in HyperTalk or another HyperCard-compatible language like AppleScript.

■ Scripts are collections of message handlers and function handlers.
CHAPTER 3

The Scripting Environment

This chapter describes the environment for creating and editing the scripts of HyperCard objects. It also describes the built-in script debugger that is part of the scripting environment.

Getting to the Script

You can get to an object’s script through the Objects menu, shown in Figure 3-1. The Objects menu has five object Info items, one for each of the five types of objects: the buttons and fields belonging to the current card and background, the current card, its background, and the stack to which the current card belongs.

Figure 3-1   The Objects menu
CHAPTER 3
The Scripting Environment

You must be at level 5
The user level must be set to 5 for you to be able to look at scripts. To change the user level, choose Preferences from the Home menu and select Scripting on the Preferences card, or set the user level to 5 from the Message box with this statement:

```
set userLevel to 5
```

To edit the script of the current card, background, or stack, choose the appropriate Info menu item for the object whose script you want. This action brings up information about the object in an Info dialog box (see Figure 3-2). To open the object’s script, click the Script button in that object’s Info dialog box.

To get to the script of a button or field, first select the button or field (with the Button tool or Field tool), then choose the appropriate Info item from the Objects menu. It is not necessary to be working in the background to open the script of an existing background button or field. You must be working in the

---

Figure 3-2  Button Info dialog box

![Button Info dialog box](image)

---

Getting to the Script
background, however, to create new background buttons and fields. (Working in the background may also help you to select background buttons and fields, because when you’re in the background, HyperCard doesn’t display card buttons and fields.)

**Shortcuts**
To get to the Info dialog box of a button or field quickly, double-click the button or field with the Button or Field tool chosen.

To open a button script directly, hold down Command-Option while you click anywhere inside the object’s surrounding rectangle. To open a field script directly, hold down Command-Option-Shift while you click anywhere inside the object’s surrounding rectangle. To open the script of the current card, press Command-Option-C. To open the script of the current background, press Command-Option-B. To open the script of the current stack, press Command-Option-S. ♦

You can close a script by choosing the Close Script command in the File menu, by pressing Command-W, or by clicking the close box in the upper-left corner. You can save a script by choosing the Save Script command in the File menu or pressing Command-S. To close the script without saving changes, press Command-period.

**Shortcut**
To save and close a script quickly, press the Enter key. ♦

---

**The Script Editor**

The HyperCard script editor lets you create and modify handlers in an object’s script. You can have more than one script open at a time. The scripts may be from within the same stack or from different stacks (see Figure 3-3). The number of possible open scripts depends on the available memory in the Macintosh computer. When you work on a script, you are working in a script window. You can resize script windows and drag them to any position on the screen.
CHAPTER 3

The Scripting Environment

Figure 3-3 Script editor window

<table>
<thead>
<tr>
<th>Script language: HyperTalk</th>
</tr>
</thead>
</table>

Script from background: Home Cards

**THE MESSAGE HANDLERS IN THIS SCRIPT:**
openStock, closeStack, openBackground, closeBackground, suspendStack,
resumeStock, newCard, deleteCard, renameCard, getCARDname,
renumberCards, reorderCards, adjustHandPercent, newLink, makeHandCCButton
drag&Button, updateAPath

**THE FUNCTION HANDLERS IN THIS SCRIPT:**
checkStr, selectHDFile, getWCPathName, manErrorScript, writeHCScript
stringsInContainer

--- Open,Pause Scripts ------------------------

on openStock
  global WhichHomeCard
  show menuBar -- needed for the "Home" menu
  if WhichHomeCard is not empty then go to WhichHomeCard
  -- remove any similar menus
  if there is a menu "Home" then delete menu "Home" -- –
  pass openStack -- so others can use it
end openStack

on closeStack
  -- Requires handler: checkForHomeMenu
  checkForHomeMenu
  pass closeStack -- so others can use it
end closeStack

on openBackground
  -- Requires handler: menuStatus
  menuStatus true, true
  pass openBackground -- so others can use it
end openBackground

**Note**
The script editor is implemented as an external window; you can replace it with a custom script editor that you define as an external command. See Appendix A, “External Commands and Functions,” for information on external commands and external windows; see the scriptEditor property in Chapter 12, “Properties,” for how to change to a custom script editor. ♦
CHAPTER 3

The Scripting Environment

While a script window is active, the script editor menu bar, which includes a menu called Script, is accessible (see Figure 3-4). The commands in the Script menu are described later in this chapter.

Figure 3-4  Script menu

When you have a script window open, you can still use the regular HyperCard menu bar by making a card window, rather than a script window, active. If any part of a card window is visible, you can make it active by clicking it. You can also use the Next Window command from the Go menu or press Command-L to bring a card window to the front.

Manipulating Text

Many standard text editing features are available in the script window. You can use the arrow keys to move the text insertion point around in the script. If your script extends beyond the right border of the script window, you can scroll horizontally by using the scroll bar at the bottom of the script window. You can save the current script with Command-S and close it with Command-W. You can also print the current script with Command-P.
The mouse manipulates an I-beam pointer with which you can place an insertion point or select text. You can double-click to select a word or triple-click to select a whole line. You can perform cut, copy, and paste operations using Command-X, Command-C, and Command-V, respectively. The selection that you’ve cut or copied remains in the Clipboard until you cut or copy again, in case you want to paste the material more than once. You can also paste it into a field as regular text or on a card or background as Paint text. You can undo a cut, copy, clear, or paste or any typing operation with Command-Z or by choosing Undo from the Edit menu.

You can change the font or size in which script text is displayed with the properties *scriptTextFont* and *scriptTextSize*. See Chapter 12, “Properties,” for more information about these properties.

Searching for Text

The Find command in the Script menu is different from the HyperCard Find command in the regular HyperCard Go menu. If you choose Find from the Script menu (or press Command-F), you get the dialog box shown in Figure 3-5. The script editor locates and selects the first occurrence, following the current insertion point, of a string you type into the Find field. You can search for a whole word or a partial word. If you don’t check Case Sensitive, Find ignores whether the letters in the word are uppercase or lowercase. If you check Case Sensitive, the case of each letter in the search string and the target string must match exactly. If you check Wraparound Search, Find searches for a string starting from the current insertion point to the end of the script, and then wraps around to the beginning of the script to continue the search. If Wraparound Search isn’t checked, Find locates a string only if it is after the current insertion point.

**Figure 3-5** Find dialog box

![Find dialog box](image_url)
Repeating Text

Choosing Replace in the Script menu or pressing Command-R brings up the dialog box shown in Figure 3-6. Replace locates and replaces the first occurrence, following the current insertion point, of a string you type into the Find field with the string you type into the “Replace with” field. For locating the string to be replaced, you have the same options as for the Find command: whole or partial word, case sensitive or not, and wraparound search or not. You can also specify that HyperCard replace the string, replace every occurrence of the string, or just find and highlight the string.

Figure 3-6 Replace dialog box

Entering Comments

You can put comments in your script by preceding the comment text with two hyphens (--). HyperCard ignores any text on a line after the double hyphen. If a comment wraps to the next line in a script, it must have a double hyphen at the beginning of that line, too. You can insert a double hyphen at the insertion point by choosing Comment from the Script menu, by pressing Command-hyphen, or by typing two hyphens.

You can remove the double hyphen from a line with Command–equal sign or by choosing Uncomment from the Script menu. For the Uncomment command to work properly, you must place the insertion point next to one of the hyphens, or select any part of the line, as long as the selection includes the hyphens.
CHAPTER 3

The Scripting Environment

The double hyphen is also useful when debugging or trying to improve the syntax in your scripts. You can precede statements in a handler with the double hyphen to prevent them from executing. To have HyperCard ignore an entire handler, you need to add the double hyphen to just the first line. This allows you to comment out a part of your script to see if that part is causing a problem.

Formatting Scripts

The HyperCard script editor indents control structures for you. It automatically indents all of the lines inside a handler structure when you finish typing a statement and press the Tab key, press the Return key at the end of the last line, or close its script window. (See Figure 3-7.) When if and repeat structures are nested inside each other or within handlers, the lines are indented further. (You can't nest handler structures inside each other or any other structure.)

Error checking

Automatic formatting provides some degree of error checking while you write a script: if you press the Tab key and the ending line in your handler isn’t flush with the left margin of the script editor window, you probably left something out or made a syntax error in a HyperTalk command. *

Figure 3-7 Nested control structures
CHAPTER 3

The Scripting Environment

Line Length and Script Size

The script editor doesn’t wrap lines that are too long to fit in a script window. Lines too long to fit in the script window simply extend out of sight. Line length is, however, limited to 255 characters. A single script cannot exceed 30,000 characters, including spaces, return characters, and other invisible characters. If you reach this limit, think about moving some of your handlers to another object, such as a hidden field or button, and send messages to it as required. If you don’t want statements to extend beyond the right boundary of the script window, you can break a single statement into multiple lines by pressing Option-Return where you want a line to break. This “soft” return appears in HyperCard scripts as a logical NOT symbol (¬). HyperCard treats lines broken in this way as single HyperTalk statements continuing to the next actual return character.

You can’t break a literal

You can’t put a “soft” return inside a quoted literal expression. (Chapter 6, “Values,” describes literals.) ◆

Script Editor Command Summary

Table 3-1 is a summary of the script editor commands you can invoke from the keyboard.

<table>
<thead>
<tr>
<th>Key press</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-A</td>
<td>Select all</td>
</tr>
<tr>
<td>Command-C</td>
<td>Copy selection to Clipboard</td>
</tr>
<tr>
<td>Command-D</td>
<td>Set or clear temporary checkpoint at selected line</td>
</tr>
<tr>
<td></td>
<td>(for debugger)</td>
</tr>
<tr>
<td>Command–equal</td>
<td>Uncomment selected line</td>
</tr>
<tr>
<td>sign</td>
<td></td>
</tr>
<tr>
<td>Command-F</td>
<td>Display Find dialog box</td>
</tr>
</tbody>
</table>

continued
### Chapter 3
The Scripting Environment

#### Table 3-1 Script editor command summary (continued)

<table>
<thead>
<tr>
<th>Key press</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-G</td>
<td>Find again</td>
</tr>
<tr>
<td>Command-H</td>
<td>Find the string currently selected elsewhere in the script</td>
</tr>
<tr>
<td>Command-hyphen</td>
<td>Comment selected line</td>
</tr>
<tr>
<td>Command-K</td>
<td>Check syntax (enabled only when <code>scriptingLanguage</code> is <code>AppleScript</code>)</td>
</tr>
<tr>
<td>Command-L</td>
<td>Next window to front</td>
</tr>
<tr>
<td>Command-Option-B</td>
<td>Open the script of the current background</td>
</tr>
<tr>
<td>Command-Option-C</td>
<td>Open the script of the current card</td>
</tr>
<tr>
<td>Command-Option-S</td>
<td>Open the script of the current stack</td>
</tr>
<tr>
<td>Command-P</td>
<td>Print script or selection</td>
</tr>
<tr>
<td>Command-period</td>
<td>Close script without saving changes</td>
</tr>
<tr>
<td>Command-R</td>
<td>Display the Replace dialog box</td>
</tr>
<tr>
<td>Command-S</td>
<td>Save script</td>
</tr>
<tr>
<td>Command-T</td>
<td>Replace again</td>
</tr>
<tr>
<td>Command-V</td>
<td>Paste Clipboard contents at insertion point</td>
</tr>
<tr>
<td>Command-W</td>
<td>Close script</td>
</tr>
<tr>
<td>Command-X</td>
<td>Cut selection to Clipboard</td>
</tr>
<tr>
<td>Command-Z</td>
<td>Undo last operation</td>
</tr>
<tr>
<td>Enter</td>
<td>Save changes and close script</td>
</tr>
<tr>
<td>Option-click</td>
<td>Set or clear temporary checkpoint at selected line (for debugger)</td>
</tr>
<tr>
<td>Option-click close box</td>
<td>Close all open scripts</td>
</tr>
<tr>
<td>Option-Return</td>
<td>Carry statement onto new line (&quot;soft&quot; return ¬)</td>
</tr>
<tr>
<td>Tab</td>
<td>Format script</td>
</tr>
</tbody>
</table>
This section describes the HyperTalk script debugger. The script debugger is integrated with the script editor to provide a set of easy-to-use debugging tools. You may find it easier to understand the features of the script debugger after learning more about HyperTalk, so you may want to skip this section and return to it later.

The HyperTalk script debugger has the following features:

- integration with the script editor for setting checkpoints within scripts
- debugging tools in the Debugger menu
- two windows for watching the progress of a script while it executes or while you step through the statements in a handler

In HyperTalk, a checkpoint serves the same purpose as a breakpoint in traditional development environments. You set a checkpoint at the location in a script at which you want to enter the debugger. You can also enter the debugger with the Command-Option-period key combination anytime while a script is executing. For example, you can get to the debugger by pressing Command-Option-period right after clicking a button that goes to another stack.

When HyperTalk enters the debugger, it pauses execution of the script, displays the script in a window, puts a box around the next line of the script to be executed, and displays a Debugger menu at the right end of the HyperCard menu. The Debugger menu, shown in Figure 3-8, has several menu items you can use to debug your scripts.

You can step through the remaining statements in the script with Command-S or by choosing Step from the Debugger menu. Each step executes the statement with a box around it, then moves the highlight to the next statement in that handler.
You can step into the trail of subhandlers with Command-I or by choosing Step Into from the Debugger menu. Step Into allows you to follow execution among multiple handlers when one handler calls another. When you choose Step Into and a message is sent from the currently executing handler, you go to the location in the current object’s script or any object’s script in the message-passing hierarchy that has a handler for that message. That message handler becomes a subhandler to the originally executing handler. You can then continue to step through the subhandler until its completion. After completion of the subhandler and any of its subhandlers, you go back to the line in your original handler following the statement that sent the message to the first subhandler. Subhandlers could be in any object script within the current message-passing hierarchy. (See Chapter 4, “Handling Messages,” for information about handlers calling handlers and the message-passing hierarchy.)

You can trace the current handler to completion by choosing Trace from the Debugger menu. When you choose Trace, HyperCard executes each line in the current handler. Use Command-T or choose Trace Into in the Debugger menu to execute each line of the current handler including all of the subhandlers until the script’s completion without having to manually step through the handlers. You can set the amount of time HyperCard waits between execution of the lines in a handler during a trace by choosing Trace Delay in the Debugger menu. The
trace delay value can also be set with the global property `traceDelay`. See Chapter 12, “Properties,” for information about the `traceDelay` property.

You can exit the debugger with Command-G or by choosing Go from the Debugger menu.

### Setting Checkpoints

You can set temporary and permanent checkpoints in a script. To set a temporary checkpoint in a HyperTalk script, set the insertion point anywhere in a line of a handler at which you want to enter the debugger and choose Set Checkpoint from the Script menu or press Command-D. You can also click anywhere in the line while pressing the Option key. A temporary checkmark appears in the margin to the left of the chosen location. To remove a temporary checkpoint, perform any one of the previously mentioned operations on the line with the checkmark. You can clear all the checkpoints in a script by clicking any checkpoint in the script while pressing Shift-Option.

You can have up to 16 temporary checkpoints per script in a maximum of 32 scripts. Temporary checkpoints are not saved with the script when you quit HyperCard.

You set permanent checkpoints in a script by inserting the HyperTalk statement `debug checkpoint` anywhere within a handler. There is no practical limit to the number of permanent checkpoints in a script. Permanent checkpoints are permanent in that they are saved with the script—they can be removed by deleting the `debug checkpoint` statement.

Checkpoints are ignored by HyperCard when `userLevel` is set lower than Scripting (user level 5).

### HyperTalk Debugger Windows

The debugger windows are named the Message Watcher and the Variable Watcher. You can display one or both of these windows in two ways. They can be called by HyperTalk commands in a script or the Message box, or they can be displayed by choosing Variable Watcher or Message Watcher from the Debugger menu while you’re in the debugger environment.
CHAPTER 3

The Scripting Environment

Message Watcher

The Message Watcher, shown in Figure 3-9, is an external window. It appears as a floating movable window that displays both HyperTalk messages and XCMD-generated messages as they are sent. You can display the Message Watcher window with either of these HyperTalk statements:

show Message Watcher
set the visible of window "Message Watcher" to true

When you’re in the debugger, you can also display the Message Watcher window by choosing Message Watcher from the Debugger menu.

Figure 3-9  The Message Watcher window

The Message Watcher window is always in front of the active window, which may be a card window or script window. (If you call the Message box, it appears in front of the Message Watcher.) You can close the Message Watcher window by clicking the close box or with either of these statements:

hide Message Watcher
set the visible of window "Message Watcher" to false

When you’re in the debugger, you can close the window by choosing Message Watcher from the Debugger menu.

When “Hide unused messages” is checked, the Message Watcher displays only those messages that are handled by scripts. If “Hide unused messages” is not checked, you see all messages that are sent whether or not they are intercepted...
CHAPTER 3

The Scripting Environment

by a handler. Each message that isn’t handled in the message-passing hierarchy
is displayed in parentheses. For example, (mouseWithin) displayed in the
list indicates that there was no handler for the mouseWithin message in the
message-passing hierarchy.

When “Hide idle” is checked, the Message Watcher doesn’t display idle
messages. If “Hide idle” isn’t checked, idle messages are displayed.

The Message Watcher stores the last 150 lines of messages. Older messages are
removed as the new messages fill the buffer.

Variable Watcher

The Variable Watcher, shown in Figure 3-10, is an external window. It appears
as a floating window that displays the HyperTalk global variables and local
variables set by the current script. (Variables are explained in Chapter 6,
“Values.”) Variables are displayed in a two-column format with the variable
name in the left column and the current values in the right column. You can
display the Variable Watcher window with either of these HyperTalk
statements:

show Variable Watcher
set the visible of window "Variable Watcher" to true

When you’re in the debugger, you can also display it by choosing Variable
Watcher from the Debugger menu.

Figure 3-10  The Variable Watcher window
CHAPTER 3

The Scripting Environment

The Variable Watcher window is always in front of the active window, which may be a card window or a script window. You can close the Variable Watcher window by clicking the close box or with either of these statements:

```
hide Variable Watcher
set the visible of window "Variable Watcher" to false
```

When you’re in the debugger, you can close the window by choosing Message Watcher from the Debugger menu.

You can temporarily modify the values of global and local variables to see how they affect your stack application or scripting environment with the Variable Watcher. Drag the Variable Watcher size box to expand the window so that you can see the area below the thick horizontal line in the window. Click the variable name that you want to modify. The variable value is highlighted and placed in the field below the horizontal bar, as shown in Figure 3-11. You can edit the variable value by typing into the field. When you have finished changing the value, press the Enter key. The value for that variable is changed.

![Figure 3-11 A selected variable in the Variable Watcher window](image)

Custom Message Watcher and Variable Watcher XCMDs

The Message Watcher and Variable Watcher are implemented as external windows, so they can be replaced with custom external commands written in a programming language such as Pascal.
To replace either the Message Watcher or Variable Watcher window with your own custom tools, set the global property `messageWatcher` or `variableWatcher` to the name of your tool with a HyperTalk statement like:

```
set variableWatcher to "MyToolWindow"
```

For more information about creating external windows, see Appendix A, “External Commands and Functions.”

### Debugger Command Summary

Table 3-2 is a summary of the debugger commands you can invoke from the keyboard.

<table>
<thead>
<tr>
<th>Key press</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-A</td>
<td>Stop the handler</td>
</tr>
<tr>
<td>Command-D</td>
<td>Clear the current checkpoint</td>
</tr>
<tr>
<td>Command-G</td>
<td>Go back to application</td>
</tr>
<tr>
<td>Command-I</td>
<td>Step into and follow the path through any subhandlers</td>
</tr>
<tr>
<td>Command-Option-period</td>
<td>Open the debugger while a script is running</td>
</tr>
<tr>
<td>Command-period</td>
<td>Stop the handler</td>
</tr>
<tr>
<td>Command-S</td>
<td>Step to the next line in current handler; step through any subhandler</td>
</tr>
<tr>
<td>Command-T</td>
<td>Trace the handler and its subhandlers to completion; same as Step Into, but user interaction isn’t required</td>
</tr>
</tbody>
</table>
Chapter Summary

Here is a summary of the material covered in this chapter:

- You can create and edit scripts with the HyperCard script editor.
- You can debug scripts with the built-in debugger.
This chapter explains how HyperCard objects send and receive messages and how HyperCard executes scripts.

The HyperCard Environment

HyperCard provides the environment in which HyperTalk scripts execute. The HyperCard environment consists of objects connected by a message-passing hierarchy and the HyperTalk language through which they communicate.

Although you could write a stand-alone program in a single HyperTalk script, you would not be making use of the power and flexibility of the HyperCard environment. Instead, you use HyperTalk to define the ways in which objects interact with each other and with the user.

HyperCard is user oriented. When using HyperCard, the user opens and closes cards, reads and changes text in fields, draws pictures on cards, and so on. HyperCard constantly sends messages to objects in response to these actions (and to the user’s inactivity when doing nothing), and the objects in turn respond with other messages and other actions. The basic purpose of HyperTalk scripts is to enable objects to handle those messages and to specify succeeding actions by sending further messages.

Most of the time, scripts carry out specific actions for the user: setting properties of objects, going to other cards, and so on. HyperTalk can do automatically almost everything the user can do manually with the mouse and keyboard.
CHAPTER 4

Handling Messages

Sending Messages

All HyperCard actions are initiated by messages sent to objects. Messages are sent to objects in four ways:

- An event (such as a mouse click or a key pressed on the keyboard) can cause HyperCard to send a system message.
- Handler statements (other than keywords) are sent as messages when a handler executes.
- HyperCard sends the contents of the Message box as a message when the user presses Return or Enter.
- HyperCard sometimes sends a message when it executes a command.

System Messages

HyperCard sends system messages constantly in response to events in the Macintosh environment. For example, if you move the pointer so that it’s over a button on the screen, as soon as the pointer enters the button’s rectangle, HyperCard sends the message `mouseEnter` to the button. As long as the pointer remains inside the button rectangle, HyperCard continuously sends the message `mouseWithin` to the button. As soon as you move the pointer outside the button area, HyperCard sends the message `mouseLeave` to the button.

HyperCard sends other system messages when you press certain keys on the keyboard, close a field, choose a menu item, or quit HyperCard. When you open a card, HyperCard sends the message `openCard` to the card itself; when you leave the card, it sends `closeCard`. Similar messages are sent to cards when their backgrounds and stacks are opened and closed. If nothing at all is happening, HyperCard continuously sends the message `idle` to the current card.

One of the most commonly used messages is `mouseUp`. Buttons often contain handlers that respond to the `mouseUp` message; the `mouseUp` message is sent to a particular button when you click it. (HyperCard actually sends two messages to a button when it is clicked: `mouseDown` and `mouseUp`. The
CHAPTER 4

Handling Messages

MouseDown and mouseUp messages are sent only if you release the mouse button with the pointer over the same screen button it was over when you pressed it down.

HyperCard also sends mouse messages to a locked field when you click it. If the field isn’t locked, mouseDown and mouseUp aren’t sent—the click opens the field for text editing and HyperCard sends the message openField to the field. (You can send mouse messages to an unlocked field, however, by holding down the Command key while you click the field.)

Clicking outside all buttons and fields sends mouseDown and mouseUp directly to the current card.

Chapter 8, “System Messages,” describes all of HyperCard’s system messages.

Statements as Messages

When a handler executes, its statements are sent as messages, first to the object that contains that handler, then to succeeding objects in the message-passing hierarchy (described later in this chapter). When an object gets a message it can handle—that is, for which it has a handler in its script—the statements contained in the handler are in turn sent as messages. When all statements in the handler (and in any other handlers invoked along the way) have executed, the action stops.

Message Box Messages

When you type something into the Message box and press Return or Enter, HyperCard does one of these things: evaluates a valid expression and puts the result into the Message box, sends what you typed as a message to the current card, or sends a message to another destination if you use the send command. (See Chapter 7, “Expressions,” for an explanation of evaluating expressions.)

You use send to direct a message to a specific object rather than sending it to the current card. Send is one of the HyperTalk keywords. You can use the keywords do, if...then...else, and send in the Message box. If you try to use a keyword other than these in the Message box, HyperCard displays an error dialog box. Table 4-1 contains all of HyperTalk’s keywords.

Chapter 9, “Control Structures and Keywords,” contains explanations of HyperTalk’s keywords.
CHAPTER 4

Handling Messages

Table 4-1: HyperTalk’s keywords

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>do</td>
<td>next</td>
</tr>
<tr>
<td>else</td>
<td>on</td>
</tr>
<tr>
<td>end</td>
<td>pass</td>
</tr>
<tr>
<td>exit</td>
<td>repeat</td>
</tr>
<tr>
<td>function</td>
<td>return</td>
</tr>
<tr>
<td>global</td>
<td>send</td>
</tr>
<tr>
<td>if</td>
<td>then</td>
</tr>
</tbody>
</table>

Messages Resulting From Commands

HyperCard sometimes sends a system message to the current card while executing a command. For example, when you create a card with the New Card menu command, HyperCard sends the message `newCard` to the card as soon as it’s created; when you delete a card, it sends `deleteCard`. Similar messages are sent when other objects are created and deleted. These messages are among the results of commands executing, rather than commands themselves—they are like announcements of what is happening.

External commands can send messages

Experienced programmers can write definitions for new commands in development languages such as Pascal, C, and 68000 assembly language. Such external commands act much like built-in HyperTalk commands. External commands can send messages to the current card when they execute. See Appendix A, “External Commands and Functions,” for information about external commands.
Receiving Messages

As senders and receivers of messages, objects all work exactly the same way. Every object has a script, and the type of object makes no difference to the execution of its handlers.

**How objects differ**

As elements of the HyperCard user interface, objects differ according to their function: buttons share a set of properties or characteristics that determine how they look and act; fields also share a set of properties, but it is different from the set of button properties. (See Chapter 12, “Properties,” for a description of object properties.)

When a message is sent to an object, HyperCard checks the object’s script for a handler whose name—the second word on the first line of the handler—matches the message name—the first word of the message. If it finds a match, it executes each statement in the handler. (See Figure 4-1.) After the handler has run, the message is sent no further, unless it is explicitly passed with the **pass** keyword (discussed in Chapter 9).

**Figure 4-1**  Handler that responds to message **openStack**

```plaintext
on openStack
  global userName
  if userName is empty then
    ask "Please enter your name"
  if it is not empty then put it into userName
  end if
  pass openStack
end openStack
```

If handler name matches message name, then lines in that handler execute.
CHAPTER 4

Handling Messages

If the object has no handler for the message, the message passes to the next object in the hierarchy, and the process repeats. The message-passing hierarchy is explained in the next section.

If no object in the hierarchy has a handler matching a message name, HyperCard looks for a command by that name. Commands are like built-in handlers that cause some action to take place; mouseUp and most other system messages have no built-in handlers and cause no action. If a message gets all the way through the hierarchy and is not a system message or a command, HyperCard displays an error dialog box with the words *Can't understand* followed by the name of the message.

**External commands can be in stacks**

External commands can exist in stack files, as well as in the HyperCard application itself. See Appendix A, “External Commands and Functions,” for information about external commands. ♦

---

**Message-Passing Hierarchy**

Messages are passed to objects according to a **message-passing hierarchy**. The message-passing hierarchy determines the path by which messages are passed from one object to another: buttons and fields are at the same level, followed (in order) by card, background, stack, the Home stack (the one stack that HyperCard requires), and HyperCard.

HyperCard allows you to add stacks to the message-passing hierarchy so you can use their scripts as shared-code libraries. More about the user-definable message-passing hierarchy is explained later in this chapter.

**Where Messages Go**

The position of an object in the message-passing hierarchy determines whether or not the object receives a given message and where subsequent messages that the object sends go. Most system messages are initially sent by HyperCard to the current card, as shown in Figure 4-2.
CHAPTER 4

Handling Messages

Figure 4-2  Message-passing hierarchy

Message Pass - Passing Hierarchy

- HyperCard
- Home stack
- Stack
- Background
- Card
- Buttons and fields
- Menu
- New Button and other events
- Message box
- New Card and other events
- Keyboard
- Mouse
CHAPTER 4

Handling Messages

Messages to Buttons and Fields

Any mouse message (for example, `mouseEnter`) is sent initially to the topmost button or field, if there are any, under the pointer. Any buttons or fields that are layered farther under the one initially receiving the message are ignored. Figure 4-3 shows layered buttons and fields. If the topmost button or field doesn’t have a handler for the mouse message, the message is passed to the current card.

![Layered buttons and fields](image)

Background buttons and fields come before cards

HyperCard first sends mouse messages to the topmost button or field under the pointer, whether the button or field belongs to the card or the background, before passing the message on to the card. Background buttons and fields, however, are always farther away than card buttons and fields.

Other than mouse messages, the only system messages that are sent first to buttons are `deleteButton` and `newButton`; system messages sent first to fields are `closeField`, `commandKeyDown`, `deleteField`, `enterInField`, `exitField`, `keydown`, `newField`, `openField`, `returnInField`, and `tabKey`. The entry point in the hierarchy for all other system messages is the current card.

For a complete list of all system messages, see Chapter 8, “System Messages.”
The current hierarchy consists of the buttons and fields belonging to the current card and its background, the card and background themselves, their stack, the Home stack, and HyperCard. System messages and those typed directly into the Message box always traverse the current hierarchy. Messages sent from executing handlers traverse the hierarchy in which their containing object belongs—in most cases, the current one. Figure 4-4 shows how a message traverses the current hierarchy.

When a handler executes, HyperCard sends each statement as a message, unless it begins with a keyword. It sends the message first to the object containing that handler, as shown in Figure 4-5. If that object doesn't have a
CHAPTER 4

Handling Messages

handler for the message, the message is passed down the object hierarchy; if none of the succeeding objects has a handler for it, the message ends up at HyperCard itself.

**Function calls use the message-passing hierarchy**

Function calls work like messages in the way they traverse the object hierarchy. When you make a function call with the syntax that uses parentheses, HyperCard looks in the script of each object in the hierarchy for a matching function handler. If none is found, the function call is passed to HyperCard itself. See Chapter 11, “Functions,” for information about functions.

---

**Figure 4-5** Command sent as a message

![Diagram showing the message-passing hierarchy](image-url)
CHAPTER 4
Handling Messages

The Target

The object to which the message is first sent is the **target**. If HyperCard finds a handler in the target that matches the message name, the handler’s statements start executing. If, however, the target has no matching handler, the message is passed down the hierarchy. HyperCard may find a matching handler in another object, which then begins executing, as shown in Figure 4-6.

The function `the target` returns the name of the original target, so that handlers in succeeding objects can determine where a message was originally sent. In Figure 4-6, although the executing handler is in the background script, `target`, used in the background handler, results in identifying the new button that originally received the system message `newButton`.

Figure 4-6  The target

```plaintext
| Buttons and  | Cards          | Backgrounds |
| fields      |                |             |
|            | on newButton   | on newButton|
|            | set autoHilite | set autoHilite|
|            | of the target  | of the target|
|            | to true        | to true     |
|            | end newButton  | end newButton|
```

HyperCard
The User-Defined Hierarchy

HyperCard allows you to add stacks to the message-passing hierarchy, thereby extending the current hierarchy. Stacks added to the message-passing hierarchy act as shared code libraries. The code that they share is their script.

HyperTalk’s message-passing hierarchy always includes the Home stack and HyperCard itself. Scripts in any stack can call handlers in the script of the Home stack, and external commands in the resource fork of either the Home stack or HyperCard because they are always in the hierarchy.

When a stack is added to the hierarchy, it is inserted between the current stack and the Home stack, as shown in Figure 4-7. That stack’s script and all of the stack’s resources can now be shared with objects higher in that hierarchy.

To add a stack to the message-passing hierarchy, use the HyperTalk command `start using` in a handler like this:

```hyperTalk
on openStack
    start using stack "HD:Trees"
end openStack
```

After the `start using` command in the example handler is executed, the handlers in the script of the stack `Trees` and any of its external commands and resources are available for use by `Flora`. 

```hyperTalk
```
Figure 4-7  One stack added to the message-passing hierarchy

The statement `start using stack "HD:Trees"` inserts the stack Trees in the message-passing hierarchy.
Each additional stack that is added to the hierarchy is inserted after the current stack, as shown in Figure 4-8. The maximum number of stacks you can have in a user-defined hierarchy is 16. If a stack that is already being used is used again with the start using command, it is moved in the hierarchy to the location just before the last stack that was inserted with the start using command.

**Figure 4-8** Two stacks added to the message-passing hierarchy

- Buttons and fields
- Cards
- Backgrounds
- Current stack
- Last added stack
- Added stack
- Home stack
- HyperCard
The current hierarchy described earlier in this chapter isn’t changed when you create a user-defined hierarchy—it is extended with the new stacks added to the hierarchy. Messages still traverse the hierarchy in the same way: they go down from the buttons and fields belonging to the current card and its background, to the card and background, to their stack, to any added stacks, to the Home stack, and finally to HyperCard.

**Note**

If you have a handler with the same name in more than one stack in a user-defined hierarchy, the handler highest in the hierarchy is executed when a message that calls that handler is sent. So if there’s a handler for that message in the current stack, that’s the one that gets executed. If there are handlers with the same name in two stacks that have been added to the hierarchy, the one in the most recently added stack is executed because it’s higher in the hierarchy.

The names of the stacks in the current hierarchy are stored in the global property `stacksInUse` in the form of a return-separated list of the stacks in the order in which they receive messages. If you create a card field, and then use the statement `put the stacksInUse into card field "Myfield"` in the Message box, it will return the list of stack names in the card field you created. Each stack that is placed in the hierarchy with the `start using` command is on a separate line in the field. If no stacks are added to the hierarchy, `stacksInUse` returns empty.

You can remove a stack from the user-defined hierarchy with the command `stop using` in the following statement:

```
stop using stack "Trees"
```
The stack is removed from the hierarchy, as shown in Figure 4-9.

**Figure 4-9**  Removing a stack from the message-passing hierarchy

The statement `stop using stack "Trees"` removes the stack Trees from the message-passing hierarchy.
The Dynamic Path
When a message is traversing the hierarchy of a card different from the current one, HyperCard inserts a dynamic path into the static path the message normally follows. The static path is the route defined by an object’s own hierarchy. For example, a card passes messages to its own background, the background passes them to its own stack, and so on. When that hierarchy is not the one stemming from the current card (the one currently active), HyperCard passes messages through the current card’s hierarchy as well—that’s the dynamic path.

Examples of situations in which a message traverses a hierarchy different from the current one, invoking the dynamic path, are

- when an executing handler contains a command that takes you to another card (such as go or a command to create or delete the current card)
- when you use the send keyword to send a message to an object not in the current hierarchy

When any message that has not been received by a handler reaches the stack, HyperCard checks to see if the current card is in a different hierarchy. If so, HyperCard passes the message to the current card, and it traverses the current card, background, and stack before it passes to the Home stack.

If any handler receives the message and passes it explicitly with the pass keyword, HyperCard does not invoke the dynamic path unless the current hierarchy is in a different stack from the static path. If either of the hierarchies is in the Home stack, the message is not passed again to the Home stack.

The Go Command and the Dynamic Path
Figure 4-10 and Figure 4-11 show how a handler containing a go command invokes the dynamic path.
In Figure 4-10, the mouseUp handler executes the statement `beep 2`, which is sent as a message along the current hierarchy beginning with the button containing the handler. After the `go` command executes, the current card changes. Nonetheless, the button’s mouseUp handler continues to execute, sending subsequent statements as messages through its own hierarchy, in this case the `beep 3` statement. In addition, however, HyperCard now sends messages to the card, background, and stack of the new current hierarchy, as shown in Figure 4-11.
The Send Keyword and the Dynamic Path

You can use the send keyword to direct a message to

- any object in the current stack
- any other stack on any disk or file server accessible to your Macintosh computer (but not any individual object in those stacks, unless you go to that stack first); the stack need not be in the current hierarchy
- HyperCard itself
For example, you can type the following statement into the Message box:

```
send "greetings" to stack "Birds"
```

HyperCard looks in the script of the object to which the message is sent (in this case, stack "Birds") for a matching handler, just as if it were in the current hierarchy. If the matching handler isn’t found (in this case, a handler named greetings), the message goes down the hierarchy stemming from the object to which it was sent (that is, from stack "Birds"). If the target of the send command is an object other than the current one, HyperCard invokes the dynamic path.

Figure 4-12 shows the path of a message directed with the send keyword.

The executing handler, the one currently in control, need not be in the hierarchy belonging to the current card. Which handler has control is determined solely by which object receives a message.

---

**Figure 4-12** Using the `send` keyword

```
send "greetings" to stack "Birds"
```

Cards

Backgrounds

Stacks

Home stack

HyperCard

---
For details about the send keyword, see Chapter 9, “Control Structures and Keywords.”

** Handlers Calling Handlers **

When a handler executes, HyperCard sends each statement as a message first to the object containing the executing handler so that other handlers in the same script, as well as those in any other script lower in the message-passing hierarchy, can be used as subroutines. A handler can also call itself, which is known as **recursion**.

** Subroutine Calls **

You can use handlers in HyperCard the way you use procedures or subroutines in other languages. You invoke a subroutine call in HyperTalk by executing a statement that begins with the name of a handler. That name is sent as a message, first to the object that contains the executing handler, then along the current object hierarchy.

You can include a subroutine in a script by writing a handler in the same script (or any other script lower in the object hierarchy) with whatever name you’d like to call it. In the following example, the handler **greetings** is defined in the same script as the one from which the message **greetings** is sent:

```hypercard
on mouseUp
  greetings
end mouseUp

on greetings
  Put "You've just been drafted!" into the Message box
end greetings
```

When HyperCard executes the statement consisting of the subroutine handler name and a match is found between the name and its handler, control passes to the subroutine handler. After it has finished executing, control passes back to the calling handler. But it’s entirely possible for the subroutine handler to issue a similar message, beginning execution of a third handler. The third handler
must finish executing before control passes back to the second handler, which in turn must finish executing before control passes back to the first. The execution of a handler that has invoked another handler is suspended until the handler it has called finishes executing.

**Stopping execution**
A handler can avoid giving control back to pending handlers by executing the `exit to HyperCard` keyword statement. You can interrupt an executing handler at any time (and bypass pending handlers) by pressing Command-period.

Any handler can act as a subroutine for any other handler. The called handler either has to be in the same script or in a script lower in the object hierarchy. However, you can also use the `send` keyword to send the message (the subroutine handler name) directly to the object that contains the handler. (See Chapter 9, “Control Structures and Keywords,” for details on using `send`.) Generally, handlers that act as subroutines are placed in the same script as the handlers that call them.

**IMPORTANT**
Handlers can’t be nested. That is, they can’t be defined with one inside another—a handler definition must not appear between the `on` statement and the `end` statement of another handler.

**Recursion**
If you need to repeat an operation over and over, you can have a handler call itself. In the following example, the handler `decrement` subtracts 1 from a number in the Message box until the number is reduced to 1 (a number must be in the Message box before you call the handler). To do the subtraction, the handler summons itself:

```plaintext
on decrement
    subtract 1 from the message box
    if the value of the message box > 1 then decrement
end decrement
```
Generally, subroutine calls and recursion don’t cause any problems. In fact, they are natural consequences of the good programming technique of separating scripts into functional units. However, HyperCard has a limit on the number of pending handlers. The actual number depends on the complexity of the handlers and other factors. It doesn’t matter whether a handler is invoking itself or another handler—either type of invocation causes another level of pending execution.

In particular, watch out for endless recursion, as in the following handler (if it were in a stack script or the script of every card):

```plaintext
on openCard
  go to next card
end openCard
```

The `go to next card` command results in an `openCard` message, so the handler recurses again and again, and you get an error dialog box. The HyperCard limit for such a recursion is limited by memory.

### Using the Hierarchy

Where you place a handler in the hierarchy determines when it will be called. All objects that are higher in the hierarchy can call handlers in objects lower in the hierarchy. Lower objects can’t call handlers in higher objects unless they use the `send` keyword. Messages that are sent when a statement in a handler executes always go first to the object containing the executing handler. Then they traverse the hierarchy stemming from that object until they find a matching handler or reach HyperCard itself. Therefore, the farther down the hierarchy a handler is placed, the greater the number of objects that can pass messages to it.

### Sharing Handlers

In effect, every object has access to the handlers of all the objects lower than it in the hierarchy, which also includes the handlers in stacks put into the message-passing hierarchy with the `start using` command. If you want
CHAPTER 4
Handling Messages

every card in a stack to have a certain capability (that is, to respond to a certain message), you put the appropriate handler in the stack script. Every card can use the handler by passing the message down to the stack.

Figure 4-13 and Figure 4-14 show the effect of placing a handler at different positions in the hierarchy. The example handler responds to the message moveOn (the message name is for example only). The handler takes you to the next card:

```
on moveOn
  go to next card
end moveOn
```

**Figure 4-13** Handler in a card script

When the message is sent to cards 1–4, it goes through the hierarchy to HyperCard without being handled.
You can place the handler in the script of one card, as in Figure 4-13. Then, if you send `moveOn` from the Message box, you invoke the handler and go to the next card only if the current card is the one with the handler. If the current card is not the one with the handler, however, the `moveOn` message produces an error.

In Figure 4-14, the handler is in the script of the stack, so the handler is invoked by sending `moveOn` to any card in the stack.
Intercepting Messages

You can also make any card you want an exception in the way it responds to a given message, without affecting the other cards in the stack, by putting a special handler for the message in that card’s script: you write two different handlers with the same message name—one in the stack script and one in the card script. Then, for that same message, if the message comes through that particular card, the card’s handler runs; from any other card, the stack’s handler runs.

For instance, in the previous example, putting the handler in the stack script caused the message moveOn to take you to the next card from any card in the stack:

```hypercard
on moveOn
    go to next card
end moveOn
```

But if you want the last card in the stack to be an exception, from which the message moveOn takes you back to the Home stack, put the following handler in the last card’s script:

```hypercard
on moveOn
    go to stack "home"
end moveOn
```

Figure 4-15 illustrates this example of one object intercepting a message.

A handler can intercept a HyperTalk command

In the same way that you can give one card a unique way of handling a message that would ordinarily be handled in the background or stack script, you can write a handler with the same name as a HyperTalk command and place it anywhere in the hierarchy. But remember that your handler is the one that will ordinarily run in response to the command message, not HyperCard’s built-in one. HyperTalk functions can be redefined in a similar manner, and the same warning applies.
Parameter Passing

When a HyperTalk message is sent, the first word is the message name. For example, in the message

```
searchScript "WildCard","Help"
```

the message name is `searchScript`. Any other words (or characters) are the **parameters**. In the example, the parameters are "WildCard" and "Help". Each receiving object in the hierarchy looks for a message handler with a matching name. If the object finds a matching handler, the parameters are passed into that handler.
CHAPTER 4

Handling Messages

Parameters are passed into handlers as a list of comma-separated expressions. (Chapter 7, “Expressions,” describes expressions.) These expressions are evaluated before the message is sent by the current object and, when the message is received by the receiving object, placed into a list of comma-separated parameter variables appearing on the first line of the matching handler definition. (See Figure 4-16.) That is, parameters are passed by value into handlers. In the searchScript handler example shown in Figure 4-16, the parameter variables pattern and stackName are replaced by the parameter values "WildCard" and "Help".

Parameter variables are local variables of the handler in which they appear. Parameter variables are also called formal parameters, to contrast them to the actual parameters, which are the parameter values passed to them.

Function handler parameters
HyperCard passes parameters into function handlers and message handlers in the same way, except that the syntax of the function call requires the parameters to be placed between parentheses. Placement of the parameter variables on the first line of function handlers is identical to that of message handlers. ♦

The value of the first expression in the message is placed into the first parameter variable in the handler, the value of the second expression into the second parameter variable, and so on. If there are more expressions in the message’s parameter list than there are parameter variables in the first line of the handler, the extra parameters are ignored. If there are more parameter variables than parameters, the extra parameter variables are given an empty value (equal to a string of zero length).

Passing parameters to redefined commands
HyperTalk command parameters are often more complex than a comma-separated list of expressions. Some built-in commands take parameters to which user-written handlers have no access. So, if you redefine a command, you may not be able to pass all of the parameters to your handler. ♦
Chapter Summary

Here is a summary of the material covered in this chapter:

- The HyperCard environment consists of objects related to each other in a hierarchy using HyperTalk as the means of communicating.
- Messages sent to objects initiate all HyperCard actions.
- Messages are generated by system events, executing handlers, statements typed into the Message box, and the execution of some commands.
- When an object receives a message, HyperCard tries to match the message name with a handler in the object’s script; if it finds a match, it executes the handler; otherwise it passes the message to the next object.
CHAPTER 4

Handling Messages

- The object hierarchy determines how messages are passed from one object to another.
- You can modify the message-passing hierarchy to use stacks as shared-code libraries.
- You can send a message directly to any object in the current stack, to another stack, or to HyperCard using the `send` keyword.
- A handler can initiate execution of another handler as a subroutine call.
- Every object can use the handlers of objects lower than it in the hierarchy by passing messages; conversely, an object can intercept a message to perform a different action.
- The values of a series of expressions following the first word of a message statement are passed to variables in the first line of the receiving handler.
CHAPTER 5

Referring to Objects, Menus, and Windows

This chapter explains how to refer to HyperCard’s objects, menus, and windows.

A HyperCard object has three characteristics:

- It can send and receive messages.
- It has properties, which are its defining characteristics, and one of those properties is its script.
- It has a visible representation on the Macintosh screen (although the object need not always be visible).

HyperCard menus, windows, and the menu bar share many of the characteristics of HyperCard objects, except that they don’t have the script property.

You refer to an object when you use the go command (to go to a particular card, background, or stack) or the send keyword (to send a message to a particular object), and when you want to manipulate an object’s properties. Fields are unique because they are HyperCard objects and are also sources of values (described in Chapter 6, “Values”).

You can think of HyperCard itself as an object, because it can send and receive messages and has global properties, including a “script” of built-in handlers or commands. When this book talks about objects, however, it usually refers to buttons, fields, cards, backgrounds, and stacks.

Names, Numbers, and IDs

You refer to objects using object descriptors. An object descriptor is formed by combining a generic name with its specific designation. Generic names are stack, card (abbreviated cd), background (abbreviated bkgnd or bg),
button (abbreviated btn), field, part (which refers to a field or button), or family (which refers to a group of buttons).

HyperCard considers all references to buttons, button families, and parts to be card buttons, card families, or card parts and all references to fields to be background fields unless you specify otherwise. For example, button "buttonName" and card button "buttonName" both refer to the card button, and field "fieldName" refers to the same field as background field "fieldName". To refer to background buttons, you must include background in the descriptor—for example, bkgnd button "buttonName". If you want to refer to card fields, you must include card—for example, card field "fieldName".

The only specific designation of a stack is its name. (See “Identifying a Stack,” later in this chapter.) The specific designation of all other objects (buttons, fields, backgrounds, and cards) can be the object’s name, number, or ID number. The unambiguous form of an object descriptor begins with an object’s generic name, immediately followed by its particular name, number, or ID number. (See Figure 5-1.)

---

**Figure 5-1** Card Info dialog box and descriptors for the same card

---

Card Info dialog box and descriptors for the same card

---

Card Name: Welcome to...
Card number: 1 out of 9
Card ID: 3915
Contains 0 card fields.
Contains 13 card buttons.

- Card Marked
- Don’t Search Card
- Can’t Delete Card

OK
Cancel

Card 1
First card
Card one
Card "Welcome to "
Card ID 3916

---

Names, Numbers, and IDs
Descriptor phrasing
Be careful to phrase descriptors so that they mean what you intend. For example, using a descriptor such as card field id 7, you could mean that the name of the card is in the background field with ID number 7, or you could be referring to the card field with ID number 7. HyperCard assumes that you’re referring to the card field. If you want HyperCard to get the card name from the background field, enclose its descriptor in parentheses:

card (field id 7)

Object Names
Names are optional for cards, backgrounds, buttons, and fields. You assign a name for any of these objects by typing into the Name box in the object’s Info dialog box, which appears when you choose the object’s Info command from the Objects menu.

Object names can include any characters, even spaces. When you use a name (background button "belly") in a statement, put quotation marks around the object name to ensure that HyperCard recognizes it as a literal and doesn’t look for a variable by that name. Names are not optional for stacks. You must provide a name for each new stack you create. A stack name must be a valid Macintosh filename.

Be careful with names
It’s difficult to manipulate a name that extends out of the naming window, although you can scroll it left and right (and up and down if it has more than one line) by dragging. It’s also difficult to refer by name to an object if you put a double quotation mark in its name. Also, if you use numbers for an object’s name, HyperCard could misinterpret the name: it takes card "1812" to mean a card whose number, rather than name, is 1812.
CHAPTER 5

Referring to Objects, Menus, and Windows

The name of an object is one of its properties. (See Chapter 12, “Properties,” for an explanation of properties and a description of the name property.)

The name property of an object has three forms—long, abbreviated, and short. The long name of an object includes the type of object, its name, its enclosing object (either a card or background), and the full pathname of its stack:

card button "Rolo" of card "Home" of stack "MyHardDisk:Home"

The abbreviated form includes the type of object and its name:

card button "Rolo"

The short form includes just the name:

"Rolo"

Object Numbers

Buttons, fields, cards, and backgrounds always have numbers by which you can refer to them. An object’s number represents its position within its enclosing object: buttons and fields are ordered within a card or background, as are card parts (remember that this term refers to both buttons and fields) and button families; cards and backgrounds are ordered within their stack.

For objects and parts with numbers one through ten, there are three ways to express an object’s number: use an integer following its generic name (card 2), use one of the numeric constants one through ten following its generic name (card two), or use one of the ordinal constants first through tenth preceding its generic name (second card). For objects with numbers higher than ten, you have to use the integer value.

Object numbers are contiguous from one through the number of currently existing objects within the enclosing object: card buttons and card fields within their card; background buttons and background fields within their background; cards within their stack (not their background); and backgrounds within their stack. If you delete an object, its number is reassigned to the object following it in order, and so on for the succeeding objects as well.
Part Numbers

A part is the generic name for either a button or field. HyperCard thinks of buttons and fields as parts of either their enclosing backgrounds or cards and, as such, numbers them as they are generated. If you don’t specify, HyperCard assumes you are referring to card parts.

The reference to generic parts rather than specific buttons or fields makes it much easier to iterate through all the buttons and fields in a stack. You can use the `number` function to count the number of parts of a card or background. (See `number` in Chapter 11, “Functions.”)

This example shows a handler that searches through all the scripts of a stack to find a word pattern. It is actually a shorter form of the `searchScript` handler, which you can find in your Home stack, rewritten to iterate through script parts rather than buttons and fields:

```on searchScript pattern
global ScriptFindString

push card -- remember where we are
set lockMessages to true
set lockRecent to true

if pattern is empty then
    ask "Search for what string?" with ScriptFindString
    if (it is empty) or (the result is "Cancel")
        then exit searchScript

    put it into pattern -- otherwise save it in pattern
end if

put pattern into ScriptFindString
set cursor to busy

-- search the stack script of the stack
if the script of this stack contains pattern
then edit script of this stack
```
CHAPTER 5

Referring to Objects, Menus, and Windows

-- search the background scripts
repeat with curBkgnd = 1 to the number of bkgnds
  set cursor to busy
  go to card 1 of bkgnd curBkgnd
  if the script of this bkgnd contains pattern
    then edit script of bkgnd curBkgnd
end repeat

-- search the scripts of background parts
-- (bg buttons and fields)
  repeat with curPart = 1 to the number of bg parts
    set cursor to busy
    if the script of part curPart contains pattern
      then edit script of part curPart
  end repeat

-- search the card and card part scripts of the stack
repeat with curCard = 1 to the number of cards
  set cursor to busy
  go card curCard
  if the script of this card contains pattern
    then edit script of this card

-- otherwise search through the card buttons and fields
  repeat with curCdBtnOrFld= 1 to the number of ¬
    card parts
    set cursor to busy
    if the script of card part curCdBtnOrFld ¬
      contains pattern
      then edit script of card part curCdBtnOrFld
  end repeat
end repeat
pop card -- return to where we were
set lockMessages to false
set lockRecent to false
CHAPTER 5

Referring to Objects, Menus, and Windows

```
restoreUserLevel -- set userLevel back to whatever it was
answer "Search script done!" -- Δ
end searchScript
```

Button Families

Button families are specified by number only; they do not have names or IDs. Only the numbers from 1 to 15, inclusive, are valid. An example of a HyperTalk reference to a button family is

```
cd family 1
```

Special Ordinals

In addition to the ordinal constants first through tenth, HyperTalk has three special ordinals: middle, last, and any. The values of the special ordinals are resolved according to the number of objects in the set. Middle resolves to half the number of objects (rounded down to the nearest integer) plus 1. The ordinal last resolves to the number of objects. Any resolves to a random number between 1 and the number of objects. (The special ordinals also work with chunk expressions, which are described in Chapter 7, “Expressions.”)

Object Numbers and Tab Order

The sequence of object numbers determines tab order for fields: you can move from field to field within a background and card using the Tab key—it moves from the lowest numbered field to the highest through the background fields first, then the card fields. The sequence also determines which button or field gets a message when several are layered on top of each other (the highest numbered one is closest and gets the message), and it determines which card or background is referred to by the special descriptor next or previous (see the section “Special Object Descriptors” later in this chapter).

Reassigning object numbers

You can reassign object numbers of buttons and fields with the Bring Closer and Send Farther menu commands. See the HyperCard Reference Guide for details. ◆
CHAPTER 5

Referring to Objects, Menus, and Windows

Object ID Numbers

HyperCard generates an object ID number for each object within a stack. This number is unique for that type of object within its enclosing object. For example, each button (the type of object) on a card (the enclosing object) has a different ID number. Object ID numbers never change, and if an object is deleted, its number is not reassigned to a newly created object (until the HyperCard object limit, listed in Appendix G, has been reached). An object’s ID number is its generic name, followed by the word ID (in uppercase or lowercase), followed by an integer (for example, card id 5734).

The ID number of a copied object is different

If you copy an object and paste it into a different enclosing object, the copy is then a different object from the original, and it has a different ID number. For example, if you copy a card and paste it into a different stack, the ID number of the pasted card is different from the ID number of the card you copied. Therefore, you can’t assume that you have “moved” the card when you copy it, paste it, and delete the original—a button that took you to the original will probably not take you to the copy.

Because ID numbers are unique and unchanging for all objects within a stack, HyperCard uses them internally to identify objects (for example, to identify the target of a go command generated with the LinkTo feature in the Button Info dialog box). HyperCard can generally find objects faster if they are identified by ID number. If you ask for the name of an object that has no name (put the name of last card), HyperCard returns its ID number.

The ID of an object is one of its properties. The ID property of an object has three forms that are similar to the three forms of the name and are differentiated by the same adjectives—long, abbreviated, and short. The long ID of an object includes the type of object, its ID number, its enclosing object if necessary, and the full pathname of its stack:

card id 2590 of stack "HyperDisk:HyperCard:Stacks:Home"

The abbreviated form includes the type of object and its ID number:

card id 2590
The short form includes just the ID number:

2590

All objects except stacks always have ID numbers; stacks never have ID numbers.

**Special Object Descriptors**

You can use the special descriptor `this` to refer to the current card, background, or stack—for example:

`put the id of this card into whereFound`

You can't use `this` with buttons or fields.

You can refer to the card or background preceding the current one, within the stack, as `previous`, which can be abbreviated `prev`. Similarly, you can refer to the card or background following the current one as `next`—for example:

`go to next background`

You can refer to the card that was current immediately prior to the current one as `recent`.

You use `me` within a script to specify the object containing the currently executing handler. For example, this statement in a field script would put the value of the `textHeight` property of the field into the variable `height`:

`put the textHeight of me into height`

**Identifying a Stack**

A stack is a HyperCard document. In some cases when you’re writing a script or using the Message box, you can refer to a stack by its name alone. To do that, the stack must be in the current folder, in the folder containing the HyperCard application, or in a folder listed in the global variable `stacks`. (The `stacks` variable gets its list of folders on startup from the Stacks Search Paths card of...
the Home stack. The Stacks Search Paths card is one of three Search Paths cards that contain lists of search paths, or pathways through the folders on your disk, that HyperCard should follow to retrieve a stack, application, or document.) When the stack is located anywhere else, you must let HyperCard know the full pathname by which it can find the stack.

A full pathname is a concatenation of the volume name, directory name or names, and filename, separated by colons. The volume name is the name of the disk or server containing the stack. The directory names are the names of all the folders, if any, that HyperCard has to open to get to the stack. (HyperCard sometimes might have to open several folders because folders may contain other folders to any depth.) The filename is the stack name. Figure 5-2 shows the structure of a pathname.

Figure 5-2 A pathname

The only unambiguous way to refer to a stack in a script or in the Message box is the word stack followed by its name in quotation marks. When you refer to a stack, you can use the full pathname to specify the stack's exact location:

```
go to stack "myDisk:myFolder:mystack"
```

You can also type the folder name in the Stack Search Paths card of the Home stack. If HyperCard can't find a stack you request, it displays a dialog box that allows you to click your way through the directories until you reach the stack. HyperCard notes your path and, once you've found the stack, automatically records its folder on the Search Path card in the Home stack.
Ambiguous stack descriptors
HyperCard tries to derive a proper stack name from an ambiguous expression in a place where it expects a stack descriptor, but it cannot always succeed. In that case, HyperCard displays the directory dialog box to allow the user to find the stack file.

Naming a Stack

You must name a stack when you create it. (For all other objects, names are optional.) You create a stack with the New Stack command in the File menu or with the create stack command. (See Chapter 10, “Commands,” for more information about the create stack command.) When you use the New Stack command, a dialog box appears in which you type the name for the new stack. (See Figure 5-3.) You can also select the card size for your stack either by dragging the rectangle on the right side of the dialog box or by selecting one of the preset sizes available from the pop-up menu in the upper-right corner, as shown in Figure 5-4.

Figure 5-3  New Stack dialog box

Identifying a Stack
Combining Object Descriptors

To refer to objects within a stack, you combine object descriptors using either of the prepositions of and in between an object descriptor and that of its enclosing object. Combined object descriptors proceed left to right from the smaller to the larger:

first field of last card of this background

This syntax lets you refer directly to any object within the current stack—you don’t have to go to the card containing a particular field to get the field’s contents or put something into it. For example, if the current card were the first in the stack, you could still execute the following command:

put the selection into field "undoHolder" of last card

You cannot refer to an object within another stack. You have to go to the stack before you can address its objects directly.

You can further combine field descriptors with chunk expressions, which are described in Chapter 7, “Expressions.”
Referring to Menus and Menu Items

HyperCard menus and menu items have names and numbers, as well as ID numbers.

Menu and Menu Item Names

Menus and menu items all have names. The name of a menu or menu item is one of its properties. You assign a name for a menu when the menu is created with the \texttt{create} command. You assign a name to a menu item when you put the menu item into a menu with the \texttt{put} command.

Menu and menu item names can include any characters, including spaces. When you use a menu or menu item name in a statement, put quotation marks around the name to ensure that HyperCard recognizes it as a literal and doesn’t look for a variable by that name, for example, \texttt{menuItem "User Preferences"}. When you refer to menus by name, you must precede the menu name with the word \texttt{menu}, for example, \texttt{menu "Notes"}. When you refer to existing menu items, you precede the name of the menu item with the word \texttt{menuItem}, for example, \texttt{menuItem "Power Tools"}. You must also specify which menu the menu item is in, for example, \texttt{menuItem "Power Tools" of menu "Utilities"}. You can use either \texttt{of} or \texttt{in}. There is an exception to these rules: you do not have to precede the name of a menu item with \texttt{menuItem} or specify the menu that a menu item is in when referring to a menu item in a \texttt{doMenu} statement:

\begin{verbatim}
doMenu "Power Tools"
\end{verbatim}

Menu and Menu Item Numbers

Menus and menu items always have numbers by which you can refer to them. The menu number refers to the menu’s position within the menu bar. Menus are ordered from left to right in the HyperCard menu bar. The menu item number refers to the menu item’s position within its enclosing menu. Menu items are ordered from top to bottom.
Referring to Menus and Menu Items

You can get the name of a menu item by referring to it with its number. For example, typing the following statement into the Message box

```
menuItem 3 of menu "Utilities"
```

puts the name of the third menu item in the Utilities menu into the Message box.

You can also use ordinals when referring to menus and menu items. For example, typing the following statement into the Message box

```
the third menuItem in the fifth menu
```

puts the name of the third menu item in the fifth menu into the Message box.

You cannot get the number of a menu or menu item by referring to it by name. For example, the statement

```
the number of menuItem "Card Info..." of menu "Objects"
```

would result in an error dialog box.

The dashed line used in menus to visually separate menu items that have different functions is also counted as a menu item in the number of menu items. For example, if you have a menu with a dashed line separating the second and third menu choices, the dashed line is referred to as `menuItem 3`.

You always refer to the dashed line by a menu item number because it has no name property. To put a dashed line in an existing menu, use the `put` command syntax as follows:

```
put "-" after menuItem itemName of menu menuName
```

To get a return-delimited list of all of the menus in the current HyperCard menu bar, use the function `the menus`. To get a return-delimited list of all the menu items in a menu, use the menu name, number, or ordinal: `menu "Power Tools", menu 6, or the sixth menu`.

To determine how many menus are in the current HyperCard menu bar, use the function `number of menus`. To determine how many menu items are in a specified menu, use the function `number of menuItems in menu menuName`. To determine the name of a menu or menu item, use the `name` property.
Referring to Menus and Menu Items

You can use menu commands, functions, and properties with the HyperCard built-in menus with the exceptions noted in the descriptions of the menu commands, functions, and properties.

The following statements use some of the menu commands and properties to create the simple example of a custom (user-defined) menu shown in Figure 5-5.

```
create menu "MyMenu"
put "Item 1" into menu "MyMenu"
put "-" after menuItem "Item 1" of menu "MyMenu"
put "Item 3" after the second menuItem of menu "MyMenu"
put "Item 4" after menuItem "Item 3" of menu "MyMenu"
put "-" after menuItem "Item 4" of menu "MyMenu"
put "Item 6" after the fifth menuItem of menu "MyMenu"
set the enabled of menuItem "Item 4" of menu "MyMenu" to false
set the cmdChar of menuItem "Item 6" of menu "MyMenu" to "]"
set the checkMark of menuItem "Item 3" of menu "MyMenu" to true
```

![Figure 5-5](image)
Referring to Windows

In addition to referring to a window by name, you can refer to a window by ID number or by a number representing its place in the window layer.

Card windows are referred to as card window (for example, set the location of card window to 45,65). Card window always refers to the card window of the current stack. You can’t use the card window syntax to refer to the card window of another stack unless you go to that stack first. You can, however, refer to another visible or hidden stack’s window with the syntax window "stackName" (for example, set the location of window "Dizzie" to "45,65").

You refer to the Message Watcher window as Message Watcher (for example, show Message Watcher) or window "Message Watcher" (for example, set the loc of window "Message Watcher" to "65,80").

You refer to the Variable Watcher window as Variable Watcher (for example, set the loc of Variable Watcher to "64,124") or window "Variable Watcher" (for example, set the hBarLoc of window "Variable Watcher" to "65,80").

The Tools palette and Patterns palette are referred to in HyperTalk as tool window and pattern window, or window "tools" and window "patterns", respectively.

The Message box is referred to as message window or message box. Message can be abbreviated msg in any of these forms.

The Scroll window is referred to as scroll window or window "Scroll".

In addition to HyperCard’s built-in windows, there are also external windows you create with the picture command. They are referred to by the name you give to the window at the time of its creation—for example,

set the rect of window "MyBestPicture" to "60,90,300,300"
You can also retrieve the ID of any window, including pictures and palettes. The unique IDs of picture windows could be particularly useful for referring to different windows that have two pictures with the same name.

In addition to its name and ID number, you can refer to a window by its number—that is, its position in the front-to-back order of windows. (Keep in mind that a window’s number is a read-only property, and that this number changes as other windows are selected, opened, or closed.)

The `windows` function evaluates to a list of all the windows (listed by name in front-to-back order) that are currently available to HyperCard. The list could include the Message box, Scroll window, Message Watcher window, Variable Watcher window, FatBits window, Patterns palette, Tools palette, the windows of the Home stack and any currently open stacks, and any user-defined external windows.

Chapter Summary

Here is a summary of the material covered in this chapter:

- You refer to a HyperCard object using an object descriptor—its generic name and its specific designation.
- Cards, backgrounds, buttons, fields, and windows always have unique ID numbers that never change, always have object numbers that may change, and may optionally be given names.
- You can use special ordinals—`middle`, `last`, and `any`—to refer to objects by their position within their enclosing object.
- You can refer to the current card, background, or stack with `this`. You can refer to the card or background preceding the current one with `previous`, and to the one following the current one with `next`. You can refer to the card that was current prior to the current one with `recent`.
- The term `me`, in a script, refers to the object containing the script.
- The only unambiguous object descriptor for a stack is the word `stack` followed by the stack’s filename within quotation marks.
Chapter Summary

- You can combine object descriptors to refer directly to any object in the current stack.
- Menus and menu items can be referred to by their name or number.
- Built-in HyperCard windows are referred to by names that are defined in the HyperTalk vocabulary. An external window you create with the `picture` command is referred to by the name you give it when you create it. Windows can also be referred to by the number of their position in the window layer or their unique ID numbers.
Values

This chapter describes the elements of HyperTalk that contain values. **Values** are the information on which HyperTalk operates. A HyperTalk **expression** is a description of how to get a value.

The **sources of values** in HyperTalk are

- constants
- literals
- functions
- properties
- numbers
- containers

These sources of values are the most basic expressions.

HyperCard does not have data types: values are stored simply as strings of characters. (For mathematical operations, numbers are represented internally in a more efficient format; see the Standard Apple Numerics Environment description in this chapter.)

## Constants

A **constant** is a named value that never changes. It’s different from a variable in that you can’t change it, and it’s different from a literal in that its value is not always the string of characters making up the name. For example, the constant **empty** is the same as the null string (the literal ""), and the constant **space** is the same as the literal " ". All HyperTalk constants are described in Appendix B, “Constants.”
CHAPTER 6

Values

Literals

A literal is a text string whose value is the string, exactly as it appears. Literals are denoted by double quotation marks at both ends of the string. (You must use the straight double quotation mark, not the printer’s double quotation marks typed with the Option–left bracket and Option–Shift–left bracket keys.) Any character except double quotation mark, return, or “soft” return (generated by pressing Option-Return) can be part of a literal string. A literal can be of any length. For example, "This is a literal string" is a literal.

Unquoted literals are not supported

Do not use unquoted literals in HyperTalk. The value of an unquoted literal is the literal of itself—as though you had entered put "fred" into fred. Always put double quotation marks around a word you want HyperCard to take as a literal.

Functions

A function is a named value that HyperCard calculates when the statement in which the function is used executes. The value of a function varies according to conditions of the system or according to the value of parameters you pass to the function when you use it.

For example, the built-in function named the time returns the current time in place of itself in a HyperTalk statement:

put the time into the message box

If the current time were 5:12 PM, the above example would put 5:12 PM into the Message box.

You can also define your own functions in scripts using the function handler structure described in Chapter 9, “Control Structures and Keywords.”

All built-in HyperTalk functions are described in Chapter 11, “Functions.”
Properties

A property is a named value representing one of the defining characteristics of an element of the HyperCard environment. Different types of objects and other elements have different properties, according to their purpose. For example, fields share a set of properties, many of which are different from the set shared by buttons.

You get the value of most properties by using the property name as a function in a script or in the Message box. For example, the following statement retrieves the location property (two integers separated by a comma) of button 1, and it puts the value into the Message box:

```
put the location of button 1 into msg
```

This next example returns a value of either true or false for the enabled property of the menu Clues.

```
put the enabled of menu "Clues"
```

You can also change most properties with the set command. All HyperCard properties are described in Chapter 12, “Properties.”

Numbers

A number in HyperCard is a character string consisting of any combination of the numerals 0 through 9, representing a decimal value. A number can include one period (.) representing the decimal point, but it can have no other punctuation nor a space character. A number can be preceded by a hyphen or minus sign to represent a negative value (HyperCard doesn’t recognize a plus sign as part of a number). Numbers that consist only of numerals are integers. Numbers that include a period are real and, when used with mathematical operators, are manipulated with floating-point operations.
CHAPTER 6

Values

Standard Apple Numerics Environment

HyperCard performs mathematical operations with Standard Apple Numerics Environment (SANE) routines, but you don’t have to worry about how to represent the values. You always enter numbers into HyperCard containers as numeric strings.

When performing a mathematical operation, HyperCard automatically converts the strings representing the numbers to SANE numeric values. If you put the result of the operation into a variable, it’s stored as a SANE numeric value; if you put it into a field or the Message box, HyperCard automatically converts it back to a string with a precision of up to 19 decimal places. The same conversion takes place if you put the variable into a field or the Message box at a later time, or if you use it in a way that implies a string (character 2 of varName). So although SANE values are used internally for handling numbers with speed and precision, you can always think of HyperTalk numbers as strings.

Precision

The precision of the decimal string, resulting from putting a SANE numeric value into a field or the Message box, is controlled by the numberFormat global property (see Chapter 12, “Properties,” for a detailed description). For example, the command

\[
\text{set numberFormat to 0.00}
\]

would result in a string with at least one digit to the left of the decimal point and exactly two digits to the right of the decimal point.

The numberFormat property does not affect the precision with which mathematical operations are executed, only the precision with which the results are displayed. When you put a number into a field or the Message box to display it, however, HyperCard converts it to a decimal string. So any extra precision it may have had (beyond the numberFormat specification in effect at the time) is lost.
CHAPTER 6

Values

Number Handling

The following example shows how number handling works. These three HyperTalk statements put the constant \( \pi \) into a variable, set the `numberFormat` property, and put the value of the variable into the Message box, respectively:

\[
\begin{align*}
\text{put } \pi \text{ into joe} \\
\text{set } \text{numberFormat} \text{ to 0.00} \\
\text{put } \text{joe} \text{ into msg}
\end{align*}
\]

The result shown in the Message box is 3.14159265358979323846. In this case, \( \pi \) is entered into the variable `joe` as a string, and it remains a string, so `numberFormat` has no effect. If, however, you perform a mathematical operation on the variable, HyperCard converts it to a SANE numeric value:

\[
\begin{align*}
\text{put } \pi \text{ into joe} \quad \text{-- joe contains a string} \\
\text{add } 0 \text{ to joe} \quad \text{-- mathematical operation makes it a number} \\
\text{set } \text{numberFormat} \text{ to 0.00} \quad \text{-- affects the format of joe} \\
\text{put } \text{joe} \text{ into msg}
\end{align*}
\]

The result shown in the Message box is 3.14. In this case, `numberFormat` takes effect when `joe` is converted from a SANE numeric value to a string as it’s put into the Message box. The example statements for number handling work only when placed inside a handler. If entered one at a time in the Message box, the result is in the default format, because HyperCard resets `numberFormat` to its default value during idle time.

Containers

A container in HyperCard is a place where you can store a value. Containers include fields, buttons, variables, menus, the current selection, and the Message box. Containers other than fields and buttons can store values of any length, including zero length. Containers other than the Message box can have more than one line in them; each line ends with a return character (which can be the only character in the line).
CHAPTER 6

Values

Fields

A field is a HyperCard object for holding and displaying editable text. Fields are interesting objects because they are also containers—a field’s value is the text string it contains. Fields can also act as expressions; for example, put field 1 into it puts the value of the expression field 1 into the variable it. Variables are described in the section “Variables” later in this chapter, and expressions are described in Chapter 7, “Expressions.”

You can refer to fields directly by name, number, or ID number. (See Chapter 5, “Referring to Objects, Menus, and Windows,” for more about how to refer to fields.)

Fields belong to cards or to backgrounds; the text held by a field, however, usually remains with the card (unless the sharedText property is true), even if the field belongs to the background. A field can contain up to 30,000 characters, including spaces, return characters, and other invisible characters. If you put more than that many characters into a field, the extras are ignored.

You can search through text with the find command unless the dontSearch property of the field is true. You can edit it using the I-beam pointer of the Browse tool when the field isn’t locked.

About Paint text

You can also put text onto cards and backgrounds as Paint text—pictures that look like characters. Paint text can’t be edited once it has been fixed onto the card or background (although you can paint over it or erase it as you can any part of a picture). See the HyperCard Reference Guide for more information on Paint text.

Buttons

As mentioned in Chapter 2, “HyperTalk Basics,” buttons are action objects or “hot spots” on the screen that can also contain text. Buttons, like fields, are objects that are also containers—a button’s value is the text string it contains. Buttons can also act as expressions; for example,

put btn 6 into cd fld 1

puts the value of the expression btn 6 into the card field with the number 1.
Pop-up buttons contain text that they display as menu items. The following code fragment creates a new pop-up button whose menu items are the currently running programs:

```hypercard
doMenu "New Button"
set style of last button to popup
put the programs into last button
```

See the `programs` function in Chapter 11, “Functions,” and the `style` property in Chapter 12, “Properties.” Variables are described in the next section, and expressions are described in Chapter 7, “Expressions.”

### Variables

A **variable** is a named container that has no visible representation other than its name. Its value is a character string of any length. The variable name is a HyperTalk **identifier**. An identifier can be of any length, always begins with an alphabetic character, and can contain any alphanumeric character plus the underscore character ( _ ).

You assign a value to a variable with the `put` command. You cannot read from a nonexistent variable—you must create it by putting something into it before you use it. The constant `empty`, the null string, counts as something you can put into a variable. This example puts a numeric value 12 into the variable `fudge`, adds 5 to that variable, and then puts the result, 17, into the Message box. Enter each line separately in the Message box.

```
put 12 into fudge
add 5 to fudge
put fudge into msg
```

HyperCard assumes that an unquoted word used in an expression is a variable when it can’t interpret the word as some other source of value (the string is not a function, constant, property, or other container name). If you haven’t put a value into a variable by that name, HyperCard treats it as an unquoted literal.
Scope of Variables

HyperCard has both local and global variables. A **local variable** is valid only during the current invocation of the currently executing handler. You don’t need to declare a local variable before you use it—just put something into it. A **global variable** is valid for all handlers. You must declare a variable as global by using the `global` keyword in each handler before you use the variable:

```
global useMeEverywhere, useMeToo
```

HyperTalk assumes a variable to be local unless you specifically use the `global` keyword.

For more details on the `global` keyword, see Chapter 9, “Control Structures and Keywords.”

Parameter Variables

You create parameter variables when you put their names after the message name in a handler:

```
on messageName firstParam, secondParam
```

When the handler is called, these variables are assigned the values, if any, of the items in a comma-separated list of expressions following the message name in the calling statement. Parameter variables are local to their handler. See Chapter 4, “Handling Messages,” for an explanation of parameter passing.

The Variable `It`

The local variable named `It` is the destination of the commands `get`, `ask`, `answer`, `read`, and `request`. For example, `get the name of field 1` puts the value of that background field’s name into `It`. `Convert` puts its results into `It` if another destination isn’t specified.

For information on these commands, see Chapter 10, “Commands.”
CHAPTER 6
Values

Menus

When a menu reference does not refer to a menu as a HyperCard element (for example, get enabled of menu "Home"), then it behaves as a container. Like variables, you assign a value (made up of text) to a menu with the put command. The text becomes the menu’s menu items. In this manner, a menu evaluates to a list of its menu items; the statement

```plaintext
put menu "Edit" into editMenuItems
```

stores a list of the menu items in the Edit menu in the variable `editMenuItems`.

The Selection

The selection is a container that holds the currently selected area of text. You can put values into, before, or after the selection or put the selection (or any chunk of the selection) into another container. Figure 6-1 shows an example that puts a string into the selection to replace the highlighted text.

---

**Figure 6-1** Manipulating the selection

Starting with this selection

```
The selection is always highlighted.
In inverse video on the Macintosh.
```

this HyperCard command

```plaintext
put "easy to change using a" into the selection
```

produces this result

```
The selection is always easy to change using a Macintosh.
```
If the phrase I'm the selected text is selected, and your handler issues the statement

put the selection into the Message box

then I'm the selected text appears in the Message box. (Both instances of the word the in the example are optional.)

Found text isn't selected
Text found by the find command is indicated by a box around it—it is not placed into the selection. To get information about text found with the find command, use the functions foundText, foundChunk, foundLine, and foundField, which are described in Chapter 11, “Functions.”

You must select some text with the mouse or the click, drag, or select command before you can manipulate the selection container.

You can also get information about a chunk of text or a line in a field that has been clicked with the clickChunk or clickLine function, described in Chapter 11, “Functions.”

The Message Box

The Message box is a special container. Any HyperTalk expression can be put into the Message box. Typically, you use the Message box to send a HyperTalk message directly to an object or to HyperCard. The Message box is a single-line container, as shown in Figure 6-2. If you put more than one line from a multiple-line container into the Message box (put card field 2 into msg), only the first line is copied into the Message box.

Referring to the Message box
There are several forms you can use when referring to the Message box. The forms are message box, message, and msg.
CHAPTER 6

Values

Figure 6-2  The Message box

The Message box is the default destination for the put command. You can put a value directly into the Message box without specifying the Message box by using one of these forms of the put command:

put property [of element]
put container
put function

Property is an expression that yields any HyperCard property, element yields the descriptor of a HyperCard element (an object, menu, menu item, or window), container yields a container, and function yields any HyperCard function.

If you put something into the Message box when it’s hidden, HyperCard shows it automatically. You can toggle the Message box between being hidden or shown by pressing Command-M.

The Message box can be specified by just the word message or its abbreviation msg. Optionally, you can follow either of those with either box or window, and you can precede either with the word the.

See the description of the put command in Chapter 10, “Commands,” for more information about the values you can put into the Message box.

Chapter Summary

Here is a summary of the material covered in this chapter:

- The most basic expressions in HyperTalk are constants, literals, functions, properties, numbers, and containers.

- HyperTalk’s values can always be treated as strings of characters.

- Containers—fields, variables, the selection, and the Message box—are places to store values.
This chapter describes the expressions you use to refer to values. An expression is a description of how to get a value. It may be as simple as a single source of a value, or it can be a complex expression built with operators.

This chapter also describes HyperTalk’s operators, the elements of the language that you use in expressions to manipulate and calculate values, described in Chapter 6, “Values.”

Complex Expressions

You can build complex expressions using values and operators. As a complex expression is evaluated, the values of its basic components are manipulated to derive a final value in place of the entire expression. (The original values are not changed in the process.) Complex expressions are evaluated according to rules of precedence, and restrictions apply to the values that can be used, depending on their operators.

**Chunk expressions are different**

Chunk expressions are a different type of expression: they designate pieces of the strings representing values. Chunk expressions are described later in this chapter. ♦

Factors

A factor is a single element of value in an expression. The following constructs are factors:

- a simple source of value
- an expression enclosed in parentheses
Expressions

- an element with a minus sign in front of it that evaluates to a number
- an expression with the word `not` in front of it that evaluates to `true` or `false`

An expression can be a single source of value, or it can be any two expressions with an operator between them.

The difference between a factor and an expression is important to the syntax of HyperTalk commands and functions. Where a built-in HyperTalk command parameter permits an expression, you can specify as complex an expression as you wish. HyperCard derives the final value before passing the parameter to the command. For example, the `add` command accepts a complex expression as its first parameter:

```
add 46+12*monthlyRate to total
```

In contrast, when a built-in HyperTalk function requires a factor, HyperCard takes the value of the first factor as the parameter to pass to the function. For example, the `sqrt` function takes the first factor following its name as its parameter. This is illustrated by the following expression, which you can type into the Message box or use in a statement:

```
the sqrt of 4+12
```

In the example, the `sqrt` function takes the factor `4` as its parameter, rather than the value of the expression `4+12`. So the entire expression evaluates to `14`, rather than `4`, which would be the value if `sqrt` accepted an entire expression. (To specify the entire expression `4+12` as the parameter, you can enclose it in parentheses, which turns it into a single factor.)

**Two hyphens always indicate a comment**

You can put a hyphen in front of a factor to create another factor, and you can put another hyphen in front of that and still have a factor. However, two hyphens in sequence indicate a comment, so you must separate the hyphens with a space or enclose the inner factor in parentheses for HyperCard to recognize the construct as a factor.

HyperTalk’s built-in commands and functions are described in Chapters 10 and 11, respectively.
CHAPTER 7

Expressions

HyperTalk Operators

Operators are used in complex expressions to derive values from other values. Operators fall into several categories:

- Arithmetic operators work on numbers and result in numbers.
- Comparison operators work on numbers, text, and Boolean values (true or false) and result in Boolean values.
- Logical operators work on Boolean values and result in Boolean values.
- Text operators manipulate text strings and result in text strings.

Table 7-1 is a list of all the operators in HyperTalk.

Table 7-1 HyperTalk operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>Concatenate: Text string operator that joins the text string yielded by the expression on its left to the text string yielded by the expression on its right.</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Concatenate with space: Text string operator that joins the text string yielded by the expression on its left to the text string yielded by the expression on its right, with a space between them.</td>
</tr>
<tr>
<td>/</td>
<td>Divide: Arithmetic operator that divides the number to its left by the number to its right.</td>
</tr>
<tr>
<td>=</td>
<td>Equal: Comparison operator that results in true if the expression to its left and the expression to its right have the same value. The expressions can be arithmetic, text string, or logical.</td>
</tr>
<tr>
<td>^</td>
<td>Exponent: Arithmetic operator that raises the number to its left to the power of the number to its right.</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than: Comparison operator that results in true if the expression to its left has greater value than the one to its right. The expressions can be both arithmetic or both text.</td>
</tr>
</tbody>
</table>

continued
### Table 7-1 HyperTalk operators (continued)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>≥</code></td>
<td>Greater than or equal to: Same as <code>&gt;=</code>. The <code>≥</code> character is obtained on the Macintosh keyboard by pressing Option-period (.)</td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>Greater than or equal to: Same as <code>≥</code>. Comparison operator that results in <code>true</code> if the expression to its left has greater value than the one to its right or the same value. The expressions can be both arithmetic or both text.</td>
</tr>
<tr>
<td><code>(</code></td>
<td>Grouping: Expressions within the innermost pair of parentheses are evaluated first. Parentheses don’t force a new level of evaluation; they change the sequence in which the current level of evaluation proceeds.</td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>Less than: Comparison operator that results in <code>true</code> if the expression to its left has less value than the one to its right. The expressions can be both arithmetic or both text.</td>
</tr>
<tr>
<td><code>≤</code></td>
<td>Less than or equal to: Same as <code>&lt;=</code>. The <code>≤</code> character is obtained on the Macintosh keyboard by pressing Option-comma (,).</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>Less than or equal to: Same as <code>≤</code>. Comparison operator that results in <code>true</code> if the expression to its left has less value than the one to its right or the same value. The expressions can be both arithmetic or both text.</td>
</tr>
<tr>
<td><code>-</code></td>
<td>Minus: Arithmetic operator that makes negative the number to its right or, if it is between two numbers, subtracts the one on the right from the one on the left.</td>
</tr>
<tr>
<td><code>*</code></td>
<td>Multiply: Arithmetic operator that multiplies two numbers it appears between.</td>
</tr>
<tr>
<td><code>≠</code></td>
<td>Not equal: Same as <code>&lt;&gt;</code>. The <code>≠</code> character is obtained on the Macintosh keyboard by pressing Option–equal sign (=).</td>
</tr>
</tbody>
</table>
CHAPTER 7
Expressions

Table 7-1  HyperTalk operators (continued)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&gt;</td>
<td>Not equal: Comparison operator that results in true if the expression to its left and the expression to its right have different values. The expressions can be arithmetic, text, or logical.</td>
</tr>
<tr>
<td>+</td>
<td>Plus: Arithmetic operator that adds two numbers it appears between.</td>
</tr>
<tr>
<td>and</td>
<td>AND: Logical operator that results in true if both the expression to its left and the expression to its right are true.</td>
</tr>
<tr>
<td>contains</td>
<td>Contains: Comparison operator that results in true if the text string yielded by the expression on its right is found in the text string yielded by the expression on its left.</td>
</tr>
<tr>
<td>div</td>
<td>Divide and truncate: Arithmetic operator that divides a number to its left by a number to its right, ignoring any remainder, resulting in just the whole part.</td>
</tr>
<tr>
<td>is</td>
<td>Is: Same as =.</td>
</tr>
<tr>
<td>is a</td>
<td>Is a, is an: Comparison operator that tests for types. Types include number, integer, point, rect, date, empty, and logical.</td>
</tr>
<tr>
<td>or</td>
<td>Is or, is an: Comparison operator that tests for types. Types include number, integer, point, rect, date, and logical.</td>
</tr>
<tr>
<td>is in</td>
<td>Is in: Converse of contains; comparison operator that results in true if the text string yielded by the expression on its left is found in the text string yielded by the expression on its right.</td>
</tr>
</tbody>
</table>
| is not   | Is not: Same as <>.
| is not a | Is not a, is not an: Comparison operator that tests for types. Types include number, integer, point, rect, date, and logical. |
| or       | Is or, is not an: Comparison operator that tests for types. Types include number, integer, point, rect, date, and logical. |
| is not in| Is not in: Opposite of is in; comparison operator that results in true if the text string yielded by the expression on its left is not found in the text string yielded by the expression on its right. |

continued
### Table 7-1  HyperTalk operators (continued)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>is within</td>
<td>Is within: The <em>is within</em> operator tests whether or not a point lies inside a rectangle; it results in a Boolean value: <em>true</em> or <em>false</em>.</td>
</tr>
<tr>
<td>mod</td>
<td>Modulo: Arithmetic operator that divides the number to its left by the number to its right, ignoring the whole part, resulting in just the remainder.</td>
</tr>
<tr>
<td>not</td>
<td>NOT: Logical operator that results in <em>true</em> if the expression on its right is <em>false</em>, and <em>false</em> if the expression on its right is <em>true</em>.</td>
</tr>
<tr>
<td>or</td>
<td>OR: Logical operator that results in <em>true</em> if either the expression to its left or the expression to its right is <em>true</em>.</td>
</tr>
</tbody>
</table>
| there is a or     | There is a, there is an: Unary operator that results in *true* if the item exists. Items include the descriptor of a window, menu, menu item, file, button, field, card, card picture, background, background picture, part, stack, folder, document, file, or program. You can use this operator to check for any currently executing System 7–friendly program. The expression  
| there is an       | *there is a program* *programName*  
|                   | returns *true* if the program is both System 7–friendly and currently executing. When searching for an application or document, this operator uses the search paths stored in the Home stack. When searching for a file, it does not use the search paths. |
| there is not a or | There is not a, there is not an: Opposite of *there is a*; a unary operator that results in *true* if the specified item does not exist. Items include the descriptor of a window, menu, menu item, file, button, field, card, card picture, background, background picture, part, stack, folder, document, file, scripting language, or program. When searching for an application or document, this operator uses the search paths stored in the Home stack. When searching for a file, it does not use the search paths. |
CHAPTER 7

Expressions

Operator Precedence

Parentheses alter the order of expression evaluation. Different operators have different orders of precedence that determine how things get evaluated. The order in which HyperCard performs operations is shown in Table 7-2.

<table>
<thead>
<tr>
<th>Order</th>
<th>Operators</th>
<th>Type of operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>()</td>
<td>Grouping</td>
</tr>
<tr>
<td>2</td>
<td>–</td>
<td>Minus sign for numbers</td>
</tr>
<tr>
<td></td>
<td>not</td>
<td>Logical negation for Boolean values</td>
</tr>
<tr>
<td></td>
<td>there is a</td>
<td>Boolean test for HyperCard items</td>
</tr>
<tr>
<td></td>
<td>there is an</td>
<td>Boolean test for HyperCard items</td>
</tr>
<tr>
<td></td>
<td>there is not a</td>
<td>Boolean test for HyperCard items</td>
</tr>
<tr>
<td></td>
<td>there is not an</td>
<td>Boolean test for HyperCard items</td>
</tr>
<tr>
<td></td>
<td>within</td>
<td>Boolean test for point within rectangle</td>
</tr>
<tr>
<td>3</td>
<td>^</td>
<td>Exponentiation for numbers</td>
</tr>
<tr>
<td>4</td>
<td>*/ div mod</td>
<td>Multiplication and division for numbers</td>
</tr>
<tr>
<td>5</td>
<td>+ –</td>
<td>Addition and subtraction for numbers</td>
</tr>
<tr>
<td>6</td>
<td>&amp; &amp;&amp;</td>
<td>Concatenation of text</td>
</tr>
<tr>
<td>7</td>
<td>&gt; &lt; &lt;= &gt;= £ ‡</td>
<td>Comparison for numbers or text</td>
</tr>
<tr>
<td></td>
<td>is in contains</td>
<td>Comparison for text</td>
</tr>
<tr>
<td></td>
<td>is not in</td>
<td>Comparison for text</td>
</tr>
<tr>
<td></td>
<td>is a</td>
<td>Comparison for types</td>
</tr>
<tr>
<td></td>
<td>is an</td>
<td>Comparison for types</td>
</tr>
<tr>
<td></td>
<td>is not a</td>
<td>Comparison for types</td>
</tr>
<tr>
<td></td>
<td>is not an</td>
<td>Comparison for types</td>
</tr>
<tr>
<td>8</td>
<td>= is is not &lt;&gt; ≠</td>
<td>Comparison for numbers or text</td>
</tr>
<tr>
<td>9</td>
<td>and</td>
<td>Logical for Boolean values</td>
</tr>
<tr>
<td>10</td>
<td>or</td>
<td>Logical for Boolean values</td>
</tr>
</tbody>
</table>

Complex Expressions 117
Operators of equal precedence are evaluated left to right, except for exponentiation, which goes right to left. For example, $2^3^4$ means “3 raised to the fourth power, then 2 raised to that power,” whereas $1-2-3$ means “2 subtracted from 1, then 3 subtracted from that.” If you use parentheses, HyperCard evaluates the parenthetical expression first.

**Operators and Expression Type**

The operator you use must match the values you’re using it with: "tom" + "cat" would cause an error, because numeric values are required for addition. However, tom + cat would be acceptable if tom and cat were names of containers with numbers in them, and "tom" & "cat" would be acceptable because the & operator works on text strings (the result of this operation would be the text string tomcat). Text operators work on any value, because any value in HyperTalk can be treated as a text string; they always yield text strings.

Because numeric values are automatically converted to strings when necessary (see “Numbers” in Chapter 6, “Values”), they can be manipulated by both text operators and arithmetic operators. For example, 5 & 78 yields 578, and 5 + 78 yields 83.

Comparison operators try to treat both of their operands as numbers; if they can’t both be regarded as numbers, HyperCard treats them as text and does a lexical comparison. A lexical comparison uses the order of the ASCII table (see Appendix D, “Extended ASCII Table”). For letters, it’s the same as alphabetical order; for numerals, it’s 0–9, but it’s different from a numerical comparison because a lexical comparison looks at just one character at a time, rather than the number as a whole. For example, 9 < 10 results in true, because 9 is less than 10 arithmetically. But "9x" < "10x" results in false, because the operands are evaluated lexically and 9 is greater than 1.

**Chunk Expressions**

You use a chunk expression to specify a particular piece—a chunk—of the value of any source of value: constant, literal, function, property, number, or container. Chunk expressions can specify any character, word, item, or line in the source.
CHAPTER 7

Expressions

Syntax of Chunk Expressions

The form of a chunk expression designates the smallest part of the chunk first, then specifies each larger, enclosing part. You separate each part of the expression with the preposition of or its synonym in. For example, the expression

first character of second word of third line of field 1

specifies a single character in the field.

You modify the specification of the kind of chunk—character, word, item, or line—with the number of the particular one you want. The number can be an ordinal constant preceding the kind (tenth word) or an integer following the kind (line 2). You can also use a numeric constant in place of the integer (line two), or any numeric expression that resolves to an integer.

You can use the special ordinals middle, last, and any to specify a chunk within its enclosing part. HyperCard resolves a special ordinal to a number using the total number of chunks of the specified type within its enclosing part: middle resolves to one more than half the total (rounded down to the nearest integer), last resolves to the total, and any resolves to a random number between 1 and the total. For example,

put "Joe" into any word of line 2 of field 1

replaces a random word in the line with Joe.

It isn't necessary to specify the enclosing parts of the source in strict hierarchical order. You can designate any smaller part within any larger part:

character 35 of field 1

And, although you must go left-to-right from smaller to larger, you don't have to specify any smaller part than you want:

third item of It
CHAPTER 7

Expressions

Characters

Characters are designated by the chunk name character (or char). Spaces count as characters in any part of a source except words. (Words are delimited by spaces.) Commas count as characters except in items. (Items are delimited by commas.) Return characters count as characters in whole sources and items. (A return character delimits the last word on the line as well as the line itself.)

For example, if field 6 contains the phrase

It was the turtle, not I, who spilled the beans.

the chunk expression

character 25 of field 6

yields a comma (the one after not I).

Words

Words are composed of any characters, including punctuation, delimited by spaces and return characters, and are designated by the chunk name word:

word 2 of "Where's my cubicle?"

yields my.

Items

Items are composed of any characters, including punctuation, delimited by commas, and are designated by the chunk name item:

item three of "cat's, rat's, bat's, gnat's"

yields " bat's" (including the space character in front).
CHAPTER 7

Expressions

Lines

Lines are composed of any characters, including punctuation, delimited by return characters, and are designated by the chunk name line.

The chunk name line denotes text between the beginning of a container and the first return character, between two return characters, or between the last return character and the end of the container.

It doesn’t matter how many display lines it takes to display one container line. For example, a single line in a field might occupy several lines on the display if the text wraps around (which it does if the field isn’t wide enough to accommodate the whole line). Figure 7-1 shows two examples of lines in a field: one with text wrap, and one without text wrap.

Ranges

The preposition to in a chunk expression specifies a range of a chunk within the larger chunk:

word 1 to 5 of line 2 of field "fred"

The numbers given in a range are inclusive. For example:

char 2 to 5 of "Hedgehog"

yields edge.
CHAPTER 7

Expressions

You specify the range with integers (or with constants or numeric expressions that resolve to integers) following the chunk name, rather than with ordinal numbers preceding the chunk name. That is, you must say char 1 to 3 of "george"; you can’t say first to third char of "george".

When the first number in a character range is greater than the second, you get an empty string. For example, char 5 to 3 of "Motorcycle" yields "" or empty. For words, items, and lines, a "reversed" range evaluates to the first chunk of the range. For example, word 2 to 1 of "Motorcycle helmet" yields "helmet".

Figure 7-2 shows some chunk expressions, labeled in various valid forms of chunk expression syntax, in a hypothetical card field 1.

Figure 7-2 Chunk expressions

Chunks and Containers

Combining a chunk expression with the object descriptor of a field lets you refer directly to any piece of text down to a single character within the current stack:

put char 2 of line 2 of field 1 of last card

Figure 7-3 shows an example that refers to the single character “a”.

122 Chunk Expressions
CHAPTER 7
Expressions

**Figure 7-3** Combining chunks and objects

Third character of second word of third line of first field of fourth card.

<table>
<thead>
<tr>
<th>Character</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word</td>
<td>what</td>
</tr>
<tr>
<td>Line</td>
<td>That’s what I thought.</td>
</tr>
</tbody>
</table>

Field

```
This is line one.
This is line two.
That’s what I thought.
```

You can’t specify chunks in another stack
You can’t combine a stack name with a chunk expression; you must go to the stack first.

Chunks as Destinations as Well as Sources

Chunk expressions can be used to specify a part of the value in a container wherever a container name is used. So, the chunk can specify the destination of a value—where you’re putting it—as well as the source of a value—where you’re getting it. For example,

```
put "Mr Steve" into word 3 of field 1
```

replaces only the third word in the field with the value Mr Steve, leaving the rest of the field’s former contents intact.
Nonexistent Chunks

If you specify chunks that don’t exist as sources of values, you get nothing. For example,

```
put char 5 of "hey" into msg
```

puts empty (or nothing) into the Message box, because the word hey contains only three characters.

If you specify a nonexistent chunk as the destination of a put command, the outcome depends on the kind of chunk. If you put a value into a character or a word that doesn’t exist in a container, you put just the value. That is, if field 1 is empty, the statement

```
put "hey" into word 5 of field 1
```

puts hey (with no characters before it) into background field 1.

If you put a value into a nonexistent line, however, HyperCard puts in a return character, and if you put a value into a nonexistent item, HyperCard puts in a comma. (In both cases, you put a null chunk delimited by its particular delimiting character.) For example, if field 1 is empty, the statement

```
put "hey" into line 5 of field 1
```

puts four return characters (four null lines) followed by hey into background field 1. Similarly,

```
put "hey" into item 5 of field 2
```

puts four commas (four null items) followed by hey into the first line of background field 2.

Chapter Summary

Here is a summary of the material covered in this chapter:

- Complex expressions are built with values and operators.
- Operators are used to manipulate and calculate values.
- Chunk expressions can specify any chunk—character, word, item, or line—either in a source of value or as the destination of a put command.
This chapter describes the messages HyperCard sends in response to events, such as mouse clicks or Apple event commands, that you or an external process initiate in its environment.

Most system messages are sent by HyperCard to the current card, but those having to do with a specific button or field are sent to that object. The receiving object has the first chance to respond to the message before it goes on to the next encompassing object, as described in Chapter 4, “Handling Messages.” The receiving object can respond to the system message with a handler that begins

```
on messageName
```

where `messageName` is one of the system messages in the lists in this chapter.

**Messages and Commands**

Most system messages are informational—they cause no action if passed all the way to HyperCard, although they may be a result of a HyperTalk command executing. For example, HyperCard sends `deleteButton` to a button while it is executing either a Cut Button or Clear Button menu command. The `deleteButton` message is a result of a command, not the command itself. Consequently, you can’t prevent the deletion of buttons by intercepting the `deleteButton` message with a handler named `deleteButton`. All system messages that are a result of a command can be intercepted, but the intercepting handler will have no effect on the action of the command that sends that message.
Other system messages, however, are commands if passed to HyperCard. For example, all menu commands are passed to HyperCard as parameters of the doMenu message. (So you can prevent deletion of buttons by intercepting doMenu. But see the section “Redefining Commands” at the beginning of Chapter 10 before trying it.) All system messages that are HyperTalk commands are noted as such in this chapter and are also listed in Chapter 10, “Commands.” If a message that reaches HyperCard is neither a system message nor a command, HyperCard displays a “Can’t understand” error dialog box.

Although system messages are usually sent by HyperCard, they can be sent by other objects as well. For example, a handler could invoke a mouseUp handler in another object by executing a statement such as

\[
\text{send "mouseUp" to button 1 of card 1}
\]

The tables in this chapter correspond to the type of object to which the listed system messages are sent initially. If that object has no handler with a name matching the system message, it passes the message on to succeeding objects in the hierarchy. So, for example, a card can have a handler for a message sent initially to a button.

### Messages Sent to a Button

The only messages that are sent initially to buttons are those having to do with a specific button. They are of two types: those announcing the button’s creation or deletion, and mouse messages.

All of the mouse messages that can be sent to buttons can also be sent to fields. When buttons and fields are layered on top of each other, mouse messages are sent only to the closest one. Background buttons and fields can never overlay those belonging to the card. Both background buttons and card buttons precede the card in the message-passing hierarchy even though the background itself comes after the card.

Table 8-1 shows the system messages HyperCard sends initially to buttons.
## System Messages

### Table 8-1  Messages sent to a button

<table>
<thead>
<tr>
<th>Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>deleteButton</td>
<td>Sent to a button that is being deleted just before it disappears.</td>
</tr>
<tr>
<td>mouseDoubleClick</td>
<td>Sent to a button after a second mouse click is released when all of the following occur:</td>
</tr>
<tr>
<td></td>
<td>- The second click is within the double-click time interval set in the Mouse control panel.</td>
</tr>
<tr>
<td></td>
<td>- The second click is at a location within 4 pixels of the first click.</td>
</tr>
<tr>
<td></td>
<td>- The second click is within the same object as the first.</td>
</tr>
<tr>
<td></td>
<td>When the <code>mouseDoubleClick</code> message is sent, it’s the only system message sent as a result of the second click.</td>
</tr>
<tr>
<td></td>
<td>If someone clicks repeatedly, faster than the double-click speed, each odd-numbered click is treated as a first click and each even-numbered click is treated as a second click.</td>
</tr>
<tr>
<td>mouseDown</td>
<td>Sent to a button when the mouse button is pressed down while the pointer is inside its rectangle. (This message may also be sent to a field or card; see Table 8-2 and Table 8-3.)</td>
</tr>
<tr>
<td>mouseEnter</td>
<td>Sent to a button as soon as the pointer is moved within its rectangle. (This message may also be sent to a field; see Table 8-2.)</td>
</tr>
<tr>
<td>mouseLeave</td>
<td>Sent to a button as soon as the pointer is moved outside its rectangle. (This message may also be sent to a field; see Table 8-2.)</td>
</tr>
<tr>
<td>mouseStillDown</td>
<td>Sent to a button repeatedly while the mouse button is held down and the pointer is inside its rectangle. (This message may also be sent to a field or card; see Table 8-2 and Table 8-3.)</td>
</tr>
</tbody>
</table>
CHAPTER 8

System Messages

Table 8-1 Messages sent to a button (continued)

<table>
<thead>
<tr>
<th>Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>mouseUp</td>
<td>Sent to a button when the mouse button is released while the pointer is inside its rectangle. The pointer must be in the same button rectangle it was in when the mouse button was pressed down for the message to be sent. (This message may also be sent to a field or card; see Table 8-2 and Table 8-3.)</td>
</tr>
<tr>
<td>mouseWithin</td>
<td>Sent to a button repeatedly while the pointer is inside its rectangle. (This message may also be sent to a field; see Table 8-2.)</td>
</tr>
</tbody>
</table>
| newButton | Sent to a button as soon as it has been created; because a button can have no script with which to respond to this message (unless it was created by pasting), the message passes to objects lower in the hierarchy that can respond with handlers such as on newButton
  set autoHilite of the target to true
end newButton |

Messages Sent to a Field

The only messages that are sent initially to fields are those having to do with a specific field. They are of three types: those announcing the field’s creation or deletion, those announcing its opening for text entry or closing afterward, and mouse messages.

All of the mouse messages that can be sent to fields can also be sent to buttons. When buttons and fields are layered on top of each other, mouse messages are sent only to the closest one. Background buttons and fields can never overlay those belonging to the card. Both background fields and card fields precede the card in the message-passing hierarchy even though the background itself comes after the card.
Table 8-2 shows the system messages HyperCard sends initially to fields.

<table>
<thead>
<tr>
<th>Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>closeField</td>
<td>Sent to an unlocked field when it is closed after text editing by clicking outside the field, moving the text insertion point to the next field with the Tab key, pressing the Enter key, clicking a button, going to another card, or quitting HyperCard. The message is not sent unless some text was actually changed.</td>
</tr>
<tr>
<td>deleteField</td>
<td>Sent to a field that is being deleted just before it disappears.</td>
</tr>
<tr>
<td>enterInField</td>
<td>Sent to a field when the Enter key is pressed while there is an insertion point or selection in the field. If enterInField is not intercepted by a handler and the contents of the field have been changed, HyperCard sends the closeField message.</td>
</tr>
<tr>
<td>exitField</td>
<td>Sent to an unlocked field when it is closed without having its text changed.</td>
</tr>
<tr>
<td>mouseDoubleClick</td>
<td>Sent to a locked field after a second mouse click is released when all of the following occur:</td>
</tr>
</tbody>
</table>

- The second click is within the double-click time interval set in the Mouse control panel.
- The second click is at a location within 4 pixels of the first click.
- The second click is within the same object as the first.

When the mouseDoubleClick message is sent, it's the only system message sent as a result of the second click.

If someone clicks repeatedly, faster than the double-click speed, each odd-numbered click is treated as a first click and each even-numbered click is treated as a second click.

continued
CHAPTER 8
System Messages

Table 8-2  Messages sent to a field (continued)

<table>
<thead>
<tr>
<th>Message</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| mouseDown    | Sent to a locked field when the mouse button is pressed down while the pointer is inside it.  
MouseDown is not sent to a scrolling field when the mouse is clicked while the pointer is in the scroll bar. You can send mouseDown to an unlocked field by holding down the Command key while clicking the mouse in the field. (This message may also be sent to a button or card; see Table 8-1 and Table 8-3.) |
| mouseEnter   | Sent to a field as soon as the pointer is moved into it. (This message may also be sent to a button; see Table 8-1.)                        |
| mouseLeave   | Sent to a field as soon as the pointer is moved outside it. (This message may also be sent to a button; see Table 8-1.)                      |
| mouseStillDown| Sent to a locked field repeatedly while the mouse button is held down and the pointer is inside it. (This message may also be sent to a button or card; see Table 8-1 and Table 8-3.) |
| mouseUp      | Sent to a locked field when the mouse button is released while the pointer is inside it. The pointer must be in the same field it was in when the mouse button was pressed down for the message to be sent. (This message may also be sent to a button or card; see Table 8-1 and Table 8-3.) |
| mouseWithin  | Sent to a field repeatedly while the pointer is inside it. (This message may also be sent to a button; see Table 8-1.)                       |
| newField     | Sent to a field as soon as it has been created.                                                                                         |
| openField    | Sent to an unlocked field when it is opened for text editing by clicking the field or moving the text insertion point from the previous field with the Tab key. |

continued
CHAPTER 8
System Messages

Table 8-2 Messages sent to a field (continued)

<table>
<thead>
<tr>
<th>Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>returnInField</td>
<td>Sent to a field when the Return key is pressed while there is an insertion point or selection in the field. In response to a returnInField message, HyperCard sends a tabKey message to the field if the following conditions are true:</td>
</tr>
<tr>
<td></td>
<td>■ The field’s autoTab property (described in Chapter 12, “Properties”) is true.</td>
</tr>
<tr>
<td></td>
<td>■ The ReturnInField message is not intercepted by a handler.</td>
</tr>
<tr>
<td></td>
<td>■ The field is not a scrolling field.</td>
</tr>
<tr>
<td></td>
<td>■ The insertion point or selection is on the last line.</td>
</tr>
<tr>
<td></td>
<td>Otherwise, HyperCard inserts a return character into the field. The tabKey message, if it’s not intercepted, causes HyperCard to place the insertion point in the next field.</td>
</tr>
<tr>
<td>tabKey</td>
<td>Sent to a field when the Tab key is pressed while the text insertion point is in the field. (This message may also be sent to the current card; see Table 8-3.)</td>
</tr>
</tbody>
</table>

Messages Sent to the Current Card

System messages not sent to buttons or fields are sent initially to the current card, even when they concern the background or the stack.

Mouse messages are sent to the card only when there is no button or field, belonging to either the card or the background, under the pointer.

Table 8-3 shows the system messages HyperCard sends initially to the current card.
Table 8-3  Messages sent to the current card

<table>
<thead>
<tr>
<th>Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>appleEvent class, id, sender</td>
<td>Sent to the current card when an Apple event is received.</td>
</tr>
<tr>
<td></td>
<td>In an appleEvent message, class is the general category of the event (aevt, misc), id is the actual event received (oapp, odoc, pdoc, clos, quit, dosc, eval), and sender is the name of the application or process that sent the event.</td>
</tr>
<tr>
<td></td>
<td>Since Apple event commands are usually generated by other processes, you may want to check to make sure that they are not destructive. You can intercept an appleEvent message with an Apple event handler like this one written for the Home stack:</td>
</tr>
<tr>
<td></td>
<td>on appleEvent class, id, sender</td>
</tr>
<tr>
<td></td>
<td>answer &quot;Apple Event Alert!&quot; &amp; return &amp; ¬</td>
</tr>
<tr>
<td></td>
<td>&quot;The class is&quot; &amp;&amp; class &amp; return &amp; ¬</td>
</tr>
<tr>
<td></td>
<td>&quot;The ID is&quot; &amp;&amp; id &amp; return &amp; ¬</td>
</tr>
<tr>
<td></td>
<td>&quot;The Sender is&quot; &amp;&amp; sender &amp; return &amp; ¬</td>
</tr>
<tr>
<td></td>
<td>with &quot;Pass&quot; or &quot;Kill&quot;</td>
</tr>
<tr>
<td></td>
<td>if It is &quot;pass&quot; then pass appleEvent</td>
</tr>
<tr>
<td></td>
<td>end appleEvent</td>
</tr>
<tr>
<td></td>
<td>If you pass the event on at the end of your handler to HyperCard using the pass keyword, HyperCard executes the event; otherwise, the event is not executed.</td>
</tr>
<tr>
<td>arrowKey var</td>
<td>Sent to the current card when an arrow key is pressed (see also the textArrows property in Chapter 12, &quot;Properties&quot;). The value passed into the parameter variable var can be left, right, up, or down, depending on which arrow key is pressed. The beginning of a handler for this message could read</td>
</tr>
<tr>
<td></td>
<td>on arrowKey whichKey</td>
</tr>
<tr>
<td></td>
<td>if whichKey = &quot;left&quot; then go previous card</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>(This message is also a HyperTalk command; see Chapter 10, &quot;Commands.&quot;)</td>
</tr>
<tr>
<td>close</td>
<td>Sent to the current card, just before leaving that card, when you close a stack window with the close window command and when you click the close box of a card window.</td>
</tr>
</tbody>
</table>

continued
### Table 8-3  Messages sent to the current card (continued)

<table>
<thead>
<tr>
<th>Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>closeBackground</td>
<td>Sent to the current card, just before leaving that card, when a background is closed by going to another card that has a different background.</td>
</tr>
<tr>
<td>closeCard</td>
<td>Sent to the current card just before leaving that card.</td>
</tr>
<tr>
<td>closePalette</td>
<td>Sent to the current card when a palette that was opened with the palette command is closed.</td>
</tr>
<tr>
<td>closePicture</td>
<td>Sent to the current card when a window that was created with the picture command is closed.</td>
</tr>
<tr>
<td>closeStack</td>
<td>Sent to the current card, just before leaving that card, when a stack is closed by opening another stack.</td>
</tr>
<tr>
<td>commandKeyDown</td>
<td>Sent to the current card when a combination of the Command key and another key is pressed. The parameter variable <code>var</code> can be any character on the keyboard. The beginning of a handler for this message could read</td>
</tr>
<tr>
<td></td>
<td>on commandKeyDown whichKey</td>
</tr>
<tr>
<td></td>
<td>if whichKey = j then doMenu &quot;MyMenu&quot;</td>
</tr>
<tr>
<td></td>
<td>-- more statements...</td>
</tr>
<tr>
<td>controlKey</td>
<td>Sent to the current card when a combination of the Control key and another key is pressed. The possible values of the parameter value <code>var</code> and the keys each value corresponds to are shown in Appendix D, “Extended ASCII Table.” (See also the controlKey command in Chapter 10, “Commands.”) The beginning of a handler for this message could read</td>
</tr>
<tr>
<td></td>
<td>on controlKey whichKey</td>
</tr>
<tr>
<td></td>
<td>if whichKey = 16 then doMenu &quot;Print Card&quot;</td>
</tr>
<tr>
<td></td>
<td>-- additional statements...</td>
</tr>
<tr>
<td>deleteBackground</td>
<td>Sent to the current card when a background is deleted just before it disappears.</td>
</tr>
<tr>
<td>deleteCard</td>
<td>Sent to a card that is being deleted just before it disappears.</td>
</tr>
</tbody>
</table>

---

**continued**
### Table 8-3  Messages sent to the current card (continued)

<table>
<thead>
<tr>
<th>Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>deleteStack</code></td>
<td>Sent to the current card when a stack is deleted just before it disappears.</td>
</tr>
<tr>
<td><code>doMenu var1, var2</code></td>
<td>Sent to the current card when a menu item is selected. The parameter variable <code>var1</code> has the exact name of the menu item selected, including the three periods following menu items that invoke dialog boxes. Uppercase and lowercase don’t matter, but you must type the three periods—don’t use the Option-sicolon ellipsis character. The second parameter variable, <code>var2</code>, has the exact name of the menu in the menu bar. (This message is also a HyperTalk command, which is listed in Chapter 10, “Commands.” An example handler to intercept the <code>doMenu</code> message is shown in the section “Redefining Commands,” at the beginning of Chapter 10.)</td>
</tr>
<tr>
<td><code>enterKey</code></td>
<td>Sent to the current card when the Enter key is pressed unless the text insertion point is in a field. (This message is also a HyperTalk command; see Chapter 10.)</td>
</tr>
</tbody>
</table>
| `functionKey var` | Sent to the current card when a function key on the Apple Extended Keyboard is pressed. The parameter variable `var` can range from 1 to 15. Function keys 1 through 4 are preprogrammed for the Undo, Cut, Copy, and Paste commands, respectively. The beginning of a handler for this message could read:  
```on functionKey whichKey  
    if whichKey < 5 then pass functionKey  
    else if whichKey is 5 then doMenu "New Card"  
    else if whichKey is 6 then choose browse tool  
    else if whichKey is 7 then choose button tool  
```
You can override the preprogrammed functions of keys 1 through 4 in a `functionKey` message handler. (This message is also a HyperTalk command; see Chapter 10, “Commands.”) |
| `help` | Sent to the current card, just before leaving that card, when Help is chosen from the Go menu (or Command-? is pressed). You can intercept this message if you want to provide your own Help system for your stack. (This message is also a HyperTalk command; see Chapter 10, “Commands.”) |

*continued*
CHAPTER 8
System Messages

Table 8-3  Messages sent to the current card (continued)

<table>
<thead>
<tr>
<th>Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>hide menubar</td>
<td>Sent to the current card when the menu bar is visible and you press Command–Space bar. <em>(Hide is also a HyperTalk command; the command accepts other parameter variable values besides menubar; see its description in Chapter 10, “Commands.”)</em></td>
</tr>
<tr>
<td>idle</td>
<td>Sent to the current card repeatedly when nothing else is happening and the Browse tool is current. An idle handler can interfere with typing. For example, if you have an idle handler that puts text into a field, it can remove the insertion point from another field while the user is typing. Here is an example of such a handler: on idle    put the time into card field &quot;Time&quot;    pass idle end idle If this on idle handler were to execute during typing into another field <em>(idle is sent during a typing pause)</em>, and if the time had changed, HyperCard would remove the insertion point from the user’s field. The user would have to click the field or press the Tab key to replace the insertion point after every pause, which would be annoying and tedious.</td>
</tr>
</tbody>
</table>
| keyDown var | Sent to the current card when a key is pressed. The parameter variable var can be any character on the keyboard. The beginning of a handler for this message could read on keyDown whichKey    if whichKey = t then put "That's the key" -- more stuff...

The keyDown message is not sent for keys that generate special messages or for programmed function keys. A programmed function key would send a functionKey message and any additional messages that the specified function key is programmed to generate. See also the functionKey message description.  

continued

Messages Sent to the Current Card 135
### Table 8-3  Messages sent to the current card (continued)

<table>
<thead>
<tr>
<th>Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>mouseDoubleClick</td>
<td>Sent to a card, after a second mouse click is released, when all of the following occur:</td>
</tr>
<tr>
<td></td>
<td>■ The second click is within the double-click time interval set in the Mouse control panel.</td>
</tr>
<tr>
<td></td>
<td>■ The second click is at a location within 4 pixels of the first click.</td>
</tr>
<tr>
<td></td>
<td>■ The second click is within the same object as the first.</td>
</tr>
<tr>
<td></td>
<td>When the <code>mouseDoubleClick</code> message is sent, it's the only system message sent as a result of the second click.</td>
</tr>
<tr>
<td></td>
<td>If someone clicks repeatedly, faster than the double-click speed, each odd-numbered click is treated as a first click and each even-numbered click is treated as a second click.</td>
</tr>
<tr>
<td>mouseDown</td>
<td>Sent to the current card when the mouse button is pressed down and the pointer is not in a button rectangle or field. (This message may also be sent to a button or field; see Table 8-1 and Table 8-2.)</td>
</tr>
<tr>
<td>mouseDownInPicture</td>
<td>Sent to the current card when the mouse button is held down while the pointer is in a window created with the <code>picture</code> command. See also the <code>picture</code> command in Chapter 10, “Commands.”</td>
</tr>
<tr>
<td>pictureWindowName,</td>
<td></td>
</tr>
<tr>
<td>point</td>
<td></td>
</tr>
<tr>
<td>mouseStillDown</td>
<td>Sent to the current card repeatedly while the mouse button is held down. (This message may also be sent to a button or field; see Table 8-1 and Table 8-2.)</td>
</tr>
<tr>
<td>mouseUp</td>
<td>Sent to the current card when the mouse button is released. (This message may also be sent to a button or field; see Table 8-1 and Table 8-2.)</td>
</tr>
<tr>
<td>mouseUpInPicture</td>
<td>Sent to the current card when the mouse button is released after being down while the pointer is in a window created with the <code>picture</code> command. See also the <code>picture</code> command in Chapter 10, “Commands.”</td>
</tr>
<tr>
<td>pictureWindowName,</td>
<td></td>
</tr>
<tr>
<td>point</td>
<td></td>
</tr>
<tr>
<td>moveWindow</td>
<td>Sent when you change a card window’s <code>location</code> property with HyperTalk, drag or zoom the card window, or change the location of the card window with the <code>show</code> command. See also the <code>location</code> and <code>rectangle</code> properties in Chapter 12, “Properties,” and the <code>show</code> command in Chapter 10, “Commands.”</td>
</tr>
</tbody>
</table>

*continued*
### Table 8-3 Messages sent to the current card (continued)

<table>
<thead>
<tr>
<th>Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>newBackground</td>
<td>Sent to the current card as soon as a background has been created.</td>
</tr>
<tr>
<td>newCard</td>
<td>Sent to a card as soon as it has been created.</td>
</tr>
<tr>
<td>newStack</td>
<td>Sent to the current card when a stack is created.</td>
</tr>
<tr>
<td>openBackground</td>
<td>Sent to the current card when a background is first opened by going to a card whose background is different than that of the previous card.</td>
</tr>
<tr>
<td>openCard</td>
<td>Sent to a card when you go to it.</td>
</tr>
<tr>
<td>openPalette</td>
<td>Sent to the current card when a palette is opened with the palette command.</td>
</tr>
<tr>
<td>openPicture</td>
<td>Sent to the current card when a window is created with the picture command.</td>
</tr>
<tr>
<td>openStack</td>
<td>Sent to the current card when a stack is opened by going to a card in a different stack than that of the previous card. In this case the following three messages are sent, in order: openCard, openBackground, and openStack.</td>
</tr>
<tr>
<td>quit</td>
<td>Sent to the current card when you choose Quit HyperCard from the File menu (or press Command-Q) just before HyperCard quits.</td>
</tr>
<tr>
<td>resume</td>
<td>Sent to the current card when HyperCard resumes running after having been suspended.</td>
</tr>
<tr>
<td>resumeStack</td>
<td>Sent to the current card when you return to an already open stack.</td>
</tr>
<tr>
<td>returnKey</td>
<td>Sent to the current card when the Return key is pressed unless the text insertion point is in a field. (This message is also a HyperTalk command; see Chapter 10, “Commands.”)</td>
</tr>
<tr>
<td>show menubar</td>
<td>Sent to the current card when the menu bar is hidden and you press Command–Space bar. (Show is also a HyperTalk command; the command accepts other parameter variable values besides menubar; see the description in Chapter 10, “Commands.”)</td>
</tr>
</tbody>
</table>

*continued*
CHAPTER 8
System Messages

Table 8-3  Messages sent to the current card (continued)

<table>
<thead>
<tr>
<th>Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>sizeWindow</td>
<td>Sent to the current card when the card window is resized, such as in the following cases:</td>
</tr>
<tr>
<td></td>
<td>• The window is resized from the size box or scroll palette.</td>
</tr>
<tr>
<td></td>
<td>• The window is zoomed in or out, thereby changing the window size.</td>
</tr>
<tr>
<td></td>
<td>• A script sets the one of the rectangle properties (height, width, and so on) of the card window to a new rectangle.</td>
</tr>
<tr>
<td></td>
<td>• A script sets the rect of the card to a new rectangle and the resizing of the card causes resizing of the card window.</td>
</tr>
<tr>
<td></td>
<td>In all cases except the first, a moveWindow message also may be sent. When both messages are pending, sizeWindow is sent first.</td>
</tr>
<tr>
<td>suspend</td>
<td>Sent to the current card when HyperCard is suspended by launching another application with the open command just before the other application is launched.</td>
</tr>
<tr>
<td>suspendStack</td>
<td>Sent to the current card, just before leaving that card, when you switch to another stack.</td>
</tr>
<tr>
<td>startUp</td>
<td>Sent to the first card displayed when HyperCard is first started.</td>
</tr>
<tr>
<td>tabKey</td>
<td>Sent to the current card when the Tab key is pressed unless the text insertion point is in a field. In that case, tabKey is sent initially to the field; see Table 8-2. (This message is also a HyperTalk command; see Chapter 10, “Commands.”)</td>
</tr>
</tbody>
</table>

Message Order

For some events, HyperCard sends a sequence of system messages. The messages are sent in a specific message order. You can create message handlers that use the message sending order to set properties or perform other actions when opening stacks, creating new backgrounds, creating new cards, deleting cards, backgrounds, or stacks, moving from card to card, and moving from
CHAPTER 8

System Messages

stack to stack. For example, the `startUp` system message is always sent when HyperCard first starts up and the first stack is opened, so you can create a handler for the `startUp` system message that sets properties for the first card before it appears.

You can create handlers for any message that is sent earlier in the message sending order to change the actions that take place before the messages that follow it are acted upon by HyperCard. The HyperCard message sending order that results from some of the actions most frequently performed when creating or modifying stacks is shown in Table 8-4. You can also open the Message Watcher to watch the messages that result from other actions.

<table>
<thead>
<tr>
<th>Action</th>
<th>Resulting message order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create a new background</td>
<td><code>closeCard</code>, <code>closeBackground</code>, <code>newBackground</code>, <code>newCard</code>, <code>openBackground</code>, <code>openCard</code></td>
</tr>
<tr>
<td>Create a new card</td>
<td><code>closeCard</code>, <code>newCard</code>, <code>openCard</code></td>
</tr>
<tr>
<td>Create a new stack</td>
<td><code>closeCard</code>, <code>closeBackground</code>, <code>closeStack</code>, <code>newStack</code>, <code>newBackground</code>, <code>newCard</code>, <code>openStack</code>, <code>openBackground</code>, <code>openCard</code></td>
</tr>
<tr>
<td>Cut a card</td>
<td><code>closeCard</code>, <code>closeBackground</code>, <code>deleteCard</code>, <code>deleteBackground</code>, <code>openBackground</code>, <code>openCard</code></td>
</tr>
<tr>
<td>Delete a background</td>
<td><code>closeCard</code>, <code>closeBackground</code>, <code>deleteCard</code>, <code>deleteBackground</code>, <code>openBackground</code>, <code>openCard</code></td>
</tr>
<tr>
<td>Delete a card</td>
<td><code>closeCard</code>, <code>closeBackground</code>, <code>deleteCard</code>, <code>deleteBackground</code>, <code>openBackground</code>, <code>openCard</code></td>
</tr>
<tr>
<td>Delete a stack</td>
<td><code>closeCard</code>, <code>closeBackground</code>, <code>closeStack</code>, <code>deleteStack</code></td>
</tr>
<tr>
<td>Paste a card</td>
<td><code>closeCard</code>, <code>closeBackground</code>, <code>newBackground</code>, <code>newCard</code>, <code>openBackground</code>, <code>openCard</code></td>
</tr>
<tr>
<td>Resume HyperCard</td>
<td><code>resume</code>, <code>openStack</code>, <code>openBackground</code>, <code>openCard</code></td>
</tr>
<tr>
<td>Start up HyperCard</td>
<td><code>startUp</code>, <code>openStack</code>, <code>openBackground</code>, <code>openCard</code></td>
</tr>
</tbody>
</table>
This chapter describes the nature of control structures and the control structure vocabulary, which is made up of HyperCard keywords.

A keyword is a word whose meaning is predefined in HyperTalk. You cannot redefine keywords as variable names. Keywords are not sent as messages when they execute in scripts, nor can they be used in the Message box (except for do, if, and send). Some keywords provide the structure for handlers; others control the flow of execution within handlers.

HyperTalk has two kinds of handlers: message and function handlers, denoted by the initial keywords on and function, respectively. Message and function handlers are defined in the same way (except for the initial keyword), but they differ in how they are invoked and in how they return values.

The syntax for each keyword is given. In the syntax statements, words in italic represent general elements that you replace with a specific instance when you write a statement; brackets ( [ ] ) denote optional elements (don’t type the brackets).

**Keywords in Message Handlers**

The on keyword identifies a HyperCard message handler. Message handlers are written to define your own messages, or to modify or redefine what happens in response to any message (including a HyperTalk command). The general syntax of a message handler looks like this:

```plaintext
on  messageName  [parameterList]
    statementList
end  messageName
```
CHAPTER 9

Control Structures and Keywords

*MessageName* is an identifier: a string starting with a letter and containing no spaces or punctuation marks except underscore; *parameterList* is a series of zero or more parameter variables (separated by commas if more than one); and *statementList* is one or more HyperTalk statements.

The handler dictates the method by which its object responds to *messageName*. When a message called *messageName* is sent to an object, HyperCard checks all of that object’s message handlers to see if it has one named *messageName*. If so, the object responds according to that handler, and the message is sent no further (assuming the script has no *pass* statement, described later in this chapter). If the object has no handler to match *messageName*, HyperCard passes the message to the next object in the hierarchy.

**You can override HyperTalk**

If you name a message handler the same as a built-in command, your name overrides the built-in one if yours is anywhere along the message-passing hierarchy between the object sending the message and HyperCard.

The statements in the handler execute until an *end*, *exit*, *pass*, or *return* keyword is reached (these keywords are discussed later in this section). A message handler can return a value through the built-in function *the result* (discussed in Chapter 11, “Functions”).

---

**On**

```
on messageName [parameterList]
```

The *on* keyword marks the beginning of a message handler and connects the handler with a particular message. *MessageName* is the first word of the message to which the handler responds, and it is the name of the handler.

The optional *parameterList* allows the message handler to receive some values sent along with the message. This list is a series of local variable names, called *parameter variables*, separated by commas. When the message is sent, each source of value following the message name is evaluated; when the handler receives the message, each value is plugged into the parameter variable that appears in the corresponding position following *on messageName*. The first value in the list goes into the first parameter variable, and so on. If no values are sent with the message name, *empty* goes into each parameter variable.
Chapter 4, “Handling Messages,” explains more about parameter passing. See also the `param`, `paramCount`, and `params` functions in Chapter 11, “Functions.”

### End

```markdown
end `messageName`
```

The `end` keyword begins the last line of a handler—it is reached when all of the handler’s statements have been executed (except for any bypassed conditional blocks). When the `end` statement is reached, the message that initiated execution of the handler is sent no further unless a `pass `messageName` statement follows the `end` statement. (`Pass` is defined later in this section.) If the message that initiated this handler’s execution was part of some other handler, control passes back to the other handler.

### Exit

```markdown
exit `messageName`
exit to HyperCard
```

The `exit `messageName` statement ends execution of the handler.

The `exit to HyperCard` form makes program flow return directly to HyperCard, bypassing any pending handlers that have not finished executing.

### Pass

```markdown
pass `messageName`
```

The `pass `messageName` statement ends execution of the handler and sends the entire message that initiated execution of the handler to the next object in the hierarchy. (Ordinarily, a message is sent no further than the object containing the executing handler.)
Return

\[ \text{return expression} \]

The `return` statement ends execution of the handler and, when it appears within a message handler structure, places the value of `expression` into the HyperTalk function `the result`.

The value of the `result` set by the `return` statement is valid only immediately after it executes; each new statement resets the `result`. (See the `result` function in Chapter 11, “Functions,” for examples.)

Message Handler Example

The following example shows a handler that originates a message that in turn initiates execution of a second handler. (The second handler could be in the same script as the first or anywhere farther along the object hierarchy.)

```hyperTalk
on mouseUp
    heyNow 5,10 -- heyNow is the message name that's sent
end mouseUp

on heyNow timeVar,timeVar2 --Handler name heyNow matches message name
    play "boing" tempo 200 "c4e c dq c f eh" -- Happy Birthday
    wait timeVar seconds
    play stop
    play "harpsichord" "ch d e f g a b c5w"
    wait timeVar2 seconds
    play stop
end heyNow
```

Execution of the first handler is initiated when its object receives a `mouseUp` message. The `mouseUp` message could be generated when the user clicks the mouse or types `mouseUp` in the Message box and presses Return. It could also originate from another handler executing the statement `mouseUp` or could be sent explicitly to the handler’s object with a send command.
When the `mouseUp` handler executes, it sends its one command statement (
`heyNow 5, 10`) as a message, first to its own object. The message name (the
first word of the message) matches the handler name (the word following `on` in
the first line of the handler), so the statements in the second handler begin
executing. (If the current object had no `heyNow` message handler, that object
would pass the entire message on to the next object in the hierarchy.)

The values of the parameters following `heyNow` in the first handler are passed
into the parameter variables following `heyNow` in the second handler. So when
the second handler is executing, `timeVar` has the value 5, and `timeVar2` has
the value 10.

### Keywords in Function Handlers

A **function** is a named value that is calculated by HyperCard when a statement
in which it’s used executes. The function keyword identifies a HyperCard
function handler. You can use this structure to define your own functions, which
then can be called from any place in a statement where their values are needed.
(User-defined functions are called like built-in HyperCard functions except that
you must always use parentheses—see “Return,” later in this section.)

Like message handlers, function handlers cannot be nested inside each other
(or inside message handlers). The general syntax of a function handler looks
like this:

```hypercard
function function_name [parameter_List]
    statement_list
end function_name
```

*FunctionName* is an identifier: a string starting with a letter and containing no
spaces or punctuation marks except underscore; *parameterList* is a series of zero
or more parameter variables separated by commas; and *statementList* is one or
more HyperTalk statements.

User-defined function handlers use the message-passing hierarchy in the same
way as do message handlers. That is, when the function name appears in a
statement or in the Message box, HyperCard searches through all of the scripts
along the current message-passing hierarchy for a matching function handler. If
Keywords in Function Handlers

a match is found, the function handler executes. If none is found, the function call is passed to HyperCard; if there is no built-in function of that name, HyperCard displays an error dialog box.

If you name a function handler the same as a built-in function, your function is called when you use the function call syntax that uses parentheses. Of course, your function handler must also be in the script of an object lower or equal in the hierarchy than the originator of the function call. You can make calls to built-in functions using the function call syntax with the preceding the function name, which bypasses the message-passing hierarchy and calls HyperCard built-in functions directly.

Program flow runs through the function handler until it encounters an end, exit, pass, or return statement (discussed later in this section). A function handler returns a value directly into the statement in which its name was used.

Function

```
function functionName [parameterList]
```

The `function` keyword marks the beginning of a function handler and connects the handler with a particular function call. `FunctionName` is the function call to which the handler responds, and it is the name of the handler.

The optional `parameterList` allows the function handler to receive some values sent along with the function call. This list is a series of local variable names, called `parameter variables`, separated by commas. When the function call is made, each source of value appearing between parentheses following the function name is evaluated; when the handler begins to execute, each value is plugged into the parameter variable that appears in the corresponding position following `function functionName`, the first value in the list going into the first parameter variable, and so on.

For more details on passing parameters to function handlers, see “Return,” later in this section.
End

```
end functionName
```

The `end` statement is the last line of the handler, reached when all of the handler’s statements have been executed (except for any bypassed conditional blocks).

When the `end` statement is reached, control passes back to the handler containing the function call that originated the function handler’s execution.

Exit

```
exit functionName
exit to HyperCard
```

The `exit functionName` statement ends execution of the handler.

The `exit to HyperCard` form makes program flow return directly to HyperCard, bypassing any pending handlers that have not finished executing, including the handler containing the function call.

Pass

```
pass functionName
```

The `pass` statement ends execution of the handler and sends the entire function call that initiated execution of the handler to the next object in the hierarchy. (Ordinarily, a function call is sent no further than the object containing the executing handler.)
Return

return expression

The return statement ends execution of the handler and, when it appears within a function handler structure, dictates the returned value of the function. The value of expression replaces the function in the calling statement.

The function appears in the calling statement in the form functionName(expressionList):

put square(17) into card field 1

ExpressionList is a series of zero or more expressions separated by commas whose values are assigned to the parameter variables in the parameterList of the function handler. In the above example, the expressionList comprises only the number 17.

A user-defined function handler that would respond to the function call example square(17), shown above, is

function square x
  return x * x
end square

In this example, the function handler has one parameter variable to receive one value passed to it by the calling statement. The value 17 is passed to the function handler, where it is assigned to the parameter variable x; the value of x * x is returned by the return statement, replacing square(17) in the calling statement. So, the effect of the calling statement is to put the value 289 into card field 1.

Parentheses required

User-defined functions are always followed by parentheses (which are empty if there are no parameters to pass). Unlike built-in functions (explained in detail in Chapter 11), user-defined functions can’t be called with the form the function of. ♦
CHAPTER 9

Control Structures and Keywords

Function Handler Example

The following function handler determines whether a number passed to it as a parameter is even or odd, returning the constant true if it's even or false if it's odd:

function evenNumber numberPassed
    return numberPassed mod 2 is 0
end evenNumber

A calling statement that would invoke the evenNumber function handler could be one like the following:

if evenNumber(numberVariable) then add 1 to evenNumberCount

In the calling statement, numberVariable can be the name of any variable or other source of value (including an actual number). HyperCard evaluates numberVariable before it passes the function call along the hierarchy, and its value is given to the parameter variable numberPassed when the evenNumber function handler is found. The part of the calling statement following then is arbitrary—the point of the example is to show how the function handler receives a value, examines it, and returns another value into the calling statement, based on the result of its execution.

Repeat Structure

The repeat structure causes all of the HyperTalk statements between its first and last lines to execute in a loop until some condition is met or until an exit statement is encountered. The general syntax of a repeat structure looks like this:

repeat controlForm
    statementList
end repeat

ControlForm is one of the forms of the repeat statement described below, and statementList is any number of HyperTalk statements. Repeat structures can be used only within message handlers or function handlers.
Repeat Structure

If you want to try the examples in this chapter, be sure to put them within handlers.

Repeat Statements

The `repeat` keyword begins the first line of a repeat structure. The `repeat` statement has five forms differentiated by the second word of the statement. Additionally, the `repeat with` form has two variations.

**Repeat Forever**

```
repeat [forever]
```

The loop repeats forever, or until an `exit` statement is encountered (whichever comes first):

```
put 1 into Message box
repeat
    add 1 to Message box
    if Message box contains 6 then exit repeat
end repeat
```

The example ends with 6 in the Message box.

For information on `exit repeat`, see “Exit Repeat Statement,” later in this chapter. For information on `if`, see “If Structure,” later in this chapter.
Control Structures and Keywords

Repeat For

```
repeat [for] number [times]
```

_Number_ is a source that yields a positive integer specifying how many times the loop is executed:

```
put 1 into Message box
repeat for 5 times
   add 1 to Message box
end repeat
```

The example ends with 6 in the Message box.

Repeat Until

```
repeat until condition
```

_Condition_ is an expression that always evaluates to _true_ or _false_. The loop is repeated as long as the condition is _false_. The condition is checked prior to the first and any subsequent executions of the loop:

```
put 1 into Message box
repeat until Message Box contains 6
   add 1 to Message box
end repeat
```

The example ends with 6 in the Message box.
CHAPTER 9
Control Structures and Keywords

Repeat While

repeat while condition

Condition is an expression that evaluates to true or false. The loop is repeated as long as the condition is true. The condition is checked prior to the first and any subsequent executions of the loop:

put 1 into Message box
repeat while Message Box < 6
    add 1 to Message box
end repeat

The example ends with 6 in the Message box.

Repeat With

There are two variations of the repeat with form: one that increments a variable and one that decrements a variable.

repeat with variable = start to finish

Variable is a local or global variable name, and start and finish are sources of integers. The value of start is assigned to variable at the beginning of the loop and is incremented by 1 with each pass through the loop. Execution ends when the value of variable equals the value of finish.

repeat with increment = 1 to 6
    put increment into the Message box
end repeat

The example ends with 6 in the Message box. (This structure works much like a FOR...NEXT loop in BASIC.)

repeat with variable = start down to finish
The down to form is the same as the to form above, except that the value of variable is decremented by 1 with each pass through the loop. Execution ends when the value of variable equals the value of finish.

repeat with decrement = 6 down to 1
  put decrement into the Message box
end repeat

The example ends with 1 in the Message box.

**Exit Repeat**

exit repeat

The exit repeat statement sends control to the end of the repeat structure, ending execution of the loop regardless of the state of the controlling conditions specified in the repeat statement.

put 1 into the Message box
repeat with increment = 1 to 100
  add increment to the Message box
  if Message box > 20 then
    beep 5
    exit repeat
  end if
end if
end repeat

The example ends with 22 in the Message box.

An exit statement can appear anywhere within the structure.

For information on if, see “If Structure,” later in this chapter.
CHAPTER 9

Control Structures and Keywords

Next Repeat

next repeat

When a next repeat statement is encountered, control returns immediately to the top of the structure. (Usually, flow doesn’t return to the top of the repeat structure until an end statement is encountered.)

```plaintext
repeat 20
    put random(9) into tempVar
    if tempVar mod 2 = 0 then next repeat
    put tempVar after field "oddNumbers"
end repeat
```

The example appends only the odd random numbers to the field, skipping any even ones.

A next statement can appear anywhere within the repeat structure.

For information on if, see “If Structure,” later in this chapter. For more information about the random function, see Chapter 11, “Functions.”

End Repeat

end repeat

The end repeat statement marks the end of the loop; it’s the last line of a repeat control structure. When the controlling conditions specified in the repeat statement have been satisfied or an exit statement is encountered, control goes beyond the end statement:

```plaintext
repeat for 5 times
    beep
end repeat
```
If Structure

The if structure tests for the specified condition and, if the condition is true, executes the command statement or series of command statements that follow. The if structure has several forms, described in the following sections.

**Note**
If you want to try the examples in this section, be sure to put them within handlers.

### Single-Statement If Structure

A single-statement if structure has the form shown below:

```hypercard
if condition then statement [else statement]
```

A single-statement if structure can also occupy more than one line:

```hypercard
if condition
then statement
[else statement]
```

Within the if structure, condition is an expression that evaluates to true or false, and statement is a single HyperTalk command statement.

In the single-statement if structure, only one command statement can follow either then or else (if present), and the command statement must be on the same line with then or else.

If the condition between if and then is true, HyperCard executes the statement between then and else if else is present, or between then and the end of the line if else is not present following the statement, either on the same line or on the next line.
If the condition between if and then is false, HyperCard executes the statement between else and the end of the line if else is present, or it ignores the rest of the line if else is not present:

```hypercard
if Message box > 10 then beep 5 else beep 15
```

In this example, if the Message box holds a value greater than 10, the Macintosh beeps 5 times; if the value in the Message box is 10 or less, the Macintosh beeps 15 times.

Note

Single-statement if structures can be used in the Message box.

---

**Multiple-Statement If Structure**

A multiple-statement if structure accommodates more than one executable statement following then and, optionally, more than one statement following else:

```hypercard
if condition then
    statementList
[else
    statementList]
end if
```

You can also end a multiple-statement then clause with a single-line else, in which case no end if statement is needed:

```hypercard
if condition then
    statementList
else statement
```

*Condition* is an expression that evaluates to true or false, and *statementList* is any number of HyperTalk statements.
In the multiple-statement if structure, more than one command statement can follow either then or else (if present), and the first command statement must be on the line following then or else. That is, if you want to have more than one statement in a block following then or else, put a return character after the respective then or else. Such a multiple-statement block must be ended explicitly: a multiple-statement then block can be ended with either end if or else; a multiple-statement else block must be ended with end if.

If the condition between if and then is true, HyperCard executes the statement or statements between then and else if else is present, or between then and end if if else is not present.

If the condition between if and then is false, HyperCard executes the statements between else and end if if else is present, or it ignores what’s between then and end if if else is not present:

```plaintext
if number of this card is 10 then
    put "We're done!" into msg
    go Home
else
    put "And the next question is:" into msg
    go next card
end if
```

**Note**

Multiple-statement if structures can be used only within message handlers or function handlers.

**Nested If Structures**

If structures can be nested; that is, statements following a then or an else can include more if structures. Each nested multiple-line if structure must have its own end if, and an else always goes with the closest preceding if clause.
CHAPTER 9

Control Structures and Keywords

The next example could be used as a mouseUp handler within a button.

```
repeat
    ask "Guess a number between 1 and 10:"
    if it is empty then
        exit repeat
    else
        if it is random(10) then
            put "You guessed it!"
        else
            put "Sorry, try again."
        end if
    end if
end repeat
```

In this example, satisfying the if it is empty condition in the first if structure allows the user to exit the loop by selecting the OK button without entering a number in the dialog box that is created with the ask command. Executing this example without the first if structure results in an endless loop, which you can exit by simultaneously clicking the Cancel button and pressing the Command-period keys.

**Note**
Nested-statement if structures can be used only within message handlers or function handlers.

---

**Do**

```
do expression
do expression [as scriptingLanguage]
```

The do keyword causes HyperCard to get the value of expression; HyperCard then sends that value as a message. (The do keyword in HyperTalk does not work like the do in Pascal does.)
You can use the `do expression [as scriptingLanguage]` form to execute scripts in any scripting language supported by an OSA-compliant scripting component, as well as HyperTalk. Here are some examples:

```
do field 1 as AppleScript
do "people.dw.stuff = 1" as UserTalk
```

In versions of HyperCard before HyperCard 2.0, if `expression` was a field with more than one line, only the first line of the field was used by HyperCard and any lines that followed were ignored. HyperCard now evaluates any expression after the `do` keyword and sends it as a command. For example, if field 3 contains several lines of HyperTalk code and you say `do field 3`, the entire contents of field 3 are sent to HyperCard. Each line of the container executes just as if it were contained in a handler.

The `do` keyword can be used in the Message box.

---

**Global**

```
global variableList
```

*VariableList* is one or more HyperCard variable names separated by commas.

The `global` keyword makes a variable name known and its contents available to any script of any object in HyperCard. The following two lines are individual examples of `global` statements:

```
global myVar
global pages,sections,chapters
```

The following example handlers show a global variable being used for two handlers to access the same value:

```hs
on mouseUp
  global myVariable -- load the global here
  put 3 into myVariable
  writeResult
end mouseup
```
Control Structures and Keywords

on writeResult
    global myVariable -- now we can use the global
    put myVariable -- the value remains 3
end writeResult

Changing the value of a global variable in any script changes its value everywhere. The `global` keyword must be used in each handler in which the global variable is used.

Global variables are not saved in between sessions of HyperCard or when HyperCard is suspended by launching another application with the `open` command.

Send

send expression to program programName [with|without reply]
send expression to program ID programID [with|without reply]
send expression to this program [with|without reply]
send "messageName [parameterList]" [to object]

Expression is any valid expression or sequence of commands in the scripting language supported by the target program, programName is the pathname of the application to send the message to, programID is the signature of the application to send the message to, messageName is a string beginning with a letter and containing no spaces or punctuation marks other than underscore, parameterList is one or more expressions (separated by commas if more than one), and object is a HyperCard object descriptor or HyperCard itself. If no object is specified, the message continues along the message-passing hierarchy. (See Chapter 4, “Handling Messages,” for information on how the message-passing hierarchy works.)

The `send` keyword sends a message directly to a particular object, bypassing any handlers in the intervening message-passing hierarchy that might otherwise intercept the message.
CHAPTER 9

Control Structures and Keywords

send "hideIt" to field 3
send "addSums travel,food,hotel" to stack "expenseAccount"
send mouseUp to button "pushMe"
send "doMenu Print Card" to HyperCard
send "make waves" to program "FishingNet:MyHD:HyperCard"
send "build {project}" to program "MPW Shell" without reply

You can send a message directly to any object in the current stack or to another stack, but not to a specific object in another stack.

If the object has no message handler for messageName, the message is passed along the message-passing hierarchy stemming from the object to which the message was sent. If the object does have a matching handler, the handler executes, but the card to which it belongs does not necessarily open. Messages sent by executing the statements of the object’s handler are sent along the receiving object’s hierarchy.

The send expression to program form sends a "do script" Apple event from HyperCard to another application running remotely. You can use it to send a script to any program that understands the standard 'dosc' Apple event. By default, HyperCard waits for a reply from the target program before continuing. However, you can specify with the [without reply] option if you don’t want to wait for a reply.

If the target program is running on a different Macintosh you must specify the Macintosh name; if it is in a different zone you must specify that also. The form of such a full pathname is zone:Macintosh:program.

SCRIPT

The following handler directs a copy of HyperCard 2.2 on a Macintosh computer named “MyMac,” in the same zone as the sending computer, to go to the last card of its currently active stack. It then checks the result to make sure that the command executed:

on changeRemoteCard
    send "go last card" to program "MyMac:HyperCard 2.2"
    if the result is not empty -- problems?
        then answer "Error during send!"
end changeRemoteCard
The next handler sends a message to a copy of HyperCard running on a Macintosh computer named “Oahu” on a network zone named “Hawaii.” The message tells the remote copy of HyperCard to execute the handler “doThisNow,” already defined in the remote stack:

```pascal
on doRemoteHandler
    send "doThisNow" to program "Oahu:Hawaii:HyperCard 2.2"
end doRemoteHandler
```

NOTES

When sending Apple events to another program, if HyperCard has not established a link with the target program, the user is presented with a dialog box, through which the link is established. If a link has already been established between HyperCard and the target program, the Apple event is sent without further user interaction.

The following error messages go into the container the result of the source program when the send fails:

<table>
<thead>
<tr>
<th>Condition</th>
<th>the result contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Information returned is not recognized by HyperCard as text</td>
<td>Unknown data type</td>
</tr>
<tr>
<td>System software prior to version 7.0</td>
<td>Not supported by this version of the system</td>
</tr>
<tr>
<td>Target program didn’t handle event</td>
<td>Not handled by target program</td>
</tr>
<tr>
<td>Target program returned error number in reply, or AESend returned some other error</td>
<td>Got error &lt;errorNum&gt; when sending Apple event</td>
</tr>
<tr>
<td>Target program returned error string in reply</td>
<td>&lt;errorString&gt;</td>
</tr>
<tr>
<td>Target program timed out</td>
<td>Timeout</td>
</tr>
<tr>
<td>User canceled “Link to program” dialog</td>
<td>Cancel</td>
</tr>
</tbody>
</table>

Chapter 4, “Handling Messages,” has more information about how the send command interacts with the message-passing hierarchy.

Parameter expressions are evaluated before they are sent, even though the entire message has quotation marks around it.
The `send` keyword does not change cards when a message is sent to an object on a card other than the current card, or cause HyperCard to send `open` or `close` messages to cards, backgrounds, or stacks. For example, if you are on the second card and send a message to a button on the ninth card in the stack, the ninth card doesn’t get an `openCard` message.

**You can use it in the Message box**
The `do`, `if`, and `send` keywords, unlike all other keywords, work in the Message box.
CHAPTER 10

Commands

This chapter describes all the commands in HyperTalk, showing their syntax and meaning.

HyperTalk commands are built-in message handlers that reside in HyperCard itself. When you issue a HyperTalk command, it’s passed along the message-passing hierarchy as a message to HyperCard. In most cases there’s no handler in any script along the way to intercept the message, so HyperCard receives the message and acts on it.

Some commands (such as `arrowKey`) are system messages as well as commands. This means two things: a system event can generate the message (pressing an arrow key generates the `arrowKey` message), and HyperCard has a built-in response to the message (`arrowKey` takes you to another card).

Redefining Commands

You can write a message handler that redefines a built-in command. Redefining commands is especially useful for trapping menu commands you want to modify or that you want to prevent a user from issuing (for example, `on doMenu menuItem`).

Be wary, however: once a command—or any message—has been intercepted by a handler, it’s sent no further along the hierarchy, so your newly defined command replaces HyperCard’s built-in one. If, for example, you write a
CHAPTER 10

Commands

handler for the doMenu command, be sure to pass the message if you don’t want to prevent every instance of it from reaching HyperCard:

```hypercard
on doMenu menuItem
  if menuItem is "Delete Card" then
    answer "Are you sure?" with "Delete" or "Cancel"
    if It is not "Delete" then exit doMenu
  end if
  pass doMenu
end doMenu
```

If you inadvertently fail to pass doMenu, you may find yourself apparently unable to use any menu command, even to fix the doMenu handler. (In that case, from the Message box, execute the command edit script for the object containing the handler. If the Message box is hidden and blind typing is false, go to the last card of the Home stack and turn blind typing on.)

Syntax Description Notation

The syntax descriptions use the following typographic conventions. Words or phrases in this font are HyperTalk language elements or are those that you type to the computer literally, exactly as shown. Words in italics describe general elements, not specific names—you must substitute the actual instances. Brackets ([ ]) enclose optional elements that may be included if you need them. (Don’t type the brackets.) In some cases, optional elements change what the message does; in other cases they are helper words that have no effect except to make the message more readable.

It doesn’t matter whether you use uppercase or lowercase letters; names that are formed from two words are shown in lowercase letters with a capital in the middle (likeThis) merely to make them more readable.

The term yields indicates a specific kind of value, such as a number or a text string, that must result from evaluation of an expression when a restriction applies (for example, the expression and the destination in an add command must yield numbers). However, any HyperTalk value can be treated as a text string.
Some of the syntax statements and examples in this chapter use the soft return (~) character to continue long statements onto the next line. The soft return is used here because of the line length limitations of the page format used in this chapter. You should avoid using soft returns in your scripts so that the statements in your handlers are easier to read.

System 7 Commands

Some commands require system software version 7.0 or later. If you try to run a script with any of these commands in them while running an earlier version of system software, HyperCard sets the HyperTalk function the result to “Not supported by this version of the system.”

Command Descriptions

The rest of this chapter describes the commands supported by HyperCard 2.2.

Add

SYNTAX

\[
\text{add } \text{number to [chunk of] container}
\]

Number is an expression that yields a number. Chunk is an expression that yields a chunk of a container. Container is an expression that identifies a container, such as a field, the Message box, the selection, or a variable.

EXAMPLES

\[
\text{add 3 to It}
\]
\[
\text{add field "Amount" to field "Total"}
\]
CHAPTER 10

Commands

DESCRIPTION

The add command adds the value of number to the value of [chunk of] container and leaves the result in [chunk of] container. The value in the container when you begin must be a number; it is replaced with the new value.

SCRIPT

The following example handler sums numbers in a field, if each line of the field contains one number, and puts the result into the Message box. The name of the field is passed to the handler as a parameter.

on sumField whichField
    put 0 into total
    repeat with count = 1 to the number of lines¬
        in whichField
        add line count of whichField to total
    end repeat
    put total into msg
end sumField

Answer

SYNTAX

answer question [with reply [or reply2 [or reply3]]]
answer file promptText [of type fileType]
answer program promptText [of type processType]

Question and reply are expressions that yield text strings. If no reply is specified, HyperCard displays one button containing OK.

PromptText is an expression that yields a string of text that will appear in the dialog box as a prompt, telling the user what action to take. Use a string consisting of the space character, " ", if you do not want a prompt to appear. If you are using the answer program form of the answer command and you use the null string, " " , for the prompt text parameter, HyperCard displays the default prompt “Choose a program to link to:”. (See Figure 10-3.) PromptText can be up to 254 characters.
CHAPTER 10

Commands

Reply, reply2, and reply3 can be up to 254 characters each.

FileType is one of the following literal file types: stack, text, application, picture, paint, and painting. You can also specify the same file types with the following Macintosh four-letter file-type designators: STAK, TEXT, APPL, PICT, and PNTG.

ProcessType is a System 7–friendly process currently running on your machine or any others on the network.

EXAMPLES

answer "Which is the way the world ends?" with "Bang" or "Whimper"

answer myQuestion with myAnswer or field 7

answer file "Pick a text file:" of type text

answer file "" of type PICT

answer program "Pick a Zone, Macintosh, and Program."

DESCRIPTION

The answer command either displays a dialog box with a question and up to three buttons, each representing a different reply, or displays a standard file dialog box, with a list of either all the files of a certain type or a PPC Browser that contains a list of all the System 7–friendly processes currently running on your machine and any others on the network.

When you use the answer question form of the answer command, the last reply you specify correlates to the default button in the resulting dialog box. If no reply is specified, HyperCard displays one button containing OK. The dialog box stays on the screen until one of the buttons is clicked. Pressing Return or Enter has the same effect as clicking the default button.

When you use the answer file form of the answer command, HyperCard displays a standard file dialog box containing your prompt and a list of files. You can specify a file type using the fileType parameter. When your user selects a file from the file list, its name is placed in the local variable It.
CHAPTER 10

Commands

The answer program command displays a dialog box in which the user can choose from all the System 7–friendly programs, or processes, running on the user’s Macintosh computer or any other networked Macintosh computers.

When you select a program from the PPC Browser, its program path goes into the local variable It. If you click the Cancel button or press the Command-period keys, the variable It is empty and the result contains Cancel.

SCRIPT

The following example handler produces the dialog boxes in Figure 10-1 (the second one depends on which button you click in the first one):

```applescript
on chooseColor
    answer "Which color do you prefer?" with "Red" or "Blue" or "Yellow"
    if It is "Red" then answer "You picked Red."
    else if It is "Blue" then answer "You picked Blue."
    else if It is "Yellow" then answer "You picked Yellow."
end chooseColor
```

Figure 10-1  Answer command dialog boxes

Which color do you prefer?

Red  Blue  Yellow

You picked Blue.

OK
CHAPTER 10

Commands

The following example handler displays the standard file dialog box shown in Figure 10-2.

```on mouseUp
    answer file "Print what document?" of type text
    if It is not empty then
        put It into doc
        answer file "Use what application?" of type ¬
        application
        if It is not empty then print doc with It
        end if
    end if
end mouseUp```

![Figure 10-2](image)

Answer command display of the standard file dialog box

Execute the following line of code from either a script or the Message box to produce the PPC Browser shown in Figure 10-3.

```answer program ""```
There is no way for a script to reply to a dialog box by itself, so it’s important that a script meant to run unattended not use answer.

The text of the button clicked goes into the local variable It. If no reply is specified, HyperCard displays one button containing OK.

The question can be up to 14 lines or 254 characters.

If you use a container with more than one line for a reply, the middle line is displayed in the button. (Only the center portion shows if the line is too long to fit in the button.) However, all lines go into the local variable It when the button is clicked.

Unless you’re using container names, put the question and the replies inside quotation marks if they contain any spaces.

If you do not supply a file type for the answer file form, all file types are displayed in the dialog box. If you try to execute answer program under System 6, the result returns “Not supported by this version of the system”.

See also the ask command, later in this chapter.
CHAPTER 10

Commands

**ArrowKey**

**SYNTAX**

arrowKey direction

*Direction* is an expression that yields the name of one of the arrow keys: left, right, up, or down.

**EXAMPLES**

arrowKey left
arrowKey down

**DESCRIPTION**

The *arrowKey* command takes you to another card. The effects of the *arrowKey* command are shown in Table 10-1.

The *arrowKey* message, which invokes the *arrowKey* command if it reaches HyperCard, is normally generated by pressing any of the arrow keys on the keyboard. (Which arrow key you press determines the message’s parameter value.) You can also send *arrowKey* from the Message box or execute it as a line in a script.

<table>
<thead>
<tr>
<th>Parameter value</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td>Go to previous card in current stack</td>
</tr>
<tr>
<td>right</td>
<td>Go to next card in current stack</td>
</tr>
<tr>
<td>up</td>
<td>Go forward through recent cards</td>
</tr>
<tr>
<td>down</td>
<td>Go back through recent cards</td>
</tr>
</tbody>
</table>

Table 10-1  Effects of the *arrowKey* command
CHAPTER 10

Commands

Note
The textArrows property alters the effect of pressing the arrow keys (see “TextArrows” in Chapter 12), but it does not affect the arrowKey command.

See also the arrowKey message in Table 8-3.

SCRIPT

The following example handler works with extended keyboards. It makes function keys 9, 10, 11, and 12 send the arrowKey message with parameters of left, right, up, and down, respectively:

on functionKey whichKey -- map function keys to arrow keys
   if whichKey is 9 then arrowKey left
   else if whichKey is 10 then arrowKey right
   else if whichKey is 11 then arrowKey up
   else if whichKey is 12 then arrowKey down
   else pass functionKey
end functionKey

Ask

SYNTAX

ask question [with defaultAnswer]
ask password [clear] question [with defaultAnswer]
ask file promptText [with fileName]

Question and defaultAnswer are expressions that yield text strings. PromptText is an expression that yields a text string. FileName is an expression that yields a default filename to be displayed in the filename field of the dialog box.

Put the question and the default answer inside quotation marks if they contain any spaces or if they are telephone numbers containing a hyphen (see the example script), unless you’re using container names.
CHAPTER 10

Commands

EXAMPLES

ask "Who needs this kind of grief?" with "Not me."
ask field 1 with line 1 of field 2
ask password "Please enter your password:"
ask file "Save this file as:" with "MyTextFile"
ask file """"  

DESCRIPTION

The ask command displays a dialog box containing a question with a text box into which the user can type an answer. The optional defaultAnswer string specifies an answer that appears initially in the window, highlighted so it can be easily replaced. The dialog box appears with OK and Cancel buttons. The question and defaultAnswer can have up to 14 lines between them for all text to display properly.

The ask password form causes the answer to be encrypted as a number (which is placed into the local variable It). The encrypted answer can be stored in a field to be compared to a later answer to ask password if, for example, you want the user to be able to protect data he or she enters into the stack. Password protection built into a stack in this manner is separate from that set up by the Protect Stack command in the File menu. The text entered in the password dialog box is hidden from the user. Like the ask form, the ask password form's question and defaultAnswer can have up to 14 lines between them for all text to display properly.

The ask file form of the ask command displays a standard dialog box for saving files. You can use the optional fileName form of the command to place a default filename in the filename text entry field of the dialog box. The fileName string should be 23 characters or less in length to fit into the text entry field. The default filename is replaced by the user when the user starts typing. If you do not supply a default filename, the filename text entry field is empty when the dialog box is displayed.

The promptText string should be 7 lines or less to fit into the dialog box. If you do not want a prompt, use "".

Ask  175
CHAPTER 10

Commands

SCRIPT

The following example handler produces the dialog box in Figure 10-4:

on phone
  ask "Dial what number:" with "555-1212"
  if It is not empty then dial It
end phone

Figure 10-4  Ask command dialog box

NOTES

The text typed into the box (the answer to the question, the password, or the filename) goes into the local variable It, either when the OK button is clicked or when Return or Enter is pressed. If the Cancel button is clicked, the dialog box goes away, but the answer is not placed into It. The function the result returns “Cancel”.

If you use the ask password clear form of the ask password command, the password is not encrypted when it is typed in the dialog box.

Unless you’re using container names, put the question and the default answer inside quotation marks if they contain any spaces (or if, as in the example, they are telephone numbers containing a hyphen—to prevent HyperCard from doing subtraction).

See also the answer command, earlier in this chapter.
CHAPTER 10

Commands

Beep

SYNTAX

\texttt{beep [number]}

\textit{Number} is an expression that yields an integer.

EXAMPLES

\texttt{beep 5}
\texttt{beep line 3 of field 1}

DESCRIPTION

The \texttt{beep} command causes the Macintosh speaker to play the system beep sound \textit{number} times. If no \textit{number} is given, the speaker sounds the system beep once.

SCRIPT

The following example handler uses the \texttt{beep} command to alert the user that an \texttt{answer} dialog box, to which the user must reply, is being displayed:

\texttt{on openStack}
\hspace{1em} \texttt{beep 2}
\hspace{1em} \texttt{answer "Do you need instructions?" with "Yes" or "No"}
\hspace{1em} \texttt{if It is "Yes" then go to stack "Instructions"}
\texttt{end openStack}
CHAPTER 10

Commands

Choose

SYNTAX

choose toolName tool
choose tool toolNumber

ToolName is an expression that yields the name of any one of the tools from the HyperCard Tools palette (shown in Figure 10-5). ToolNumber is an expression that yields an integer from 1 to 18.

EXAMPLES

choose browse tool
choose tool 11
choose eraser tool

DESCRIPTION

The choose command changes the current tool as though you had chosen it from the Tools palette. Valid tool names and numbers are

browse (1) oval (14)
brush (7) pencil (6)
bucket (13) poly[gon] (18)
button (2) rect[angle] (11)
curve (15) reg[ular] poly[gon] (17)
eraser (8) round rect[angle] (12)
field (3) select (4)
lasso (5) spray [can] (10)
line (9) text (16)
CHAPTER 10
Commands

Figure 10-5  Tools palette

SCRIPT

The following example shows a typical use of the choose command in a handler:

```plaintext
on drawBox
  reset paint
  choose rectangle tool
  set lineSize to 2
  drag from 50,50 to 150,150
  choose browse tool
end drawBox
```

NOTES

You must have the user level set to Painting, Authoring, or Scripting to use the choose command, but the Tools palette need not be visible.

You can get the name of the current tool by using the function the tool in a handler or the Message box.

Setting user levels is described in the HyperCard Reference and in the userLevel global property description in Chapter 12, “Properties.” The tool function is described in Chapter 11, “Functions.”
CHAPTER 10

Commands

Click

SYNTAX

click at point [with key [key[,key2[,key3]]]]

Point is an expression yielding a point: two integers separated by a comma, representing horizontal and vertical pixel offsets (respectively) from the top-left corner of the card. Key, key2, and key3 are expressions that yield one of the following key names, separated by commas: shiftKey, optionKey, or commandKey (or cmdKey).

EXAMPLES

click at 100,100

click at the loc of button "Press me" with optionKey

DESCRIPTION

The click command causes the same actions as though you had clicked with the pointer at the specified point on the screen: the system messages mouseDown and mouseUp are sent to the objects under the pointer (but the visible pointer is not moved from its current location).

Using a with key form produces the same result as clicking the mouse button while holding down the specified key (or keys).

If point is within an unlocked field, the insertion point is set: if there is text at or past point, the insertion point is set at point; if there is text on the same line as point but point is beyond the end of text, the insertion point is set at the end of text on that line; if there is no text at point, the insertion point is set at the start of the line.

You can select a word by double-clicking it (that is, by executing the click command twice in succession at the location of the word). You can select any string of text by using click at the beginning, then using click with shiftKey at the end of the string.
CHAPTER 10

Commands

SCRIPT

The following example handler selects and displays a word from a locked field when you click the word (mouseUp is not sent to unlocked fields when you click them):

on mouseUp
    set lockText of me to false
    click at the clickLoc -- simulates double-click
    click at the clickLoc
    get the selection
    put It into the Message box
    set lockText of me to true
end mouseUp

NOTES

The pixel offset values of point are not restricted to the size of the screen but are misinterpreted if greater than 32767.

See also the drag command, later in this chapter.

Close

SYNTAX

close [docPathname [in|with]] appPathname

DocPathname is an expression yielding a text string that is a valid document name. AppPathname is an expression yielding a text string that is a valid application name or desk accessory.

You must provide the full pathname of the document if it cannot be found through the search paths. When running System 7, both parameters, docPathname and appPathname, can refer to Finder alias files.
EXAMPLES

close "HD:Applications:TeachText 7.1"
close "TestDoc" with "HD:MyApp 1.0"

DESCRIPTION

The close command closes an application, a document opened with another application, or a desk accessory, by sending one of the Apple event messages quit or clos to the other application. The form close [docPathname [in|with]] appPathname sends the clos Apple event, and the form close appPathname sends a quit Apple event.

The other application or desk accessory must be on the same Macintosh computer as HyperCard, and that Macintosh computer must be running System 7. Also, the other application must recognize the Apple event quit to close itself and the Apple event clos to close a document. (See also the system message appleEvent described in Chapter 8, “System Messages.”)

SCRIPT

The following button handler closes a specific application after the user’s intention is confirmed:

```plaintext
on mouseUp -- Button handler
    answer "Are you sure you want to close"¬
    &"OtherHyperCard?" with "No" or "Yes"
    if It is "Yes"
    then close "MacHD:OtherHyperCard"
end mouseUp
```
NOTE

The following error messages go into the container the result of the source program when the close command fails:

<table>
<thead>
<tr>
<th>Condition</th>
<th>the result contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Closing an application that’s not running</td>
<td>No such application</td>
</tr>
<tr>
<td>Target program didn’t handle event</td>
<td>Not handled by target program</td>
</tr>
<tr>
<td>Target program returned error number in reply, or AESend returned some</td>
<td>Got error &lt;errorNum&gt; when sending</td>
</tr>
<tr>
<td>other error</td>
<td>Apple event</td>
</tr>
<tr>
<td>Target program returned error string in reply</td>
<td>&lt;errorString&gt;</td>
</tr>
<tr>
<td>Target program timed out</td>
<td>Timeout</td>
</tr>
</tbody>
</table>

Close File

SYNTAX

close file fileName

FileName is an expression that yields a text string that is a valid filename.

EXAMPLES

close file myData

close file "myDisk:myFolder:myFile"
CHAPTER 10

Commands

DESCRIPTION

The close file command closes a disk file previously opened with the open file command to import or export ASCII text. The expression filename must yield a valid Macintosh filename, including pathname if required.

SCRIPT

The following example handler reads any size text file into a global variable named temp:

```script
on importText
  global temp
  put "MyFilename" into filename
  open file filename
  read from file filename until end
  put It into temp
  close file filename
end importText
```

NOTES

If the specified file is not open, an error is generated. The error is stored in the HyperCard function the result. Use the close file command to close files explicitly after you use them. HyperCard automatically closes all open files when an exit to HyperCard statement is executed, when you press Command-period, or when you quit HyperCard.

You must provide the full pathname of the file if it's not at the same directory level as HyperCard. (See “Identifying a Stack” in Chapter 5 for an explanation of pathnames.)

See also the open file, read, and write commands, later in this chapter, and the result function in Chapter 11, “Functions.”
CHAPTER 10

Commands

Close Printing

SYNTAX

close printing

DESCRIPTION

The close printing command ends a print job previously begun with the open printing command or the open report printing command. Nothing is actually printed until the close printing command is executed.

SCRIPT

The following example handler executes a printing job. It prints a specified number of cards, beginning on a specified card:

on printRange low,high
  push card
  open printing
  go to card low
  print (high-low) + 1 cards
  close printing
  pop card
end printRange

NOTE

See also the open printing, open report printing, print, and print card commands, later in this chapter.
CHAPTER 10

Commands

Close Window

SYNTAX

close window window

Window is an expression that identifies a stack window or picture window.

DESCRIPTION

The close window command closes the specified stack. The close window command works on stack windows only if more than one stack is open and the stack is active (or frontmost). The close window command also works on picture windows, external windows, and stacks that are hidden. When the specified window is closed, a close system message is also sent by HyperCard.

EXAMPLE

close window "Planets"

NOTE

You cannot use the close window command on HyperCard’s built-in palettes; use the hide command instead.

See also the go command in this chapter and the close system message described in Chapter 8, “System Messages.”
CHAPTER 10

Commands

CommandKeyDown

SYNTAX

commandKeyDown char

Char is an expression yielding a character (spaces count as characters).

EXAMPLE

commandKeyDown "H" -- go Home
commandKeyDown "y" -- paste
commandKeyDown "5" -- doMyCommand

DESCRIPTION

The commandKeyDown command passes a character, char, which represents a key pressed in combination with the Command key. HyperCard has various built-in responses to the commandKeyDown command, depending on the character passed with it.

You can use the commandKeyDown command to take advantage of the built-in meanings for Command-key combinations in HyperCard. For example, commandKeyDown "P" is the same as pressing the Command key and "P" key together or selecting Print Card from the File menu.

You also can override HyperTalk Command-key combinations. If you name a message handler the same as a built-in command, your name overrides the built-in one if yours is anywhere along the message-passing hierarchy between the object sending the message and HyperCard.

In the case of keys other than those built into HyperCard, nothing happens when a given Command-key combination is pressed or a commandKeyDown command is executed unless you have a handler for that Command-key combination in the script of the current card or in a script somewhere in the message-passing hierarchy between the current card and HyperCard.
CHAPTER 10

Commands

SCRIPT

The following example handler in a stack script in the current message-passing hierarchy opens the Navigator palette at the coordinates 20,30 when it intercepts the `commandKeyDown` system message and the parameter "Y". The handler works either if the Command-Y key combination is pressed or if you type the command `commandKeyDown "Y"` in the Message box.

```script
on commandKeyDown whichKey
  if whichKey = "Y" then
    palette "navigator","20,30"
    exit commandKeyDown
  end if
  pass commandKeyDown
end commandKeyDown
```

NOTES

CommandKeyDown is also a system message sent to the current card when the Command key is pressed in combination with another key. See Table 8-3 for more information.

See also the `keyDown` and `controlKey` commands in this chapter, and the `commandKey` function in Chapter 11, “Functions.”

ControlKey

SYNTAX

controlKey  keyNumber

`KeyNumber` is an expression that yields an integer between 1 and 127. The values 97 through 126 do not correspond to any key.

EXAMPLE

controlKey 26
CHAPTER 10

Commands

DESCRIPTION

The `controlKey` command passes a numeric value, `keyNumber`, which represents a key pressed in combination with the Control key.

NOTES

`ControlKey` is also a system message sent to the current card when the Control key is pressed in combination with another key. The acceptable values for the `keyNumber` parameter are shown in Table 10-2.

There are no built-in meanings for Control-key combinations in HyperCard. Nothing happens when a given Control-key combination is pressed or a `controlKey` command is executed unless you have a handler for that Control-key combination in the script of the current card or in a script somewhere in the message-passing hierarchy between the current card and HyperCard.

See also the `controlKey` system message in Table 8-3.

Table 10-2  ControlKey message parameter values

<table>
<thead>
<tr>
<th>Parameter value</th>
<th>Keys</th>
<th>Parameter value</th>
<th>Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a, Home</td>
<td>12</td>
<td>l, Page Down</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>13</td>
<td>m, Return</td>
</tr>
<tr>
<td>3</td>
<td>c, Enter</td>
<td>14</td>
<td>n</td>
</tr>
<tr>
<td>4</td>
<td>d, End</td>
<td>15</td>
<td>o</td>
</tr>
<tr>
<td>5</td>
<td>e, Help</td>
<td>16</td>
<td>p, all function keys</td>
</tr>
<tr>
<td>6</td>
<td>f</td>
<td>17</td>
<td>q</td>
</tr>
<tr>
<td>7</td>
<td>g</td>
<td>18</td>
<td>r</td>
</tr>
<tr>
<td>8</td>
<td>h, Delete</td>
<td>19</td>
<td>s</td>
</tr>
<tr>
<td>9</td>
<td>i, Tab</td>
<td>20</td>
<td>t</td>
</tr>
<tr>
<td>10</td>
<td>j</td>
<td>21</td>
<td>u</td>
</tr>
<tr>
<td>11</td>
<td>k, Page Up</td>
<td>22</td>
<td>v</td>
</tr>
</tbody>
</table>

*continued*
The following example handler in a stack script in the current message-passing hierarchy prints the current card when it intercepts the `controlKey` system message and the parameter 16. The handler works either if the Control-P key is pressed.
CHAPTER 10

Commands

combination is pressed or if you type the command controlKey 16 in the Message box.

on controlKey whichKey
  if whichKey = 16 then
    doMenu "Print Card"
    exit controlKey
  end if
  pass controlKey
end controlKey

Convert

SYNTAX

convert [chunk of] container|literal [[from format] [and format]] to format [and format]

Container is an expression that identifies a container, and format is an expression that yields a time or date format. The optional [and format] specification is used when both a date and time are included. Valid formats and their meanings are as follows:

- abbreviated date: The date in text form with abbreviated day of week:
  - abbrev date: Tue, October 17, 1989.
- abbr date: The abbreviated date:
- abbreviated time: Same form as short: 5:15 PM.
- abbrev time: The abbreviated time:
- abbr time: The same as abbrev time:
- dateItems: A comma-separated list of numbers representing (in order) year, month, day, hour, minute, second, and day of week.
- long date: The date in text form: Tuesday, October 17, 1989.
CHAPTER 10

Commands

long time The time in colon-separated form including seconds: 5:15:15 PM.
seconds Seconds since midnight, January 1, 1904.
short date The date in slash-separated numeric form: 10/17/89. The date separators may be different in countries other than the United States.
short time The time in colon-separated form without seconds: 5:15 PM.

EXAMPLES

This line of code puts the content of the first line of the second card field into the internationally invariant format dateItems:

convert line 1 of card field 2 from date to dateItems

If line 1 of card field 2 contains the date April 9, 1993, then HyperCard puts this into card field 2: 1993,4,9,0,0,0,6. Other examples using the convert command are:

convert card field "Date and Time" from date and time to ¬ dateItems

convert timeVariable to seconds

convert myVar from seconds to long date

convert line 1 of second cd field to long date and ¬ short time

DESCRIPTION

The convert command gets a date or time and converts it from a particular format, if you choose to specify one, to a particular format. This command works with any date format supported by an installed script.

192 Convert
The first form of the `convert` command gets the date or time from a container and places the converted date or time in that container. When you use a literal as input to the `convert` command, the resulting date or time is stored in the HyperTalk variable `It`.

If you know precisely what format the input is in, you can choose to specify a format to convert from. If the input can’t be parsed as specified, HyperTalk sets the result to "Invalid date".

Note that if you try to convert an invalid date, for example, a date that has the wrong number for the day, like Wednesday, May 6, 1993, when it should be Thursday, May 6, 1993, HyperTalk sets the result to "Invalid date".

**IMPORTANT**
A script that needs to perform calculations on dates and times should first convert to one of HyperTalk’s invariant formats for dates and times—`seconds` and `dateItems`. This avoids problems that may occur when someone tries to run your stack on a machine where the format of the date has been localized (to Swedish format, for instance) or customized (to display military time, for instance) by changing resources in the System file. To make sure your scripts and stacks are localizable, you should also be careful to save dates and times in either `seconds` or `dateItems` format. These formats are the only ones that the commands `convert` and `sort` are guaranteed to recognize on any system.

**SCRIPT**
Here’s an example that shows how to get tomorrow’s date in the short format (this works no matter what format the date is set to by system resources):

```script
on mouseUp
   get the date -- the current date in short format
   convert it to dateItems -- year, month, day, hour, -- minute, second, day of week
   add 1 to item 3 of it   -- make it tomorrow
   convert it to short date
end mouseUp
```

Convert
CHAPTER 10

Commands

The following example handler counts the seconds elapsed while a command in the Message box executes:

```plaintext
on mouseUp
    put the long time into startTime
    convert startTime to seconds
    if msg is not empty then do msg
    put the long time into endTime
    convert endTime to seconds
    answer "That took" && endTime && "seconds."
end mouseUp
```

Create Menu

**SYNTAX**

```
create menu menuName
```

*MenuName* is an expression that yields the name of a new menu to be added to the menu bar.

**EXAMPLES**

```
create menu "Home"
create menu "Vacation locations"
```

**DESCRIPTION**

You use the `create` command to add a new menu to the HyperCard menu bar. After the new menu is created, it remains in the menu bar until it is deleted with the `delete` command, or you use the `reset menubar` command, or you quit HyperCard. If you want a set of menus to remain in the menu bar while a certain stack is open, create your menus in that stack's `openStack` system message handler. You can delete or disable your menus when you close your stack or open another stack by using the `delete` or `disable` command in `closeStack` or `suspendStack` system message handlers. See Chapter 8, “System Messages,” for more information about system messages.
CHAPTER 10

Commands

You can put menu items into the new menu with the put command, described later in this chapter. You can also specify a menu message to be sent when a menu item is chosen.

NOTES

See also the checkMark, commandChar, enabled, markChar, menuMessage, name, and textStyle properties in Chapter 12, “Properties.”

See also the delete, disable, enable, and put commands in this chapter.

Create Stack

SYNTAX

create stack stackName [with background] [in a new window]

StackName is an expression that yields the name you want to assign to the new stack. Background is an expression that yields the descriptor of a background you want to use in the new stack. The background must be one in the current stack.

EXAMPLES

create stack "Ghosts" with this background
create stack "Mystery" with bkgnd 3

DESCRIPTION

The create command creates a new stack with the specified name and optionally with a specified background of the current stack.

If you do not specify a background, the stack is created with a blank background. If you use the form create stack stackName in a new window, the stack is created, appears in another window, and becomes the current stack. If you do not use the form in a new window, the current stack is closed.
CHAPTER 10

Commands

NOTES

The `create stack` command no longer allows `stackName` to begin with the period character.

To create a stack with a background in a stack other than the current stack, you must go to the stack with the background you want to use before using the `create stack` command.

If the `create stack` command generates an error, it is stored in the HyperCard function `the result`.

The new stack is created with the same card size as the current stack. You can change the card size by resetting the `rectangle` property of the `card window` or by clicking the Resize button in the Stack Info dialog box, and then dragging the lower-right corner of the representation of the card to the size you want.

See also the `result` function in Chapter 11, “Functions,” and the `rectangle` property in Chapter 12, “Properties.”

---

Debug Checkpoint

SYNTAX

```
display checkpoint
```

EXAMPLE

```
display checkpoint
```

DESCRIPTION

The `display checkpoint` command sets a checkpoint in a HyperTalk handler. When HyperCard encounters this message in a handler, it enters the debugger: it pauses execution of the handler and opens the script editor window, putting a box around the line where the checkpoint is set.

Once you are in the debugger, you can step or trace through the remaining lines of the script. You can step by pressing Command-S or choosing Step from the Debugger menu. You can step into by pressing Command-I or choosing
CHAPTER 10

Commands

Step Into from the Debugger menu. You can trace by choosing Trace from the Debugger menu. You can trace into by pressing Command-T or choosing Trace Into from the Debugger menu. With each step, the next line of HyperTalk in the current handler is selected and executed.

NOTES

In combination, checkpoints and stepping can be used to examine, diagnose, and modify the behavior of sophisticated HyperTalk handlers.

See also Chapter 3, “The Scripting Environment,” for more information about the debugger environment.

Delete

SYNTAX

delete chunk of container
delete menu
delete menuItem of menu
delete part

Chunk is a chunk expression referring to some text in a specified container, and container specifies the container. Menu is an expression that yields a menu descriptor. MenuItem is an expression that yields a menu item descriptor. Part is an expression yielding a button or field descriptor.

EXAMPLES

delete line 1 of field 1
delete char 1 to 5 of line 4 of field "Charlie" of second card
delete menuItem "Paths..." of menu "Home"
delete menu "Windows"
delete second menuItem of menu 6
delete button 1
CHAPTER 10
Commands

delete last button
delete card field "Temp Data"
delete field id 1234

DESCRIPTION
The delete command removes
- text from a container in the current stack
- a menu item
- a menu
- a part (button or field) from the current card

Note that this command lets you delete menus from the standard HyperCard menu bar, including the Tools, Patterns, Font, and Apple menus; but you can’t delete menu items within those menus. If you delete those menus, the menu items in those menus can still be activated with the doMenu command or with their Command-key equivalents.

SCRIPT
The following example handler finds and deletes a name from a list with one name per line:

on zapName
  put "Maller" & return & "Calhoun" & return & "Winkler" ~
  into list
  ask "Delete which name from the list?" with empty
  repeat with count = the number of lines in list ~
    down to 1
      if It is in line count of list then ~
        delete line count of list
    end if
  end repeat
end zapName
NOTES

Using the `delete` command to delete a chunk is not the same as using `put empty into` with the same chunk of text specified. For example, if you delete a line in a field with a statement like

```
delete line 4 of field 7
```

you delete the return character as well as the text; what was previously the fifth line becomes the fourth. The following statement leaves the return character in line 4:

```
put empty into line 4 of field 7
```

Even if you delete all of the text in a field, the field remains defined on the card or background, unlike selecting the field and choosing Cut Field or Clear Field from the Edit menu.

When you delete text in a field on a card other than the current one, the current card does not change. If you delete the text in a background field that has the `sharedText` property set to `true`, the text in that background field is deleted on every card with that background field.

Because deleting parts causes a `deleteButton` or `deleteField` message to be sent, you can delete only parts on the current card. In other words, `delete button 1 of next card` doesn’t work; however, you do not get an error message when you try to do this.

In the case of the HyperCard standard menu items, the `doMenu` command works even when the item is deleted with the `delete` command. For example, if the File menu is deleted and the following handler is executed, HyperCard exits to the Finder:

```plaintext
on mouseDown
    delete menuItem "Quit HyperCard" from menu "File"
    doMenu "Quit HyperCard"
end mouseDown
```

Command-key equivalents, however, do not work for menu items that have been deleted, with one exception: Command-Q still works after the Quit HyperCard menu item is deleted.
Chapter 5, “Referring to Objects, Menus, and Windows,” describes how to designate menus and menu items. Chunk expressions are described in Chapter 7, “Expressions.” See also the create menu, disable, enable, and put commands in this chapter.

Dial

SYNTAX

dial number [with modem [modemCommands]]

Number is an expression that yields a string with numbers in it, and modemCommands are commands for your modem.

EXAMPLES

dial steve -- if steve is a variable containing a number
dial "415-555-1212"
dial "493-996-1010" with modem "ATS0=0S7=1DT"
dial "493-973-6000" with modem

DESCRIPTION

The dial command, without the with modem option, generates the touch-tone sounds for the digits in number through the Macintosh speaker. Holding the telephone handset up to the speaker works on some telephones; for others you need a device that feeds the Macintosh audio output to the telephone.

If you use the with modem option, the dial command sets up telephone calls using the Apple Modem 300/1200, the Apple Personal Modem, or any Hayes-compatible modem attached to the Macintosh serial port. The modemCommands parameters are those described in the manual for your modem. Their default value is "ATS0=0DT".

If number yields a string with numbers including a hyphen (as in 555-1212), enclose it within quotation marks to prevent HyperCard from doing subtraction with the hyphen before passing the number to the dial command (which ignores characters other than numbers). Similarly, enclose the modemCommands parameter within quotation marks.
NOTES

You can press Command-period to exit the dial command during the ten-second wait imposed when dialing with a modem.

A one-second delay occurs between opening the modem port and first using it.

---

Disable

SYNTAX

disable menu
disable menuItem of menu
disable button

Menu is an expression that yields a menu descriptor. MenuItem is an expression that yields a menu item descriptor. Button is an expression that yields a button descriptor.

EXAMPLES

disable background button 5
disable menu "Home"
disable menu 5
disable menuItem "Get Back" of menu "Direction"
disable the fourth menuItem of sixth menu

DESCRIPTION

The disable command disables a menu, menu item, or button. When any of these objects is disabled, it is gray and inactive. The disable command also sets the enabled property to false. You should be aware that when you create a menu item, it is automatically enabled unless you use the disable command to change it.

If you try to use the disable command on a menu item, menu, or button that does not exist, HyperCard displays an error dialog box with the text, "No such menu" (or menu item or button) unless the lockErrorDialogs property is set to true.
Excerpt from a document discussing the use of commands in a software context.

**NOTE**
See also the `commandChar`, `enabled`, `markChar`, and `menuMessage` and button properties in Chapter 12, “Properties,” and the `create menu`, `enable`, and `put` commands in this chapter.

**Divide**

**SYNTAX**

```
divide [chunk of] container by number
```

*Chunk* is an expression that yields a chunk expression. *Container* is a container that holds a numeric value, and *number* is any expression that yields a numeric value.

**EXAMPLES**

```
divide field "total" by 3
divide fahrenheit by celsius -- if fahrenheit and -- celsius are variables
divide line 3 of field 4 by 10
```

**DESCRIPTION**

The `divide` command divides the value of `[chunk of] container` by the value of `number` and puts the result into `[chunk of] container`. 
The following example handler figures the percentage represented by a fraction of two numbers specified as parameters:

```script
on percent var1, var2
    divide var1 by var2
    put trunc(var1 * 100) & "%"
end percent
```

The value previously in the container must be a number; it is replaced with the new value.

Division by 0 puts the result INF into container. (INF is the SANE value for infinity.) Division is carried out to a precision of up to 19 decimal places. The value for the amount of precision is set with the numberFormat property.

See also the numberFormat global property in Chapter 12, “Properties,” and the discussion of numbers in Chapter 6, “Values.”

**DoMenu**

**SYNTAX**

```script
doMenu itemName [ , menuName][without dialog] ¬
    [with modifierKey [ , modifierKey ]]
```

*itemName* is an expression that yields the name of a menu command. *MenuName* is an expression that yields the name of a menu. *ModifierKey* is one or more comma-delimited combinations of optionKey, commandKey, and shiftKey.

**EXAMPLES**

```script
doMenu "open stack..."
doMenu "Copy Card"
doMenu "Open Stack..." with shiftKey
```
DESCRIPTION

The `doMenu` command performs the menu command specified by `itemName` as though you had chosen the command directly from the appropriate HyperCard menu. In conjunction with the `doMenu` command, you can choose to suppress a dialog box by using the `without dialog` option.

You may also use an optional modifier key to apply to your menu item choice. For instance, if someone chooses the Open Stack menu item while holding down the Shift key, the Open Stack dialog box appears with the New Window checkbox already checked. This HyperTalk command line does the same thing:

```hyper```
doMenu "Open Stack..." with shiftKey```
```

**Note**

If you write your own `doMenu` handler to intercept menu commands, you can examine `param(6)` to find out if a modifier key parameter was specified.

There are several caveats to be aware of when using or intercepting the `doMenu` command:

- Both the specified menu item and the menu in which it resides must be available at the current user level (as described in the *HyperCard Reference Guide*).

- If there are periods following the menu item, you must include them in `menuItem` (you can’t use the ellipsis character in their place). For example, "Open Stack..." is a HyperCard menu item with three periods.

- Some menu commands change with conditions (for example, Paste Card can change to Paste Button, depending on the contents of the Clipboard).

- If you write a handler to intercept the `doMenu` system message that is sent to the current card when a menu item is selected, be sure to pass the message after examining the new menu item. (See the example.) Otherwise, you may find yourself apparently unable to use any menu command, fix the `doMenu` handler, or quit HyperCard. (In that case, from the Message box, execute the command `edit script` for the object containing the handler. If the Message box is hidden and blind typing is `false`, go to the last card of the Home stack and turn blind typing on.)
CHAPTER 10

Commands

SCRIPT

The following example handler checks for a doMenu message with the Quit HyperCard menu item and puts up a dialog box when the Quit HyperCard menu item is selected:

```pascal
on doMenu menuChoice
    if menuChoice is "Quit HyperCard"
        then
            answer "Are you sure you want to Quit" ¬
            with "OK" or "Cancel"
            if it is "Cancel"
                then
                    exit doMenu
                else
                    pass doMenu
            end if
        else
            pass doMenu
        end if
    end if
end doMenu
```

Drag

SYNTAX

drag from point1 to point2 [with key [,key2[,key3]]]

Point1 and point2 are expressions, each of which yields a point: two integers separated by a comma, representing horizontal and vertical pixel offsets (respectively) from the top left of the Macintosh screen. Key, key2, and key3 are one of the following key names, separated by commas: shiftKey, optionKey, or commandKey (or cmdKey).
CHAPTER 10

Commands

EXAMPLES

drag from 100,100 to 200,200
drag from the loc of button 1 to the mouseLoc with ¬
commandKey,shiftKey

DESCRIPTION

The drag command performs the same action as though you had dragged manually, except that in order to select text in a field using the drag command, you must use with shiftKey. In all other cases, using the with key form produces the same result as dragging while holding down the specified key.

SCRIPT

The following example handler draws random-sized ovals filled with random patterns on a new card:

on mouseUp
  doMenu "New Card" -- so we don't draw on current card
  choose oval tool
  set filled to true
  repeat until the mouseClick
    set pattern to random(30)
    drag from random(319),random(199) to ¬
      random(319),random(199)
  end repeat
  choose browse tool
  doMenu "Delete Card" -- get rid of the card we just made
  go previous card -- take us back where we started from
end mouseUp

NOTES

You can use drag with any tool selected, but it has no effect with some Paint tools.

The location of the actual pointer doesn’t change from where it was before the command was issued.
CHAPTER 10

Commands

See also the click command earlier in this chapter, and the dragSpeed property in Chapter 12, “Properties.”

Edit Script

SYNTAX

edit [the] script of object

Object is an expression that yields a descriptor of an object: a stack, card, background, field, or button.

EXAMPLES

edit script of button 1
edit script of this stack

DESCRIPTION

The edit script command opens the script of the specified object with the HyperCard script editor as though you had clicked the Script button in the object’s Info dialog box.

SCRIPT

The following example handler enables you to edit the script of any button or field merely by positioning the pointer over it and pressing the Option key:

on mouseWithin
    if the optionKey is down then edit script of the target
end mouseWithin

NOTE

Refer to Chapter 3, “The Scripting Environment,” for an explanation of how the script editor works.
Enable

SYNTAX

enable button
enable menu
enable menuItem of menu

Confirmation is an expression that yields a card or background button descriptor. Menu is an expression that yields a menu descriptor. MenuItem is an expression that yields a menu item descriptor.

EXAMPLES

enable menu "Home"
enable menu 5
enable menuItem "Get Back" of menu "Direction"
enable the fourth menuItem of sixth menu
enable background button id 3
enable btn "PanAm"

DESCRIPTION

The enable command makes the specified menu, menu item, or button active by setting its enabled property to true and making its text and outline appear solid rather than dimmed.

If you enable a menu, menu item, or button that does not exist, HyperCard displays an error dialog box with the text "No such menu" (or menu item or button) unless the lockErrorDialogs property is set to true.

NOTES

See also the commandChar, enabled, markChar, and menuMessage and button properties in Chapter 12, "Properties," and the create menu, disable, and put commands in this chapter.

Command keys do not work on menu items that have been deleted or disabled.
CHAPTER 10

Commands

EnterInField

SYNTAX

```
enterInField
```

DESCRIPTION

The `enterInField` command closes a field that is open for text editing.

NOTES

The `enterInField` system message, which invokes the `enterInField` command if it reaches HyperCard, is normally sent by pressing the Enter key on the keyboard, but you can also execute it as a line in a script.

Closing a field with `enterInField` sends the `closeField` system message; if no text was changed, the `exitField` system message is sent.

See also the `enterKey` and `exitField` system messages in Chapter 8, “System Messages.”

EnterKey

SYNTAX

```
enterKey
```

DESCRIPTION

The `enterKey` command sends a statement typed into the Message box to the current card or, if a field is open for text editing, closes the field.
CHAPTER 10

Commands

NOTES

The enterKey system message, which invokes the enterKey command if it reaches HyperCard, is normally sent by pressing the Enter key on the keyboard, but you can also execute it as a line in a script.

Closing a field with enterKey sends the closeField system message; if no text was changed, the exitField system message is sent.

See also the enterKey and exitField system messages in Chapter 8, “System Messages.”

Export Paint

SYNTAX

export paint to file fileName

FileName is an expression that yields any valid Macintosh filename.

EXAMPLE

export paint to file "TreeFrogs"

DESCRIPTION

The export paint command creates a Macintosh paint file containing the image of the current card and saves it with the specified filename. If you are working in the background, only the graphics, buttons, and fields visible in the background are exported. If you are not working in the background, export paint exports both the card and background graphics plus all the visible card and background buttons and fields.

NOTES

If an error is generated while using the export paint command, the error is stored in the HyperCard function the result.
CHAPTER 10

Commands

The export paint command only works when a Paint tool is chosen. If you use export paint while using the browse, button, or field tool, an error message is put into the result.

See also the import paint command in this chapter and the result function in Chapter 11.

Find

SYNTAX

\begin{verbatim}
find [international] text [in field] [of marked cards]
fmd text [in field] [of marked cards]
find string [international] text [in field] [of marked cards]
find whole [international] text [in field] [of marked cards]
find word [international] text [in field] [of marked cards]
\end{verbatim}

Text is an expression that yields a series of one or more text strings separated by spaces, and field is an expression that yields a card field or background field descriptor.

EXAMPLES

\begin{verbatim}
find "money" in field "Charity"
find chars "Wild" in field 1
find word msg in second field
find word international "æble" in field 5
\end{verbatim}

DESCRIPTION

The find command searches through all the card and background fields (visible or not) in the stack for the text strings yielded by text. The search begins on the current card and continues through the last card, the first card, and on to the card previous to the current card. The of marked cards option searches only marked cards.
You can use the international option with the find command to enable searching that recognizes international characters like æ and ø as unique from a and o. This is important in languages such as Danish, where such characters are distinguished.

**Note**

If you write a handler to override the find command, you can examine param(1) to determine whether the international option was specified.

Choosing Find from the Go menu (or pressing Command-F) puts the find command in the Message box with the text insertion point after it between double quotation marks.

**SCRIPT**

The following example handler queries the user for search criteria, then executes the find command:

```on doMenu var
    global findString
    if var is "Find..." then
        ask "Find what string:" with findString
        if It is not empty then
            put It into findString
            answer "Match" && findString && "how:" ¬
                with "Chars" or "Word" or "All"
            if It is "Chars" then find chars findString
            else if It is "Word" then find word findString
            else find findString
        end if
    else pass doMenu
end doMenu
```
The `find` command executes faster if you use as many three-character combinations as possible in the search string. That is, three characters are faster than one, six are faster than three, nine are faster than six, and so on.

The `find` form finds the match only at the beginnings of words. The `find chars` form finds the match anywhere within words. The `find word` form matches only complete words.

The `find whole` form (also invoked by pressing Command-Shift-F) lets you search for a specific word or phrase, including spaces. For HyperCard to find a match, all the characters must be in the same field, and they must be in the same order as they appear in the string derived from `text`.

When you use `find` without `whole`, HyperCard finds a card that contains every word in the string derived from `text`, but the words can appear in different order or in different fields. That is, with `find whole`, interword spaces are part of the search string; without `whole` the spaces delimit separate search strings. With every form of `find`, you can limit the search to a specific background field.

The following example finds a card with a field that has the phrase *Apple Computer* in it; it won’t find *Apple Computers* or *This apple is a computer*. (The `find` command without `whole` would find a match in all three cases.)

```
find whole "Apple Computer"
```

Find `whole` won’t find partial-word matches, and it pays no attention to case or diacritical marks: *apple Computeer* and *aPple cOmpute* are seen as the same. (If the international option is specified, however, diacritical marks are recognized.)

The `find string` form lets you search for a contiguous string of characters, including spaces, regardless of word boundaries. (Find `whole` searches for characters at the beginnings of words.) For HyperCard to find a match, all the characters must be in the same field, and they must be in the same order as in the string derived from `text`. For strings without spaces, `find string` works the same as `find chars`. 
In the following example, HyperCard finds the string in *Apple computers* but not in *computers, not apples*. (The `find` command without `string` would not find a match in either case.)

```plaintext
find string "ple Computer"
```

If the match is on a different card, it becomes the current card; if a match isn’t found, the current card doesn’t change. If you enter the `find` command from the Message box and a match isn’t found, HyperCard sounds a beep. If it finds a match, HyperCard puts a box around the word containing the found string if the field containing the string is visible. If a match is found in a hidden field, the field’s card becomes the current card, but the field remains hidden.

As the `find` command evaluates the expression passed to it, it places the resulting values internally between quotation marks as a single parameter string. The following examples show text expressions on the left and the resulting parameter string on the right:

```plaintext
find "my" && "word"  find "my word"
find "my" & "word"  find "myword"
find a & b & c  find "xyz" -- if a = "x",
-- b = "y", c = "z"
find a && b && c  find "x y z"
```

If more than one search string (separated from each other by spaces) is included in the parameter string, all of them must be on a single card or its background for a successful search. However, they can be in any order on the card, and only the first is shown with a box around it.

The text in shared background fields and in backgrounds, cards, and fields with the `dontSearch` property set to `true` is ignored by the `find` command.

Press Command-F to display the parameter string from the most recently executed `find` command in the Message box.

An unsuccessful search sets HyperTalk’s function `the result` to `not found`. After a successful search, `the result` is empty. (See the `result` function in Chapter 11, “Functions.”)

For more information about retrieving text, see the `foundChunk`, `foundField`, `foundLine`, and `foundText` functions in Chapter 11, “Functions.”
CHAPTER 10

Commands

FunctionKey

SYNTAX

functionKey keyNumber

_keyNumber_ is an expression that yields an integer between 1 and 15.

EXAMPLES

functionKey 1
functionKey 15

DESCRIPTION

The functionKey command has built-in Undo, Cut, Copy, and Paste functions for _keyNumber_ values 1 through 4, respectively. Any other value of _keyNumber_ has no built-in effect.

SCRIPT

The following example handler uses the functionKey command to implement the message undo as a command:

on undo
  functionKey 1 -- preprogrammed as undo in HyperCard
end undo

NOTES

The functionKey message, which invokes the functionKey command if it reaches HyperCard, is normally generated by pressing one of the 15 function keys on the Apple Extended Keyboard. But you can also send it from the Message box or execute it as a line in a script.

You can program function keys 5 through 15, or reprogram keys 1 through 4, by writing an on functionKey handler in the script of any object in the hierarchy between the current card and HyperCard. The following
CHAPTER 10

Commands

functionKey handler opens the print report dialog box when function key 9 is pressed.

on functionKey whichKey
    if whichKey = 9 then
        doMenu "Print Report..."
        exit functionKey
    end if
    pass functionKey
end functionKey

See also the functionKey system message in Chapter 8, “System Messages.”

Get

SYNTAX

get expression

Expression yields any value.

EXAMPLES

get the long name of field 1
get the location of button "newButton"
get 2+3 -- puts 5 into It
get the date

DESCRIPTION

The get command puts the value of any expression into the local variable It. That is, get expression is the same as put expression into It.
CHAPTER 10

Commands

SCRIPT

The following example handler saves the current user level, sets the user level to 5, then restores the saved level:

on doMything
  get userLevel -- get the current userLevel
  put It into savedLevel -- save userLevel before
    -- changing it
  set userLevel to 5 -- set userLevel for my
    -- button or script
    -- (put my script here)
  -- restore userLevel when leaving
  set userLevel to savedLevel
end doMything

Go

SYNTAX

go [to] [stack] stackName [in a new window] [without dialog]
go [to] background [of [stack] stackName [in a new window]]
  [without dialog]
go [to] card [of background] [of [stack] stackName]
  [in a new window]] [without dialog]
go back
ogo forth
ogo [to] ordinal
ogo [to] position

StackName is an expression that yields a valid stack name. Card is an expression that yields a valid card descriptor. Background is an expression that yields a valid background descriptor. Ordinal is an expression that yields an ordinal constant. Position is an expression that yields a special object descriptor.
CHAPTER 10

Commands

EXAMPLES

```go
go card 23
go to stack "ArtIdeas"
go bkgnd field 1 -- if bkgnd field 1 contains a stack name
go "home"
go mid card of stack "clip art"
go next
```
```go
go to first card of second background of "home"
go card 2 of stack "VacationSpots" in a new window
```
```go
go stack "VacationSpots" in a new window without dialog
```
```go
go card 4 of stack "VacationSpots" without dialog
```
```go
go "hd:bigFolder:innerFolder:myStack" -- full pathname
```

DESCRIPTION

The `go` command takes you to the specified destination. If you name a stack without specifying a card, you go to the first card in the specified stack. If you don’t name a stack, you go to the specified card in the current stack. If you go to a background, you go to the next card with that background (not the first card). If the current card has the specified background, you won’t move. Go forth and go back move you forward and backward among the recent cards. You can specify a visual effect to be used on opening the card by issuing the `visual effect` command before you use the `go` command.

If the destination is in a stack other than the current one and you use the `in a new window` form of the `go` command, the destination stack is opened in addition to any existing stacks. If you do not specify the `in a new window` form, the current stack is closed before the specified stack is opened. If only one stack is open and you do not specify the `in a new window` form of the `go` command, the current stack closes and the specified stack appears without a close box.

If you use the `without dialog` form of the `go` command and the destination can’t be found, you do not get a standard dialog box for opening files. Instead, the result is set to "No such stack" or "No such card". If the destination is a marked card within a stack that has no marked cards, you won’t move.
CHAPTER 10

Commands

SCRIPT

The following example handler queries the user for a destination, then executes a go command with a visual effect:

```script
on mouseUp
    ask "Where to?" with "This card"
    if It is empty then put "this card" into It
    put It into goWhere
    visual effect fade to black
    go to goWhere
end mouseUp
```

Help

SYNTAX

help

DESCRIPTION

The help command takes you to the first card of the stack named HyperCard Help.

NOTE

See also the help system message in Chapter 8.
CHAPTER 10

Commands

Hide

SYNTAX

hide background picture
hide card picture
hide groups
hide menuBar
hide object
hide picture of background
hide picture of card
hide titlebar
hide window stackName
hide window windowName

Object yields one of the following:

a valid button descriptor in the current stack
a valid field descriptor in the current stack
message [box] or message [window] or window "message"
pattern window or window "patterns" (the Patterns palette)
tool window or window "tools" (the Tools palette)
window "navigator" (the Navigator palette)
message watcher or window "message watcher"
variable watcher or window "variable watcher"
card window

Card yields the descriptor of a card in the current stack. Background yields the descriptor of a background in the current stack. WindowName is the name of a window created with the picture command. StackName is the name of an open stack.
EXAMPLES

hide message
hide bkgnb button "goHome"
hide field id 1
hide window "Planets"
hide card window

DESCRIPTION

The hide command removes the specified object from view. Its effect is the same as setting the visible property of the specified object to false or clicking a window’s close box (except for external windows and stack windows).

The hide picture form of the hide command removes from view the graphic bitmap on the card or background, and the show picture form of the show command displays it.

The hide groups form of the hide command removes the gray 2-pixel underline made visible below group style text by the show groups form of the show command. The hide groups command affects all group style text in all fields globally.

SCRIPT

The following example handler hides a field or button when the user puts the pointer over the button or field:

on mouseWithin
  hide the target
end mouseWithin

NOTES

Message can be abbreviated msg. Background can be abbreviated bkgnb. Button can be abbreviated btn. Card can be abbreviated cd.

Hide menuBar is also sent as a system message when the menu bar is hidden.
CHAPTER 10

Commands

If the screen is locked, hide menuBar has no effect. See the lock and unlock commands later in this chapter, and the lockScreen property in Chapter 12, “Properties.”

The hide command does not affect the location property of an object or window. You can’t use the with the object name. For example, the following statement results in the error message "Can’t understand arguments of command hide":

hide the window "navigator"

Hidden fields aren’t in the tab order. (They are skipped when you move the text insertion cursor from one visible field to the next by pressing the Tab key.) The find command does search through them, however, and you can put values into them and put their values elsewhere.

Hidden card and background pictures are not displayed when a Browse, Button, or Field tool is chosen, but if you attempt to use a Paint tool manually, a dialog box appears asking if you want to make the picture visible; clicking OK displays the picture. (You can draw on hidden pictures from a script.) Whether or not you are working in the background determines whether your actions pertain to the card or background picture.

If there is another card window behind the current stack’s card window and you hide the current stack’s card window, the card window behind it becomes the current card window.

If you use either the hide card window or hide window stackName form of the hide command to hide the current card or specified stack, the cards in that stack won’t be visible again until you set the card window or stack window’s visible property to true, use the show card window or show window stackName command, or close the stack and reopen it.

You can move hidden windows without changing their visible state with the location property, which is described in Chapter 12, “Properties.”

See also the picture and show commands, later in this chapter, and the visible property in Chapter 12, “Properties.”
CHAPTER 10

Commands

Import Paint

SYNTAX

import paint from file fileName

FileName is an expression that yields a valid Macintosh filename.

DESCRIPTION

The import paint command reads in the specified paint file and makes it the current selection. The import paint command allows you to put digitized images and other pictures created with Macintosh paint programs onto backgrounds or cards.

You can use the export paint and import paint commands together to enhance graphics created with HyperCard. For example, you could export a card image to a file and open it in your favorite paint application. You could then paste in your own pictures and make changes to the file containing the HyperCard card art with your paint program. Once you complete the art work, import the finished paint file to the original card or new card or background.

NOTES

If an error is generated while using the import paint command, the error is stored in the HyperCard function the result.

The import paint command only works when a Paint tool is chosen. If you use import paint while using the Browse, Button, or Field tools, an error message is put into the result.

Imported pictures are clipped to the size of the current card. The maximum area of an imported paint file that can be displayed is 576 pixels by 720 pixels.

See also the export paint command in this chapter and the result function in Chapter 11, “Functions.”
CHAPTER 10

Commands

KeyDown

SYNTAX

`keyDown char`

*Char* is an expression yielding a character (spaces count as characters).

EXAMPLE

`KeyDown "h"

DESCRIPTION

The `KeyDown` command passes a character, *char*, which represents any character on the keyboard. The `KeyDown` command causes HyperCard to enter the character passed with the command at the insertion point in a field, if one is open for text editing, or, otherwise, in the Message box.

NOTES

`KeyDown` is also a system message sent to the current card when the user presses a character key. When the `KeyDown` message reaches HyperCard, it invokes the `KeyDown` command.

See also the `commandKeyDown` and `controlKey` commands in this chapter.
CHAPTER 10

Commands

Lock

SYNTAX

lock screen|error dialogs|messages|recent

EXAMPLES

lock screen
lock error dialogs
lock messages
lock recent

DESCRIPTION

The lock command can be used for four different unrelated purposes. Using the lock command, you can

- prevent HyperCard from updating the screen by setting the lockScreen global property to true
- prevent HyperCard from displaying error dialogs in response to an error in an executing script by setting the lockErrorDialogs property to true
- prevent HyperCard from sending automatic open, close, suspend, and resume system messages by setting the lockMessages property to true
- prevent HyperCard from recording miniature representations of each card to the Recent card by setting the lockRecent global property to true

NOTE

See also the unlock command in this chapter and the lockErrorDialogs, lockRecent, lockMessages, and lockScreen properties in Chapter 12, “Properties.”
MARK

SYNTAX

mark card
mark cards where condition
mark all cards
mark cards by finding [international] text [in field]
mark cards by finding chars [international] text [in field]
mark cards by finding string [international] text [in field]
mark cards by finding whole [international] text [in field]
mark cards by finding word [international] text [in field]

Card is an expression that yields a card descriptor. Condition is an expression that evaluates to true or false. Text is any text. Field is an expression that yields a field descriptor.

EXAMPLES

mark [the] next card
mark cards where "We be shaking" is in field 2
mark all cards
mark cards by finding whole chicken in field 1
mark cards where the short name of this bkgnd is "Clients"
mark cards where the number of buttons > 0

DESCRIPTION

The mark command sets the marked property for the specified card or cards to true. Cards can also be marked with the Card Marked option in the Card Info dialog box. By default, cards are unmarked.

The by finding form of the mark command uses chars, word, whole, and string to define the search criteria the same way the find command does. See the description of the find command for information about how to use these forms.
You can use the international option with the find command to enable searching that recognizes international characters like æ and ø as unique from a and o. This is important in languages such as Danish, where such characters are distinguished.

The mark command can be used with the unmark command in searches where you want to find and mark cards containing particular information while excluding other unnecessary information.

For example, say you want to mark and print all cards with information about “San Francisco” but not “earthquake.” You might have a script that uses the following statements:

unmark all cards
mark cards where "San Francisco" is in field 1
unmark cards where "earthquake" is in field 1

You could then show, print, or display the number of cards containing that combination of search criteria.

show marked cards
print marked cards
put the number of marked cards into word 3 of field 2

NOTES

See also the marked property in Chapter 12, “Properties,” and the unmark command, later in this chapter.

You can’t mark cards in a read-only stack, even if the userModify property is set to true. See the userModify property in Chapter 12, “Properties.”
CHAPTER 10

Commands

Multiply

SYNTAX

\[
\text{multiply} \ [\text{chunk of}] \ \text{container} \ \text{by} \ \text{number}
\]

Chunk is an expression that yields a chunk expression. Container is a container holding a numeric value, and \text{number} is an expression that yields a numeric value.

EXAMPLES

\begin{itemize}
  \item multiply \text{Subtotal} \ \text{by} \ \text{Tax}
  \item multiply \text{field 1} \ \text{by} \ \text{field 3}
  \item multiply \text{line 3 of card field 2} \ \text{by} \ 25
  \item multiply \text{It} \ \text{by} \ 2 -- \ \text{puts result into It,}
    \quad -- \ \text{replacing the old value}
\end{itemize}

DESCRIPTION

The \text{multiply} command multiplies the value in \[\text{chunk of}] \ \text{container} \ \text{by} \ \text{value of} \ \text{number} \ \text{and} \ \text{puts the result in} \ [\text{chunk of}] \ \text{container}.

SCRIPT

The following example handler adds 6 percent to the value of items in a list:

\[
\text{on} \ \text{taxMe}
\quad \text{put} \ "12.45,15.00,150.00,76.95,10.00,14.95" \ \text{into} \ \text{taxables}
\quad \text{repeat with} \ \text{count} = 1 \ \text{to} \ \text{the number of items in} \ \text{taxables}
\quad \quad \text{multiply item count of} \ \text{taxables} \ \text{by} \ 1.06
\quad \text{end repeat} -- \ \text{the new values are stored in} \ \text{taxables}
\text{end taxMe}
\]
NOTES

The value previously in the container must be a number; it is replaced with the new value.

The result is calculated to a precision of up to 19 decimal places and, if put into a field or the Message box, is displayed according to the numberFormat global property.

See also the numberFormat global property in Chapter 12, “Properties,” and the discussion of numbers in Chapter 6, “Values.”

Open

SYNTAX

open [fileName with] application

Application is the name of any application, and fileName is the name of any document on your Macintosh computer. Either one can be an expression that yields such a name.

EXAMPLES

open "Apps:BigApp"
open "Letter" with "Apps:MacWrite"
open Field 3
open FavoriteApp

DESCRIPTION

The open command launches the named application. A specific document may be opened with its own creator or a compatible application by using the open filename with application form of the open command.
The following example handler queries the user for a document and application before executing the open command:

```
on mouseUp
  ask "Open what document?" with empty
  if It is not empty then
    put It into doc
    ask "Use what application?" with empty
    if It is not empty then open doc with It
  end if
end mouseUp
```

The open command can also bring HyperCard itself to the front—when it is running under system software version 7.0 or later. When you need to make HyperCard the frontmost process, use command lines like these in your own script:

```
get the long name of HyperCard -- get pathname of HyperCard
open it
```

This is useful if, for instance, you want to transfer data from another application to HyperCard via the Clipboard. Neither MultiFinder nor the System 7 Process Manager allows access to the Clipboard when an application is in the background. So, if you want to automate a copy and paste between some other application and HyperCard, you must also automate switching between applications.

**NOTES**

If the document or application you specify isn’t at the top level of the file hierarchy (the “disk” level), then the path to it must be specified on the appropriate Search Path card of the Home stack. Alternatively, you can specify the full pathname with the open command:

```
open "MyHardDisk:Apps:Apples"
```
Note that handlers that override the `open` command may use the function `the params` to determine the parameters of the original command.

If HyperCard can’t find the requested document or application, it displays the dialog box for locating files to the user. Error messages go into the container the result of the source program when the `open` command fails.

When running single Finder in System 6 and you quit the application, you go to the card you were on in HyperCard when you executed the `open` command. However, any global variables you had previously declared are now gone, and any portions of handlers that remained unfinished when you executed the `open` command do not finish.

---

**Open File**

**SYNTAX**

```
open file fileName
```

*fileName* is the name of any file accessible to your Macintosh computer, or an expression that yields such a name.

**EXAMPLES**

```
open file "textOnly"
open file field 1
```

**DESCRIPTION**

The `open file` command opens the named file for reading or writing. Usually, the file is an ASCII text file opened in preparation for importing or exporting text. If the specified file doesn’t exist, HyperCard creates it.
CHAPTER 10

Commands

SCRIPT

The following example handler opens a given file, reads a line of data from it, then closes the file:

```
on openCard
    open file "myUpdate"
    read from file "myUpdate" until return
    put It into card field 1
    close file "myUpdate"
end openCard
```

NOTES

If the specified file is already open, an error is generated. The error is stored in the HyperCard function `the result`. The `result` is empty if the command is successful. Use the `close file` command to close files explicitly after you use them. HyperCard automatically closes all open files when an `exit to HyperCard` statement is executed, when you press Command-period, or when you quit HyperCard.

You must provide the full pathname of the file if it's not at the same directory level as HyperCard. (See “Identifying a Stack” in Chapter 5, “Referring to Objects, Menus, and Windows,” for an explanation of pathnames.)

See also the `read`, `write`, and `close file` commands in this chapter and the `result` function in Chapter 11, “Functions.”

Open Printing

SYNTAX

```
open printing [with dialog]
```
CHAPTER 10

Commands

DESCRIPTION

The `open printing` command starts a print job to be ended later by a `close printing` command.

The settings specified in the Print Stack dialog box are used unless `with dialog` is specified, in which case the dialog box is displayed and new settings can be chosen.

SCRIPT

The following example handler prints a selection of cards:

```plaintext
on printSelection
  put "1,3,8,15,21" into myCards
  open printing with dialog
  repeat with count = 1 to the number of items in myCards
    go card item count of myCards
    print this card
  end repeat
  close printing -- print the cards
end printSelection
```

NOTES

Printing cards with `open printing` is similar to printing with the Print Stack command in the File menu, except that Print Stack prints all cards in the stack, while `open printing` prints only the ones you specify with the `print card` command, described later in this chapter.

You must use some form of the `print` command and then the `close printing` command to print and then close a print job begun with `open printing`. Don’t use the `print fileName with application` command while a print job is active.

See also the `close printing`, `open report printing`, `print`, and `print card` commands in this chapter.
Open Report Printing

SYNTAX

open report printing [with template \textit{templateName}]
open report printing [with dialog]

\textit{TemplateName} is an expression that yields the name of a previously defined print template.

EXAMPLES

open report printing
open report printing with template "fieldsOnly"
open report printing with dialog

DESCRIPTION

The \texttt{open report printing} command prepares a report printing job. The print report job is sent to the printer and closed with a \texttt{close printing} command.

SCRIPT

The following script sets the report template and prints the marked cards in the current stack:

\begin{verbatim}
on PrintLabels
  open report printing with template mailing labels
  -- choose the "mailing labels" template
  print marked cards -- specify which cards to print
  close printing -- generate the report & print
end PrintLabels
\end{verbatim}
CHAPTER 10

Commands

NOTES

Settings previously specified in the Print Report dialog box are used unless the with dialog form is used, in which case the Print Report dialog box is displayed and new settings can be chosen.

If you do not specify a print report template with the with template templateName form or do not use the with dialog form, the settings from the last print report template accessed in the Print Report dialog box are used for the current print job. If no print report template has been used previously for printing and you neither specify a print report template nor use the with dialog form, nothing is printed.

Printing cards with open report printing is similar to printing with the Print Report command in the File menu, except that you can specify the report template to use in the script without using the Print Report dialog box.

You must use some form of the print command and the close printing command to send the job to the printer and then close a print job begun with open report printing. If you start another print job without closing the previous one, HyperCard notifies you that it will close the previous print job before starting the new one. Don’t use the print fileName with application command while a print job is active.

When you choose Cancel after opening the Print Report dialog box with open report printing, the function the result is set to Cancel.

See also the close printing, open printing, print, and print card commands in this chapter.

Palette

SYNTAX

palette paletteName[, point]

PaletteName is an expression that yields the name of the palette you wish to invoke. Point is an expression that yields two comma-separated integers that represent the horizontal and vertical coordinates at which the palette should appear. Point is the offset from the upper-left corner of the current card window to the upper-left corner of the palette minus the title bar. Point needs to be in quotation marks or passed in a container.
CHAPTER 10

Commands

EXAMPLES

palette "Navigator", "50,100"
palette "GeneralPalette", "150,80"

DESCRIPTION

The `palette` command displays the specified custom XCMD palette or the HyperCard palette called Navigator. If you do not specify a `point` parameter, the palette appears at the default location of 10,20 inside the current card coordinate system if it’s the first time it’s been displayed. After the first time, it appears at its last location if you don’t specify a `point` parameter. `Point` is reset to the default location at the beginning of each HyperCard session.

The `palette` command and associated palette properties have no effect on the Tools palette or Patterns palette.

NOTES

If you specify a palette that is already visible, the `palette` command moves the palette to the location specified by the `point` parameter.

You can set the `location` property of a palette window after the palette is displayed with the `palette` command:

```
set the loc of window "Navigator" to "65,80"
```

You can close a palette that is displayed by using the `close` command:

```
close window "Navigator"
```

After a palette has been displayed with the `palette` command, you can hide and show a palette by using the `hide` and `show` commands or setting the `visible` property. The `hide` and `show` commands have no effect if the palette hasn’t been displayed yet by the `palette` command. Here are some examples of statements that hide or show palettes:

```
hide window "Navigator"
set the visible of window "Navigator" to true
show window "GeneralPalette"
```
There is a group of properties that apply only to HyperCard XCMD palettes. The palette properties are `buttonCount`, `commands`, `hilitedButton`, and `properties`.

You can use the `buttonCount` property to determine the total number of buttons in a palette.

`Commands` returns a return-delimited list of the commands or messages associated with the palette's buttons. The commands are listed according to the number of the button they are associated with, that is, first the command associated with button number 1, then the command associated with button number 2, and so forth.

The `HilitedButton` property determines or sets the number of the currently highlighted button of the specified palette. Setting `HilitedButton` does not cause the message associated with the button to be sent. For example, the statement

```
set the hilitedButton of window "Navigator" to 3
```

does not send the `doMenu "Help"` message associated with button 3 of the Navigator palette.

In the case of action palettes, such as Navigator, the value of the `hilitedButton` property is always -1.

The `properties` palette property returns a comma-separated list of the names of the properties that apply to the specified palette.

HyperCard palettes consist of two resources: one of type `PLTE` and one of type `PICT`. The `PLTE` resource contains the functional code for the palette, and the `PICT` resource supplies the palette image. The two resources must have the same name and resource ID number. Because the palette image is a `PICT` resource, palettes can be in color.

See also the `hide` and `show` commands in this chapter.
CHAPTER 10

Commands

Picture

SYNTAX

picture [fileName,sourceType,windowStyle,visible,depth,floatingLayer]

FileName is the name of a file or resource of type 'PICT' or 'PNTG' on your Macintosh computer or an expression that yields such a name.

SourceType is resource, file, or clipboard. (The default source type is file.)

WindowStyle is the style of window in which the picture is displayed. The window styles are plain, rect, zoom, roundRect, dialog, document, shadow, and windoid. (The default window type is zoom.)

Visible is a Boolean value: true for visible, false for invisible. This parameter allows you to create an invisible window and set its properties before displaying it with a show command. See the description for more information about the picture window properties.

Depth is the bit depth of the offscreen buffer that the picture command creates. Bit-depth values between 0 and 32 inclusive are supported. The picture command allocates an offscreen buffer of the bit depth you specify (rounded down to a power of 2) or the bit depth of the picture file, whichever is smaller. This allows you to display picture files with a deeper bit depth in less memory, but at the cost of lower resolution. If the value is 0, the picture command doesn’t create an offscreen buffer; instead, the 'PICT' file is drawn directly into the window. A value of 0 for depth makes scrolling and zooming slower and prevents dithering from working. If available system memory is extremely low and the bit depth is set to 0, the file is spooled to the display, but picture properties do not work. It does, however, allow you to display pictures in less memory and animated 'PICT' files and format-1 'PICT' files with color.

FloatingLayer is a true or false value that signals whether the new picture is in the floating layer, whose elements always appear above elements in the document layer (where cards and scripts reside), or in the document layer, which is always behind miniwindows and palettes. If you don’t specify this parameter, HyperCard selects a layer appropriate to the window style—the floating layer for windoid, shadow, and rect styles; the document layer for plain, zoom, roundRect, dialog, and document styles.
EXAMPLES

picture "Clowns", resource, plain, false, 0
picture "MyPICT", file, rect
picture "Picnic", clipboard, roundRect

DESCRIPTION

The picture command displays color or gray-scale pictures in an external window. The pictures can come from the Clipboard, from PICT or MacPaint files, or from 'PICT' resources in the current stack or any stack in the message-passing hierarchy.

The picture command works best if HyperCard's application memory size in the Get Info dialog box is set to 2 megabytes or more. If the picture cannot be displayed because of insufficient memory, an error describing the condition is returned in the result function.

There is a set of properties that apply to windows created with the picture command. They are rect, globalRect, globalLoc, scroll, zoom, scale, and dithering. The properties are in addition to the standard HyperTalk properties location and visible, which also apply to windows created with the picture command.

The rect and globalRect properties are set just like the rectangle property for other HyperCard windows. The rect property applies to the rectangle of the window created with the picture command in coordinates local to the current card window. The globalRect property applies to the rectangle of the window created with the picture command in global screen coordinates. See the rectangle property in Chapter 12, "Properties."

The rect or globalRect property is specified as four comma-separated integers representing the bounding window rectangle. The first two integers represent the top-left corner position of the window on the screen, and the second two integers represent the bottom-right corner of the window. Four literals can also be used to set the rect and globalRect properties. They are cardScreen (or card), largestScreen (or largest), deepestScreen (or deepest), and mainScreen (or main). These literals display the picture window centered on the same screen as the card window, on the screen with the largest area, on the screen with the greatest bit depth, and the main screen, respectively.
CHAPTER 10

Commands

Here are examples that set the rect and globalRect properties:

set the rect of window "Clowns" to "120,225,300,480"
set the globalRect of window "Clowns" to "largest"

The loc (also called location) property applies to the location of the window created with the picture command in coordinates local to the current card window. The globalLoc property applies to the location of the window created with the picture command in global screen coordinates. For more information on the loc (location) and globalLoc properties, see Chapter 12, “Properties.”

The loc or globalLoc property specifies a point in the screen’s coordinate system where the top-left corner of the window is to be displayed. The point is specified as two integers that represent the horizontal and vertical offsets, respectively. Four literals can also be used to set the loc and globalLoc properties. They are cardScreen (or card), largestScreen (or largest), deepestScreen (or deepest), and mainScreen (or main). These literals display the picture window centered on the same screen as the card window, on the screen with the largest area, on the screen with the greatest bit depth, and the main screen, respectively. Here are examples that set the loc and globalLoc properties:

set the loc of window "Flowers" to "65,100"
set the globalLoc of window "Clowns" to "card"

The scroll property for windows created with the picture command is like the scroll property for cards in card windows. See the scroll property in Chapter 12, “Properties.” Scroll is specified as two comma-separated integers representing the horizontal and vertical offsets, in the picture’s coordinate system, to be displayed at the top-left corner of the window. Here is an example that sets the scroll property for a window:

set the scroll of window "Water" to "45,60"

The dithering property is a Boolean value: true for dithering, false for no dithering. (The default for the dithering property is false.) Here is an example that sets the dithering property for a window:

set the dithering of window "Garlic" to true
The `scale` property scales a picture in a window created with the `picture` command. The value for `scale` is an integer between -5 and 5, inclusive. Negative integers scale down the picture, and positive integers scale up the picture. Scaling is done using 2 raised to the `scale` power. For example, if the `scale` is -2, the picture is scaled to 25 percent. The default value for `scale` is 0. Here is an example that sets the `scale` property for a picture in a window:

```plaintext
set the scale of window "Summer fun" to 4
```

The `zoom` property applies to windows created with a window style that supports a zoom box. It zooms a window in or out. The possible values for `zoom` are `in` and `out`. Here is an example that sets the `zoom` property for a window:

```plaintext
set the zoom of window "Alligators" to out
```

**NOTES**

If you do not provide a valid name for the `fileName` parameter, a standard dialog box for opening files is displayed from which you can choose a 'PICT' or 'PNTG' file. The one exception to this is when you specify `clipboard` for the `sourceType` parameter. In that case any name can be used. If you cancel the standard file dialog box, HyperCard sets the result to `cancel`.

If the `sourceType` parameter is a resource and the filename specified in the `fileName` parameter can’t be found, the `picture` command converts the `fileName` parameter to a number and looks for a 'PICT' resource with the specified number as its ID.

If you do not set the `rect` or `globalRect` property, HyperCard displays the picture in a window that is the same size, or as close to the same size as possible, as the original picture.

If you do not set the `scroll` property, the picture is displayed with its 0,0 coordinate at the top-left corner of the external window.

The new 32-bit Color QuickDraw is fully supported. You can display 16-, 24-, and 32-bit images with the `picture` command. The `dithering` property is ignored if 32-bit Color QuickDraw isn’t installed in the System Folder.
The rect and shadow window types are created behind all the HyperCard windows. This allows you to create picture windows that appear to pop up above the card window.

You can close windows created with the picture command by clicking the close box or with the close command as follows:

```lisp
close window "fileName"
```

fileName is the filename specified in the fileName parameter for the picture command. It is also the name displayed in the title bar of the window.

If an error occurs when creating a window, the picture command sets the result to an error message that begins with "Couldn't display picture".

If the picture file is displayed successfully, the result is empty.

When you click a window created with the picture command, two system messages are sent. When the mouse button is down, a mouseDownInPicture message is sent. When the mouse button is released, a mouseUpInPicture message is sent. Each message is sent with two parameters: the name of the window and the point within the picture's default (not scaled) local coordinates at which the mouse button was clicked. You can place handlers for these messages anywhere in the message-passing hierarchy or in the stack script of the stack that invokes the picture command. You can use these messages to simulate button actions, card flipping, or anything else that you would use mouseUp and mouseDown system messages for.

The following handler checks for a mouseDownInPicture message sent by a window created by the picture command and puts its name and the location where the mouse button was clicked into the Message box:

```lisp
on mouseDownInPicture wName, cLoc
    put "You clicked window" &&quote& wName &quote&& ¬
        "at location" && cLoc
end mouseDownInPicture
```

See also the close command in this chapter, and the system messages mouseUpInWindow, mouseDownInWindow, openPicture, and closePicture in Chapter 8, “System Messages.”
CHAPTER 10

Commands

Play

SYNTAX

play sound [tempo tempo] [notes]
play stop

Sound is an expression that yields the name of a digitized sound (boing, flute, and harpsichord are included with HyperCard). Tempo is an expression that yields the speed at which the sound plays, and notes is an expression that yields a list of one or more notes representing the pitch at which the sound plays and the duration of the notes. Digitized sounds are of the Macintosh sound resource formats 1 and 2, which are described in Inside Macintosh: Sound.

EXAMPLES

play "boing" tempo 200 "c4e c dq c f eh" -- Happy Birthday
play "harpsichord" "ch d e f g a b c5w"

DESCRIPTION

The play command makes the Macintosh computer play notes through its speaker (or through the audio jack if it's plugged in). You can write a song by specifying a series of notes after the play command. The play stop form stops the current sound immediately; otherwise, it plays until it's done and stops by itself. In most cases, HyperCard continues to execute handlers and perform other actions while a sound plays. In the event of a low-memory situation, such as when playing a large sound while a large Home stack or several other stacks are in use and HyperCard is set to the default memory allocations, HyperCard may suspend other actions until the sound is finished playing. Increasing HyperCard's memory allocation should alleviate this problem.

Digitized sounds are of the Macintosh sound resource formats 1 and 2, which are described in Inside Macintosh: Sound. The resources must exist in a stack in the hierarchy or in HyperCard application. If the sound can't be found or can't be loaded into memory, the result gets set to "Couldn't load sound". If the sound isn't played because the volume is set to 0 (in the Sound control
CHAPTER 10

Commands

panel), HyperCard is running in the background, or an XCMD is using HyperCard’s sound channel, the result gets "Sound is off".

SCRIPT

The following example handler goes to each card in a stack and synchronizes playing the specified notes with each card change:

```
on tour
    repeat the number of cards
        play "harpsichord" tempo 200 "ce4 fe ae c5q ae4 cq5"
        go next card
        wait until the sound is "done"
    end repeat
end tour
```

NOTES

Tempo is a number specifying the speed at which the group of notes is played (100 is a medium tempo; higher numbers are faster). The sound and tempo are specified once for each play command.

The notes are specified in the following form:

```
noteName accidental octave duration
```

NoteName is the name of the note played (A through G); accidental is # or b, specifying sharp or flat, respectively; octave is a number specifying the pitch of the scale (4 is the “middle C” scale); and duration specifies the relative time value of the note played:

- w whole note
- h half note
- q quarter note
- e eighth note
- s 16th note
- t 32nd note
- x 64th note

244 Play
You can use a period (.) or numeral 3 following duration to specify a dotted or triplet note, respectively.

Octave and duration may be changed for each note played; if they are not changed, subsequent notes are in the same octave and have the same duration as the previous note.

The 254-character limit on note strings that existed in earlier versions of HyperCard no longer applies.

See also the sound function in Chapter 11, “Functions.”

Pop Card

SYNTAX

pop card [preposition [chunk of] container]

Preposition is into, before, or after; chunk is a chunk expression, and container is an expression that identifies a container.

EXAMPLE

pop card into field 3 of card WhereIbeen

DESCRIPTION

The pop card command retrieves the identification (full ID and stack pathname) of a card previously saved with the push card command. If you don’t provide a destination for the identification, you go directly to the card whose address is popped.
CHAPTER 10

Commands

SCRIPT

The following example handler pushes whatever card you’re on, goes to another stack, gets the value of a field property, then returns to the original card:

```script
on getTheFont
  global myStack,theFont
  push card
  go myStack
  put textFont of field 1 into theFont
  pop card -- goes to the card formerly pushed
end getTheFont
```

NOTES

After the card has been popped, its identification is removed from the memory stack—it can’t be popped again. If a container is given, however, the card’s identification is put into the container, but you don’t go anywhere.

See also the push command, later in this chapter.

Print

SYNTAX

```script
print fileName with application
print field
print expression
print button
```

FileName is an expression that yields the name of any document on your Macintosh computer, and application is an expression that yields the name of the application to which it belongs (or with which it is compatible). Field is an expression that yields any field descriptor. Expression is an arbitrary expression or container.
EXAMPLES

print "memo" with "MacWrite"
print field 1 with field "Program"
print "HD:MY DOCS:letter" with "HD:Applications:MacWrite"
print bkgnd field 1

DESCRIPTION

The print command prints the specified file, field, or expression.

The print fileName with application form of the print command suspends HyperCard, launches the named application, opens the named document, prints the document, then resumes running HyperCard. The specified application must support printing.

The print field form of the print command prints the specified field using the current font, size, style, and line height of that field.

The print expression form prints any arbitrary HyperTalk expression. Expressions are printed using global print properties, such as printTextFont.

SCRIPT

The following example handler queries the user for the name of a document to print and an application with which to print it:

on mouseUp
  ask "Print what document?" with empty
  if It is not empty then
    put It into doc
    ask "Use what application?" with empty
    if It is not empty then print doc with It
  end if
end mouseUp
CHAPTER 10

Commands

NOTES

If the document or application you specify isn’t at the top level of the file hierarchy (the “disk” level), then the path to it must be specified on the appropriate Search Path card of the Home stack. Alternatively, you can specify the full pathname with the print command.

Error messages go into the container the result of the source program when the print command fails.

Don’t use the print command while a print job started with the open printing command is active.

See also the print card command described in this chapter.

Print Card

SYNTAX

\[
\text{print card \{from point1 to point2\}} \\
\text{print marked cards} \\
\text{print all cards} \\
\text{print number cards}
\]

Card is an expression that yields a card descriptor or the word card, which refers to the current card.

Point1 is an expression that yields two comma-separated numbers representing the upper-left corner of a rectangular region you want to print on the specified card. Point2 is an expression that yields two comma-separated numbers representing the lower-right corner of a rectangular region you want to print on the specified card. Number is an expression that yields an integer or the word all.

EXAMPLES

\[
\text{print card from 0,123 to 345,512} \\
\text{print last card} \\
\text{print card id 3011} \\
\text{print all cards}
\]
print marked cards
print howMany cards -- howMany contains an integer
print card

DESCRIPTION

The print card command makes HyperCard print the specified card. It differs from the Print Card command (Command-P) in the File menu in that the File menu command prints at full size, while print card prints at the size specified in the Print Stack dialog box. The print number cards form prints the number of cards specified by number, beginning with the current card. The print marked cards form prints a group of marked cards. The print card form makes HyperCard go to the specified card, print it, and return to the current card.

SCRIPT

The following example handler queries the user for a number of cards to print whenever Print Card is chosen from the File menu:

on doMenu var
   if var is "print card" then
      ask "Print how many cards?" with one
      open printing
      if It is a number then print It cards
      close printing
   else pass doMenu -- make sure other menu choices
      -- continue to work
end doMenu

NOTES

You don’t need to use the open printing command before using the print card command. If nothing is printing, the print card command prints the specified card or cards immediately; if an open printing command is in effect, no cards are printed until a page is full (depending on how many cards per page are specified in the printing dialog box) or the close printing command is given.
Chapter 5, “Referring to Objects, Menus, and Windows,” defines card descriptors. See also the marked property in Chapter 12, “Properties,” and the close printing, mark, open printing, and open report printing commands described earlier in this chapter.

Push

SYNTAX

```bash
push card
push card [of stack stackName]
push background [of stack stackName]
push stack
```

*Card* is an expression that yields the descriptor of any card. *StackName* is the name of an open stack.

EXAMPLES

```bash
push recent card
push first card
push card
```

DESCRIPTION

The push command saves the identification of the specified card or stack in a LIFO (last-in, first-out) memory stack (an area of memory, not a HyperCard stack).

SCRIPT

The following example handler saves the current card, goes to a random card, then returns to the original card:

```bash
on nonSense
    push card -- save current card
    go any card
    pop card -- restore current card
end nonSense
```
CHAPTER 10

Commands

NOTES

The card identification can be retrieved later with the `pop card` command (usually so that you can go directly back to the pushed card). The card identification that’s saved is the full card ID and stack pathname. HyperCard holds the IDs of up to 20 cards.

Card descriptors are described in Chapter 5, “Referring to Objects, Menus, and Windows.”

See also the `pop card` command, earlier in this chapter.

Put

SYNTAX

```
put expression [preposition [chunk of] container]
put itemName preposition [menuItem of] menu
    [with menuMsg message]
```

Expression is an expression that yields a text string or number; preposition is into, before, or after; chunk is a chunk expression; and container is an expression that identifies a container.

ItemName is an expression that yields a single menu item name or a list of comma-separated or return-delimited names to be added to the specified menu. MenuItem is an expression that yields the word menuItem followed by either the name or number (integer or ordinal number) of a standard HyperCard menu item or a user-defined menu item in the specified menu. Menu is an expression that yields the word menu followed by either the name or number (integer or ordinal number) of a standard HyperCard menu or a user-defined menu in the menu bar. Message is an expression that yields a single message or a list of comma-separated or return-delimited menu messages to be sent when a specified menu item is chosen. The menu messages in the list correspond one to one for each menu item.
CHAPTER 10

Commands

EXAMPLES

put "Hello" into field 1
put "go " before field "WhereTo"
put empty into It
put It -- puts contents of It into Msg
put "Tom" into first word of field "Name"
put "." after first character of last word of field 3
put fld 2 + fld 3 into fld 4 -- adds numbers in fields
put the date into varName
put "Paths" after menuItem "Preferences" of menu "Home"
put "Paths" after first menuItem of menu "Home" with ¬
menuMsg "go card 4"
put "Vanilla,Chocolate,Strawberry" into menu ¬
"Flavors" with menuMsg "put Yummy,put Tastie,put ¬
BerryGood"

DESCRIPTION

The first form of the put command causes HyperCard to evaluate expression and copy the result into container. You use the second form of the put command to add menu items to an existing menu. Optionally, you can specify a message to be sent when the menu item it belongs to is chosen.

User-defined items with the same name as standard HyperCard menu items inherit the standard behavior of the HyperCard menu item. For example, if you put an item called Background into a menu called Special, choosing it has the same effect as the standard Background menu item from the Edit menu unless you assign a custom menu message or intercept the doMenu message.

HyperCard does not automatically check, uncheck, enable, or disable user-defined menu items as it does for its own standard menu items. It is your responsibility to make sure user-defined menu items act according to the standard Macintosh user interface. See the commands and properties listed in the notes section for more information about controlling the behavior of menu items.
CHAPTER 10

Commands

SCRIPT

The following example handler initializes three global variables when the stack it’s in is opened:

on openStack
    global var1, var2, var3
    put 0 into var1
    put empty into var2
    put empty into var3
end openStack

NOTES

If you don’t specify the destination container, the value is copied into the Message box. (HyperCard shows the Message box if it’s hidden.) If you specify a container that HyperCard doesn’t recognize, it creates a new local variable of that name and puts the value into the variable.

For the put expression form, using into replaces the contents of the container, before places the source value at the beginning of the previous contents, and after appends the source value to the end of the previous contents.

You can use the put command to put text into buttons. The lines of the text of a pop-up button become the menu items of the pop-up menu that appears when the user clicks the button. Here’s an example:

set style of button 1 to popup
put menu font into button 1

If expression is a container holding an arithmetic expression, the expression is not evaluated but is copied literally into the destination. Use the value function with the container name to have HyperCard evaluate its contents.

You can delete the contents of a container by putting the constant empty or "" into it (but this doesn’t delete the container). You can specify a chunk expression to insert, replace, or delete a portion of the contents.

Always use the form put itemName before | after menuItem of menu to add a menu item to a menu that already has menu items in it. If you use the form put menuItem into menu, you replace the contents of the menu, deleting any other menu items already in the menu.
CHAPTER 10

Commands

Because menus are like containers, you can get a list of the current menu items in a menu by using the term menu as an expression. For example, the following statement puts a return-delimited list of all the menu items in the Home menu into card field “MenuItemList”:

```
put menu "Home" into card field "MenuItemList"
```

You can get a specified menu item name with a statement like

```
put menuItem 3 of menu "Home"
```

The maximum number of menu items in a menu is 64.

See also the checkMark, commandChar, enabled, menuMessage, name, and textStyle properties in Chapter 12, “Properties.” See also the create menu, delete, disable, and enable commands in this chapter.

Read

SYNTAX

```
read from file fileName [ at [-]start]
   for numberOfChars | until char | constant
```

*FileName* is an expression yielding the name of any file on your Macintosh; *start* is an integer expression identifying the position in the file where reading starts: a positive number indicates the character offset from the beginning of the file, and a negative number indicates the character offset from the end of the file.

*NumberOfChars* is an integer expression for the total number of characters to be read.

*Char* is an expression identifying the last ASCII character to be read (upper- and lowercase are distinguished).

*Constant* is one of the following: end, eof, formFeed, quote, return, space, or tab.
EXAMPLES

read from file "import" at 4 for 20
read from file "import" until tab
read from file "File Names" until return -- reads one line
read from file "myFile" at -20 until eof -- starts reading
   -- at 20 characters from the end of file

DESCRIPTION

The read command reads from the specified file, which must be opened already with the open file command, into the local variable It. Reading starts either at the position specified or, if no start is specified, from the character following the last point read with a previously executed read command. Reading continues until the specified character or constant is reached or until the specified number of characters has been read.

SCRIPT

The following example handler opens a file, reads to the end of the file while placing its contents into a global variable, and closes the file:

on mouseUp
   global fileName, textHolder
   open file fileName
   read from file fileName until eof
   put It into textHolder
   close file fileName
end mouseUp

NOTE

If you specify more than one character with the read until form, HyperCard stops reading when it finds the first character in the file.
CHAPTER 10

Commands

Tab characters
HyperCard reads tab characters from a file into it. When text containing tabs is put into a field, the tabs are displayed as spaces. The tabs are not removed when the text is altered; however, if null characters (ASCII 0) are read in, HyperCard changes them to spaces (ASCII 32).

Use the **close file** command to close files explicitly after you use them. HyperCard automatically closes all open files when an **exit** to HyperCard statement is executed, when you press Command-period, or when you quit HyperCard.

You must provide the full pathname of the file if it’s not at the same directory level as HyperCard. (See “Identifying a Stack” in Chapter 5 for an explanation of pathnames.)

If an error is generated while using the **read** command, an error dialog appears. See also the **close file**, **open file**, and **write** commands in this chapter, and the **result** function in Chapter 11, “Functions.”

---

**Reply**

**SYNTAX**

reply *expression* [with keyword *aeKeyword*]

reply error *expression*

*Expression* yields a HyperTalk statement. *AeKeyword* is an Apple event keyword.

**EXAMPLES**

reply "Hello there, nice to hear from you"

reply error "Error in the remote stack"
CHAPTER 10

Commands

DESCRIPTION
You use the reply command to answer an incoming Apple event. If you don’t specify a keyword for the reply parameter, then the parameter becomes the direct parameter of the reply.

You can use the form reply expression only to reply to a send command from another running copy of HyperCard. This form sets the result in the sending program to expression, where expression is any string or container.

You use the reply error expression form to reply to any Apple event. This form signals an error to the sending program.

SCRIPT
The following script handles Apple events of class 'WILD' and type 'defn' by searching for a string in a background field named “Glossary Entry” and returning the contents of a background field named “Definition.”

```on appleEvent eventClass, eventID, sender
  if eventClass is "WILD" and eventID is "defn"
    request appleEvent data
    find it in field "Glossary Entry"
    if the result is empty -- find is successful
      then reply field "Definition"
    else reply error "Not found"
  else pass appleEvent
end appleEvent```

NOTES
The reply command sets the result to No current Apple event when there is no current Apple event to handle.
CHAPTER 10

Commands

Request

SYNTAX

request expression from program
request expression of|from program id programID
request expression of|from this program
request appleEvent data with keyword aeKeyword
request appleEvent data|class|id|sender|return id|sender id

Expression is an expression that can be evaluated by the target program.
Program yields a valid program pathname in this form: zone:targetComputer:
targetProgram, where zone is a set of Macintosh computers on a local
network, targetComputer is the name of the target computer, and targetProgram
is the name of the target program. ProgramID is an application's signature
(4-character string). AeKeyword is an Apple event keyword.

EXAMPLES

request "the number of cards" from ¬
program "KZone:PMac:HyperCard"
request "the name of this stack" of program "HyperCard"
request "{target}" from program "MPW Shell"

DESCRIPTION

The request command sends an “evaluate expression” Apple event from
HyperCard to another application running remotely or on the same machine.
You can use this command to send an expression to any program that under-
stands the standard 'eval' Apple event. If the target program is another
copy of HyperCard, the expression you use as your request can be a built-in
HyperTalk function or property (such as the time or the long name of
this stack) or a user-defined function call (such as day()). When the target
program executes the statement, the result of the request (the value of the
expression) goes into the local variable It.

The request appleEvent data with keyword form puts the parameter
or attribute with the specified keyword into the local variable It. For example,
you can obtain a parameter of keyword *errs*, the standard Apple event keyword for an error string, as follows:

```
request appleEvent data with keyword "errs"
put it into errorString
```

If there is no attribute or parameter with the keyword you specify, HyperCard sets the result to *Not found*.

If you don’t supply a keyword, HyperCard assumes you’re requesting the direct object of the Apple event, which is defined by the Apple event manager as the parameter with keyword "----". The other `request appleEvent` forms support special cases for important attributes of Apple events.

**SCRIPT**

The following handler shows how the `request` command can get information from another HyperCard program:

```
on getStackName -- Card handler in source stack
    request "the long name of this stack" from program HildaPath
    if the result is empty
        then answer It
    else answer the result
end getStackName
```

**NOTES**

The parameter *zone* can be omitted from the program pathname when the target computer is in the same zone as the source computer.

The signature of an application program is a four-character field stored in its signature resource, which the application assigns to the creator field of its documents. For example, HyperCard’s signature is 'WILD'.

When the reply to the 'eval' Apple event sent by HyperCard doesn’t contain a direct parameter, the `request` command puts empty into the local variable *It*. If the Apple event server encounters an error when evaluating the expression and returns an error message in the reply event, the `request` command

---

`Request` 259
puts that message into the result. The following error messages go into the result when the request command fails:

<table>
<thead>
<tr>
<th>Condition</th>
<th>the result contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error while handling a 'misc' 'eval' event</td>
<td>Can’t take the value of that expression</td>
</tr>
<tr>
<td>Information returned is not recognized by HyperCard as text</td>
<td>Unknown data type</td>
</tr>
<tr>
<td>No attribute or parameter with the specified keyword</td>
<td>Not found</td>
</tr>
<tr>
<td>System software prior to version 7.0</td>
<td>Not supported by this version of the system</td>
</tr>
<tr>
<td>Target program didn’t handle event</td>
<td>Not handled by target program</td>
</tr>
<tr>
<td>Target program returned error number in reply, or AESend returned some other error</td>
<td>Got error &lt;errorNum&gt; when sending Apple event</td>
</tr>
<tr>
<td>Target program returned error string in reply</td>
<td>&lt;errorString&gt;</td>
</tr>
<tr>
<td>Target program timed out</td>
<td>Timeout</td>
</tr>
<tr>
<td>User canceled “Link to program” dialog</td>
<td>Cancel</td>
</tr>
</tbody>
</table>

See also the send keyword.

Reset Menubar

SYNTAX
reset menubar

DESCRIPTION
The reset menubar command reinstates the default HyperCard menus and removes any custom menus created with the create command and custom menu items put into standard HyperCard menus with the put command.
CHAPTER 10

Commands

NOTES

If you are creating a stack to be used by others, use this command with some restraint, because it removes all custom menus and menu items, not just those you created for your stack.

When your stack is closed, remove any custom menus or menu items you created by deleting them. Use a closeStack, suspend, or suspendStack system message handler to remove your custom menus when your stack is closed or is no longer the current stack, and an openStack, resume, or resumeStack system message handler to reinstate your custom menus when your stack is opened or resumed.

See also the create menu and put commands, earlier in this chapter, and Chapter 8, “System Messages.”

Reset Paint

SYNTAX

reset paint

DESCRIPTION

The reset paint command reinstates the default values of all the painting properties. The painting properties and their default values are

<table>
<thead>
<tr>
<th>Property</th>
<th>Default Value</th>
<th>Property</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>brush</td>
<td>8</td>
<td>pattern</td>
<td>12</td>
</tr>
<tr>
<td>centered</td>
<td>false</td>
<td>polySides</td>
<td>4</td>
</tr>
<tr>
<td>filled</td>
<td>false</td>
<td>textAlign</td>
<td>left</td>
</tr>
<tr>
<td>grid</td>
<td>false</td>
<td>textFont</td>
<td>geneva</td>
</tr>
<tr>
<td>lineSize</td>
<td>1</td>
<td>textHeight</td>
<td>16</td>
</tr>
<tr>
<td>multiple</td>
<td>false</td>
<td>textSize</td>
<td>12</td>
</tr>
<tr>
<td>multiSpace</td>
<td>1</td>
<td>textStyle</td>
<td>plain</td>
</tr>
</tbody>
</table>
Reset Printing

SYNTAX

reset printing

DESCRIPTION

The reset printing command reinstates the default values of all the printing properties. The printing properties and their default values are:

- printMargins: 0,0,0,0
- printTextAlign: left
- printTextFont: Geneva
- printTextHeight: 13
- printTextSize: 10
- printTextStyle: Plain

NOTE

The printing properties are described in Chapter 12, “Properties.”
DESCRIPTION

The `returnInField` command enters a return character into a field that is open for text editing.

NOTES

The `returnInField` message, which invokes the `returnInField` command if it reaches HyperCard, is normally generated by pressing the Return key on the keyboard, but you can also send it from the Message box or execute it as a line in a script.

See also the `returnInField` system message in Table 8-3.

ReturnKey

SYNTAX

`returnKey`

DESCRIPTION

The `returnKey` command sends a statement typed into the Message box to the current card.

NOTES

The `returnKey` message, which invokes the `returnKey` command if it reaches HyperCard, is normally generated by pressing the Return key on the keyboard, but you can also send it from the Message box or execute it as a line in a script.

See also the `returnKey` system message in Table 8-3.
CHAPTER 10

Commands

Save

SYNTAX

save stack stackName as [stack] fileName
save [this] stack as [stack] fileName

StackName is an expression that yields a valid stack name. FileName is an expression that yields a valid Macintosh filename.

EXAMPLE

save stack "Pottery" as "NewPottery"

DESCRIPTION

The save command saves a copy of the specified stack with the given filename. The stack is saved without a dialog box. Both stackName and fileName must be enclosed in quotation marks.

NOTE

If the save command produces an error, the error is stored in the HyperCard function the result.

Select

SYNTAX

select object
select [preposition] chunk of field
select [preposition] text of field
select line number [to number] of field
select line number of button
select empty
CHAPTER 10

Commands

Object is an expression that yields the descriptor of a button or field, or me, or target; preposition is before or after; chunk is a chunk expression; field is the descriptor of a field; number is an expression that evaluates to an integer; and button is the descriptor of a pop-up button.

(Button and field descriptors and the special descriptor me are explained in Chapter 5, “Referring to Objects, Menus, and Windows.” The special descriptor target is explained in Chapter 4, “Handling Messages.”)

EXAMPLES

select button 1
select before char 1 of field 2
select after text of field 2
select char 1 to 5 of card field "name"
select line 1 to 2 of field 1
select line 4 of button "My Pop-up"

DESCRIPTION

The select command creates a selection or highlights lines in a list field or pop-up button. The select object form chooses the appropriate tool and selects the object specified as though you had chosen the tool and clicked the object manually with the mouse. The forms specifying a field select text in the specified field and open the field for editing, unless the field is a list field. Before and after can be used to place the insertion point relative to the specified text or chunk of text. Using a chunk expression without a preposition selects the entire chunk, highlighting the characters in the chunk.

If the specified field is a list field (that is, its autoSelect and lockText properties are both true), you can use the select line form to select one or more whole lines, which then appear highlighted. You can also specify a pop-up button with the select line form to select one line from its contents, which then appears within the button rectangle.

The select empty form deselects highlighted text or removes the insertion point from a non-list field. It does not affect highlighted text in a list field or pop-up button.
CHAPTER 10

Commands

NOTES

For button families, the `selectedButton` function returns the descriptor of the button that is currently highlighted—selected by the user from the choices the button family represents. To set the `selectedButton` in a family from a script, set its `hilite` property to `true`.

You can select only the parts on the current card, so using `select button 1 of next card` won’t work. You do not get an error message when you try to do this.

See also the `selectedButton`, `selectedChunk`, `selectedField`, `selectedLine`, and `selectedText` functions in Chapter 11, “Functions.”

Set

SYNTAX

```
set [the] property [of element] to value
```

*Property* is a characteristic of a HyperCard object, menu, menu item, window, or chunk of a field. *Element* is an expression that yields the descriptor of an object, menu, menu item, window, or chunk of a field. *Value* is an expression that yields a valid setting for the particular property.

EXAMPLES

```
set name of field 1 to "Soccer"
set location of button "newButton" to the mouseLoc
set the visible of field 1 to "false" -- hide the field
set userLevel to 5 -- scripting
set the cmdChar of menuItem "Home Cards" of ¬ menu "Home" to 5
set loc of window "ask" to 10,10
set the textStyle of word 1 to 2 of field "Name" to bold
```
CHAPTER 10

Commands

DESCRIPTION
The set command changes the state of a specified property. If the element
to which the property belongs is not specified, the property must be a global
property.

Some properties cannot be changed with the set command. These exceptions
are pointed out in the property descriptions in Chapter 12, “Properties.”

SCRIPT
The following example handler automatically draws a circle on the
current card:

on mouseUp
    choose oval tool
    set linesize to 2
    set centered to true
    set dragspeed to 75 -- speed of expansion
    drag from 155,70 to 285,200
    choose browse tool
end mouseUp

NOTES
The properties of objects depend on the type of object. Generally, they are the
characteristics shown in the Info dialog boxes under the Objects menu. All of
the HyperCard properties are described in detail in Chapter 12, “Properties.”

Among the commands that set properties are: disable, enable, hide, mark,
reset menubar, reset printing, show, and unmark, which are described
in this chapter.
CHAPTER 10

Commands

Show

SYNTAX

show background picture
show card picture
show groups
show menuBar
show object [at point]
show picture of background
show picture of card
show titlebar
show window stackName [at point]
show window windowName [at point]

Object yields one of the following objects:

a valid button descriptor in the current stack
a valid field descriptor in the current stack
message [box] or message [window] or window "message"
pattern window or window "patterns" (the Patterns palette)
tool window or window "tools" (the Tools palette)
window "navigator" (the Navigator palette)
message watcher or window "message watcher"
variable watcher or window "variable watcher"
card window

Point is an expression yielding two integers separated by commas representing
the horizontal and vertical pixel offsets, respectively, on the screen. Card yields
the descriptor of a card in the current stack. Background yields the descriptor of
a background in the current stack. WindowName is the name of a window
created with the picture command or a custom external window. StackName
is the name of an open stack.
CHAPTER 10

Commands

EXAMPLES

show msg at 50,300
show tool window
show field "Names" at 1,1
show groups
show Message Watcher

DESCRIPTION

The show command displays a specified window or object at a specified location on the screen. If positioning offsets aren’t given, the window or object is displayed at its previous location.

The picture form of the show command displays a graphic bitmap on the card or background that has been hidden with the hide command. The show groups form of the show command makes a gray 2-pixel underline below group-style text. The show groups command affects all group-style text in all fields globally.

The show titlebar form of the show command shows the stack window title bar, if it was hidden. If it wasn’t hidden, it has no effect.

SCRIPT

The following example handler displays the Tools palette, the Patterns palette, and the Message box at their default locations when HyperCard first starts running:

on startUp
  show tool window
  show pattern window
  show msg
end startUp

NOTES

The show command sets the visible and, optionally, location properties of the window or object.
CHAPTER 10

Commands

If the menu bar is hidden and the screen is locked, show menubar has no effect. See the lock and unlock commands in this chapter and the lockScreen property in Chapter 12, “Properties.”

On Macintosh Plus and Macintosh SE screens, visible horizontal offsets range from 0 to 511, and visible vertical offsets range from 0 to 341. Members of the modular Macintosh family have variable visible offsets depending on the monitor currently in use.

Message can be abbreviated msg. Background can be abbreviated bkgnd or bg. Button can be abbreviated btn. Field can be abbreviated fld. Card can be abbreviated cd.

For buttons and fields of the current card, point specifies the distance from the top-left corner of the card window to the center of the button or field.

Card window refers to the current card window; for it and all stack windows, point specifies the distance from the top-left corner of the screen to the top-left corner of the card window, disregarding the title bar at the top of the window. For the other windows, point specifies the distance from the top-left corner of the card window to the top-left corner of the other window, disregarding the drag bar at the top of the window.

The default location for the Message Watcher window is the lower-left corner of the screen. The default location for the Variable Watcher is the lower-right corner of the screen.

The menu bar always shows at the top of the screen.

You can use the show and hide commands with the Navigator window only after it has been invoked with the palette command.

Valid window names for the show command are any of the windows returned by the windows function.

When you change a card window’s location property with the show card window at location form, the system message moveWindow is sent. The moveWindow message is also sent when you drag the window to a new location or zoom it in or out with the zoom box, causing the location property to change.

See also the hide, palette, and set commands in this chapter; the show system message in Table 8-3; the windows function in Chapter 11, “Functions”; and the location and visible properties in Chapter 12, “Properties.”
CHAPTER 10

Commands

Show Cards

SYNTAX

show [number] cards
show all cards
show marked cards

*Number* is an expression yielding an integer.

EXAMPLES

show all cards
show ten cards
show 26 cards
show marked cards
show howMany cards -- howMany contains an integer
show cards

DESCRIPTION

The *show cards* command displays the specified cards in the current stack in turn, beginning with the next card or, for the *show marked cards* form, the first marked card. If no parameter is used, *show cards* displays all cards in the stack continuously.

SCRIPT

The following example handler “prewars” the stack when you open it, so that going to cards in the stack subsequently will be faster, by caching the cards in RAM:

```
on openStack
  set lockScreen to true
  show all cards
  set lockScreen to false
end openStack
```

Show Cards

271
NOTES

The `show all cards` form shows all cards in the stack. HyperCard doesn’t send the `openCard` system message when a card is displayed by `show cards`, nor do visual effects occur. After the cards are shown, the last one shown (where you began in the case of `show all cards`) is the current card.

See also the `marked` property in Chapter 12, “Properties.”

Sort

SYNTAX

```
sort [sortDirection] [sortStyle] by sortKey
sort [this] stack [sortDirection] [sortStyle] by sortKey
sort [marked] cards [of this stack] [sortDirection] ¬
             [sortStyle] by sortKey
sort background [sortDirection] [sortStyle] by sortKey
sort [marked] cards of background [sortDirection] ¬
             [sortStyle] by sortKey
sort [lines|items of] container [sortDirection] ¬
             [sortStyle] [by sortKey]
```

*Container* is a field expression, variable, or the variable `each`.

*SortDirection* can be either ascending or descending; the default value is ascending.

*SortStyle* can be `text`, `numeric`, `dateTime`, or `international`; the default is `text`.

*SortKey* is any expression.

*Background* is an expression that yields a background descriptor.
EXCEPTION

sort lines of field 1 by last word of each
sort items of field 5 descending numeric by word 2 of each
sort numeric by second word of field 1
sort descending text by last word of field "Names"
sort cards of this stack by field "Names"
sort marked cards descending numeric by bg field 2
sort marked cards of background "Notes" by bg field 2
sort this background by field 1
sort lines of field 3 ascending
sort items of field 3 dateTime
sort field 3
sort it numeric

DESCRIPTION

The sort command can sort and order

- the lines or items in a container; if you do not specify lines or items when sorting a container, the default is lines (if you choose items, remember that HyperCard recognizes an item by its comma delimiter)
- all the cards or marked cards in a single background or a stack

You can customize the sort command by

- setting sortDirection to ascending or descending; it is ascending by default
- setting sortStyle to text, numeric, dateTime, or international; the default is text
- setting sortKey to any expression; for example, you could sort the lines in a field by the last word in each line by setting sortKey to the last word of each
CHAPTER 10
Commands

SCRIPT

The following example handler shuffles the cards in a stack randomly when the user goes to it from another stack:

```plaintext
on openStack
    sort numeric by random(the number of cards)
end openStack
```

NOTE

The international sort style assures correct sorting of non-English text containing diacritical marks and special characters, according to the international resources in your System file, your version of HyperCard, the Home stack, and the current stack.

The dateTime style sorts the stack using one of the forms of date or time (shown with the convert command, in this chapter), with earliest placed first in the ascending direction. The dateTime style also works correctly with non-English forms of date and time that have been modified by international resources in the System file.

See also the marked property in Chapter 12, “Properties,” and the mark command in this chapter.

Start Using

SYNTAX

```plaintext
start using stack stackName
```

*StackName* is an expression that yields a stack name.

EXAMPLE

```plaintext
start using stack "HD80:myStack"
```
CHAPTER 10

Commands

DESCRIPTION

The `start using` command inserts the specified stack between the current stack and the Home stack in the message-passing hierarchy. Each successive stack that is added to the message-passing hierarchy is inserted just after the current stack. If a stack that is already in use is “used” again, its previous place in the message-passing hierarchy changes to the place just after the current stack.

The `start using` command allows you to use the stack script and resources of any other HyperCard stack, not just the Home stack. Once the `start using` statement is executed, the handlers in the script of the specified stack and the XCMDs and other resources in its resource fork are available for use.

SCRIPT

To change the message-passing hierarchy when opening a stack, simply place a `start using` statement in the `openStack` handler:

```plaintext
on openStack
  start using stack "myStack:ScriptStack"
end openStack
```

To remove the stack from the hierarchy when closing a stack, place a `stop using` statement in the `closeStack` handler:

```plaintext
on closeStack
  stop using stack "myStack:ScriptStack"
end closeStack
```

NOTE

See also the description of the message-passing hierarchy in Chapter 4, “Handling Messages,” and the next command, `stop using`.

Start Using
CHAPTER 10

Commands

Stop Using

SYNTAX

\[
\text{stop using stack } stackName
\]

*StackName* is an expression that yields a stack name.

EXAMPLE

\[
\text{stop using stack } "HD80:myStack"
\]

DESCRIPTION

The *stop using* command removes the specified stack from the message-passing hierarchy.

SCRIPT

To remove the stack from the message-passing hierarchy when closing a stack, place a *stop using* statement in the *closeStack* handler.

```plaintext
on closeStack
    stop using stack "myStack:ScriptStack"
end closeStack
```

NOTE

See also the description of the message-passing hierarchy in Chapter 4, “Handling Messages,” and the previous command, *start using.*


CHAPTER 10

Commands

Subtract

SYNTAX

subtract number from [chunk of] container

*Number* is an expression that yields a number. *Chunk* is an expression that yields a chunk of a container. *Container* is an expression that identifies a container, such as a field, the Message box, the selection, or a variable.

EXAMPLES

subtract 2 from It
subtract field 1 from field 2

DESCRIPTION

The *subtract* command subtracts the value of *number* from the value of [chunk of] *container*, leaving the result in [chunk of] *container*. The value previously in *container* must be a number; it is replaced with the new value.

TabKey

SYNTAX

tabKey

DESCRIPTION

The *tabKey* command opens the first unlocked field on the current background or card (placing the text insertion point in the field) and selecting its entire contents. If a field is already open, *tabKey* closes it and opens the next field, selecting its contents.
CHAPTER 10

Commands

SCRIPT

The following example handler sets the insertion point in the first field so that the user can type something when the card is opened:

```on openCard
  tabKey
end openCard```

NOTES

The `tabKey` system message, which invokes the `tabKey` command if it reaches HyperCard, is normally generated by pressing the Tab key on the keyboard. But you can also send it from the Message box or execute it as a line in a script.

The `tabKey` command opens fields in the following order: from the lowest number to the highest, through the background fields first, then through the card fields.

See also the `tabKey` system message in Table 8-3.

Type

SYNTAX

```type text [with commandKey]```

*Text* is an expression that yields a text string.

*CommandKey* can be abbreviated *cmdKey*.

EXAMPLES

```type "Now is the time for all good persons."
type "p" with commandKey -- print card```
CHAPTER 10

Commands

DESCRIPTION

The type command enters the value of text at the text insertion point, as though you had typed it manually. If the with commandKey form is used, no text appears at the insertion point; rather, the action defined for the Command-key combination is carried out.

SCRIPT

The following example handler chooses the Browse tool, clicks the center of the specified field, and types a literal string:

on autoType
  choose browse tool
  click at the loc of field "whereToType"
  type "Automatic writing appears before your eyes..."
end autoType

NOTES

The text insertion point is placed by clicking an unlocked field with the Browse tool or by sending the tabKey message. Manipulating the text insertion point is described in the HyperCard Reference.

Paint text can be typed at the text insertion point on a card or background with the Paint Text tool selected.

Unlock

SYNTAX

unlock screen with [visual [effect]] effectName [speed]
  [to image]
unlock messages|error dialogs|recent

EffectName is an expression that yields any of the effect names described under the visual command later in this chapter. Speed is one of the following: fast, very fast, slow, slowly, very slow, or very slowly. Image is one of the following: black, card, gray, grey, inverse, or white.
CHAPTER 10

Commands

EXAMPLES

unlock screen with dissolve to black
unlock error dialogs
unlock recent

DESCRIPTION

The unlock command can be used for four different unrelated purposes. Using the unlock command, you can reset HyperCard to

- update the screen by setting the lockScreen global property to false. In addition, you can specify a single visual transition to occur when the screen is updated by using the visual effect option.

- send system messages such as openCard, closeCard, and so on, by setting the lockMessages property to false.

- display error dialogs in response to errors in executing scripts by setting the lockErrorDialogs property to false.

- record miniature representations of each card on the Recent card by setting the lockRecent global property to false.

NOTE

Visual effects can’t be compounded using unlock screen, as they can be using the visual command.

See also the visual and lock commands in this chapter and the lockErrorDialogs, lockMessages, lockRecent, and lockScreen properties in Chapter 12, “Properties.”
CHAPTER 10

Commands

Unmark

SYNTAX

unmark card
unmark cards where condition
unmark all cards
unmark cards by finding [international] text [in field]
unmark cards by finding chars [international] text [in field]
unmark cards by finding string [international] text [in field]
unmark cards by finding whole [international] text [in field]
unmark cards by finding word [international] text [in field]

Card is an expression that yields a card descriptor. Condition is an expression that yields the criteria on which you want to base the unmarking of cards. Text is an expression that yields any text. Field is an expression that yields a field descriptor.

EXAMPLES

unmark the next card
unmark cards where "We be shaking" is in field 2
unmark all cards
unmark card by finding word "fire" in bkgnd field 3

DESCRIPTION

The unmark command sets the marked property for the specified card or cards to false. The marked property of a card can also be changed with the Card Marked option in the Card Info dialog box. By default, the marked property of a card is false.

The by finding form of the mark command uses chars, word, whole, and string to define the search criteria the same way the find command does. See the description of the find command for information about how to use these forms.
CHAPTER 10

Commands

The unmark command can be used with the mark command in searches where you want to find and mark cards containing particular information while excluding other unnecessary information. See the description and script example used for the mark command, which is described earlier in this chapter.

NOTE

See also the marked property in Chapter 12, “Properties,” and the mark command, earlier in this chapter.

Visual

SYNTAX

visual [effect] effectName [speed] [to image]

EffectName is one of the following:

barn door close | open scroll up | down
checkerboard shrink to top | center | bottom
dissolve stretch from top | center | bottom
iris close | open venetian blinds
plain wipe left | right
push left | right wipe up | down
push up | down zoom close | open
scroll left | right zoom in | out

Speed is one of the following:

fast very fast
slow[ly] very slow[ly]
CHAPTER 10

Commands

*Image* is one of the following:

- black
- card
- gray
- inverse
- white

EXAMPLES

- visual effect barn door open
- visual dissolve slowly to white

DESCRIPTION

The *visual* command specifies a visual transition for HyperCard to use the next time it opens a card, as the current card is closed. The default *plain* visual effect causes all of the current image to be replaced immediately by the image of the next card. If you use the *to image* form, the visual effect occurs as a transition from the current card to a completely white, gray, or black screen image, to the inverted image of the next card, or to the image of the next card; *to card* is the default.

SCRIPT

The following example handler stacks two visual effects, which occur in succession, so that the transition appears as a fade to black, then to the next card:

```plaintext
on fadeOut
  visual effect dissolve to black
  visual effect dissolve to card
  go next card
end fadeOut
```

Visual 283
CHAPTER 10

Commands

NOTES

Visual effects don’t happen when you use the arrow keys or the show cards command to change cards; they occur only when go is executed, so they must be set up in a handler that also contains a go command. If a go command is not executed, visual effects set up in the handler are canceled when the handler finishes executing.

You can stack up several visual effects that will occur one after the other when you go to the next card.

See also the unlock command, earlier in this chapter.

Wait

SYNTAX

wait [for] time [seconds|ticks]
wait while | until condition

Time is an expression that yields an integer, and condition is an expression that yields true or false.

EXAMPLES

wait 60 seconds
wait until the mouse is down

DESCRIPTION

The wait command causes HyperCard to pause before executing the rest of the handler, either for a specific length of time, until a specified condition becomes true, or while a specified condition remains true.

If seconds is not specified for time, HyperCard uses ticks (1/60 second).
CHAPTER 10
Commands

SCRIPT
The following example handler allows time to view each card:

on slideshow
    repeat the number of cards
    visual effect dissolve slowly
    go next card
    wait 2 seconds
    end repeat
end slideshow

Write

SYNTAX
write text to file fileName [at [-]start|end|eof]

Text is an expression that yields text. FileName is an expression that yields a filename.
Start is an integer expression identifying the position in the file where reading starts. A positive number indicates the character offset from the beginning of the file; a negative number specifies a character offset from the end of the file.

EXAMPLES
write field "address" to file "myDisk:myFile"
write "first line" & return & "second line" to file "twoliner"

write someStuff to file "myFile" at -15
CHAPTER 10

Commands

DESCRIPTION

The write command causes HyperCard to copy the specified text to the specified disk file.

You can choose to specify the starting point at which to write the text. A negative number indicates the starting point to be a number of characters from the end of the file. If you specify either of the constants end and eof as the place to start, HyperCard appends the new text to the end of the file.

If you don’t specify a starting point, the first write command executed after opening a file replaces the previous contents of a file. HyperCard does not ask if you want to write over the existing file.

For the write command to work, you must have already opened the file with the open file command, and you should close it, when writing is completed, with the close file command.

SCRIPT

The following example handler opens a file specified in a global variable, writes the entire contents of the specified field to the file starting at character 5, then closes the file:

on writeFile
    global filename
    open file filename
    write background field 1 to file filename
    close file filename
end writeFile

NOTES

If a file is open for writing and you write to a file at a certain offset or a specified position, like eof or end, then HyperCard will not replace the file with the new text.

HyperCard replaces the previous contents of a file when it is opened and then written to sequentially, using write commands that do not specify the offset at which to write into the file. If any one of them does specify the offset, the file will contain all of the newly written data but will also include any of the preexisting text that was not specifically overwritten.
You must provide the full pathname of the file if it’s not at the same directory level as HyperCard. (See “Identifying a Stack” in Chapter 5 for an explanation of pathnames.)

If the file is locked or its disk is full, HyperCard displays an error dialog box and closes the file. HyperCard automatically closes all open files when an `exit to HyperCard` statement is executed, when you press Command-period, or when you quit HyperCard.

See also the `close file`, `open file`, and `read` commands in this chapter.
CHAPTER 11

Functions

This chapter describes HyperTalk's built-in functions.

A function is a named value that is calculated by HyperCard when the statement it is in executes. The value of a function changes according to conditions of the system or according to values of parameters that you pass to the function when you use it. When HyperCard reads a function name in a line of HyperTalk, it places the function's current value—its result—in that location before completing other actions.

Function Calls

To make a function call, that is, to use it in a HyperTalk statement, you must either use the word the before the function name or append parentheses after it. If a single parameter is passed to a function, the parameter can be enclosed in the parentheses or can follow the word of. (When of is used in this way to indicate the function call, the word the preceding the function name is optional.) If more than one parameter is passed to a function, all parameters must be enclosed in the parentheses and separated from each other by commas. Here are some examples of function calls:

put the time into msg
put time() into background field "Time"
put the length of myVariable into card field "howLong"
put average(total_1,total_2,total_3) into Projection
get the clickChunk

You can define your own functions in HyperTalk using the function handler structure described in Chapter 9.
User-defined functions override built-in ones with the same name
If you define your own function having the same name as a built-in one, yours overrides the built-in one if the function call is made with the parentheses syntax (unless the function call is made farther along the hierarchy than the handler’s script).

You can call the built-in functions of HyperCard directly and bypass any user-defined functions by using the word the before the function name. You can also use of, rather than using the parentheses syntax; however, functions having more than one parameter always require parentheses.

Syntax Description Notation

The syntax descriptions use the following typographic conventions. Words or phrases in this font are HyperTalk language elements or are those that you type to the computer literally, exactly as shown. Words in italics describe general elements, not specific names—you must substitute the actual instances. Brackets ([ ]) enclose optional elements that may be included if you need them. (Don’t type the brackets.)

It doesn’t matter whether you use uppercase or lowercase letters; names that are formed from two words are shown in lowercase letters with a capital in the middle (likeThis) merely to make them more readable.

The terms factor and expression are defined in Chapter 7, “Expressions.” Briefly, a factor can be a constant, literal, function, property, number, or container, and an expression can be a single factor or a complex expression built with factors and operators. Also, a factor can be an expression within parentheses.

The term yields indicates a specific kind of value, such as a number or a text string, that must result from evaluation of an expression when a restriction applies (for example, the factor or expression used with the abs function must yield a number). However, any HyperTalk value can be treated as a text string.
Function Descriptions

The rest of this chapter describes the functions supported by HyperCard 2.2.

Abs

**SYNTAX**

the abs of factor
abs(expression)

*Factor and expression yield numbers.*

**EXAMPLE**

put abs(a-b) into field "theOffset"

**DESCRIPTION**

The abs function returns the absolute value (makes the sign positive) of the number passed to it.

Annuity

**SYNTAX**

annuity(rate, periods)

*Rate and periods are expressions that yield numbers.*
EXAMPLES

put myPayment*annuity(.015,12) into presentValue
put myPayment*annuity(.015,12)*compound(.015,12)¬
       into futureValue

DESCRIPTION

The annuity function is used to compute the present or future value of an
ordinary annuity. Rate is the interest rate per period, and periods is the number
of periods over which the value is calculated. The formula for annuity is

\[
anuity(rate, \text{ periods}) = \frac{1 - (1 + rate)^{-periods}}{rate}
\]

The annuity function is more accurate than computing the formula above
using basic arithmetic operations and exponentiation, especially when rate
is small.

NOTE

See also the compound function, later in this chapter.

Atan

SYNTAX

the atan of factor
atan(expression)

Factor and expression yield numbers.

EXAMPLE

put atan(1.0) into field "arcTan" -- yields 0.785398
CHAPTER 11

Functions

DESCRIPTION

The atan function returns the trigonometric arc tangent (inverse tangent) of the number passed to it: that is, the angle whose tangent is equal to the given value. The result is expressed in radians.

Radians can be converted to degrees by multiplying by 180 and dividing the result by the value of the constant pi.

SCRIPT

The following example handler converts a value in radians to degrees and puts the result into the Message box:

on radiansToDegrees var
    put round((atan(var)*180)/pi) into msg
end radiansToDegrees

Average

SYNTAX

average(list)

List is a sequence of comma-separated expressions that yield numbers, or it is a single container that contains such a sequence.

EXAMPLE

put average(1,2,3) into field "avg"

DESCRIPTION

The average function returns the average of the numbers passed to it.
The following example handler displays the average of a list of numbers contained in one line of a field:

```
on avgSupplyPrice
  put "12.95,10.50,14.75,15.00,9.95" into line 3 of field "suppliers"
  answer "Average widget cost: " && average (line 3 of field "suppliers")
end avgSupplyPrice
```

**CharToNum**

**SYNTAX**

```
the charToNum of factor
charToNum (expression)
```

*Factor and expression yield a character.*

**EXAMPLE**

```
put the charToNum of "a" into It -- yields 97
```

**DESCRIPTION**

The `charToNum` function returns an unsigned integer representing the ASCII equivalent value of the character passed to it.

**NOTES**

If more than one character is passed, `charToNum` returns the ASCII value of the first character. If `factor` is a literal, it must appear within quotation marks. See also the `numToChar` function, later in this chapter.
CHAPTER 11

Functions

ClickChunk

SYNTAX

the clickChunk

clickChunk()

EXAMPLES

put the clickChunk into card field "ExpressMyClick"

get the clickChunk

DESCRIPTION

The clickChunk function returns a chunk expression referring to the text clicked in a field and is typically something like char 1 to 3 of background field 3.

ClickChunk refers to the single word clicked, with the definition of a word being any characters delimited by white space (commas, tabs, spaces, returns, and so on). If the location clicked has the style group, then the largest contiguous run of text that has the group style is returned, thus allowing ranges or phrases rather than just single words to be referred to. Group is a possible value of the textStyle property.

SCRIPT

Because chunks of text also have properties, you could use the clickChunk function to examine the textStyle property of a chunk of text in a field and then take the appropriate action based on that style of text. For example, you might have an application that provides definitions for any boldface words when one of the words is clicked.

The following handler placed in a script of a locked field examines the style of a chunk of text clicked in that field and, if that text is boldface, calls up another field containing definitions:

on mouseUp
  if the textStyle of the clickChunk is bold
    then show card field "definitions"
end mouseUp
CHAPTER 11
Functions

NOTE
See also the clickLine, clickLoc, and clickText functions in this chapter and the textStyle property in Chapter 12, “Properties.”

ClickH

SYNTAX
the clickH
clickH()

EXAMPLE
put the clickH into card field "horizontalOffset"

DESCRIPTION
The clickH function returns an integer that represents the number of horizontal pixels from the left side of the card window to the place where the mouse was last clicked.

NOTE
See also the clickV function in this chapter.

ClickLine

SYNTAX
the clickLine
clickLine()
CHAPTER 11
Functions

EXAMPLES
put the clickLine into card field "MyClick"
get the clickLine

DESCRIPTION
The clickLine function returns the specification of the line (based on actual return characters, not display lines) that was clicked. A typical result is line 5 of card field 2. One sentence or line of text may not fit in the width of a field and have to wrap onto subsequent lines on the display. It may appear as though more than one line is in the field, but if the lines are not delimited with return characters, clickLine returns an expression like line 1 of card field 5 when a user clicks the text in that field. If you want clickLine to return a unique line number for each line that is displayed within a field, be sure to end each line in the field with a return character.

NOTE
See also the clickChunk, clickLoc, and clickText functions in this chapter.

ClickLoc

SYNTAX
the clickLoc
clickLoc() 

EXAMPLE
put the clickLoc into card field "mostRecentClick"

DESCRIPTION
The clickLoc function returns the point on the screen where the user most recently clicked before the handler started executing. The location is determined at the time the message is first sent—the mouse could be elsewhere by
ClickText

the time the message is received. The location point is returned as two integers separated by a comma, representing horizontal and vertical pixel offsets from the top-left corner of the card.

SCRIPT

The following example handler, when it is in the script of a locked field, selects a word in the field when the user clicks the word:

```
on mouseUp
    set locktext of me to false -- unlock the locked field
    -- next two lines double-click the location
    click at the clickLoc
    click at the clickLoc
    put "You clicked the word:" && the selection
    set lockText of me to true -- must lock it again
end mouseUp
```

ClickText

SYNTAX

```
the clickText
```

EXAMPLES

```
put the clickText into card field "ExpressMyClick"
getch the clickText
```

DESCRIPTION

ClickText returns the text of a single word clicked, with the definition of a word being any characters delimited by white space (commas, tabs, spaces, returns, and so on). If the location clicked has the style group, then the largest
CHAPTER 11

Functions

contiguous run of text that has the group style is returned, thus allowing ranges or phrases rather than just single words to be referred to and analyzed. Group is a possible value of the textStyle property.

SCRIPT

The clickText function can be used to implement glossary lookup or provide other hypertext-type functions with something like the following handler, which should be placed in the script of the field being clicked:

```script
on mouseUp
  get the clickText -- must be done before leaving card
  lock screen
  go stack "Glossary"
  find it in field "Words" -- is it a glossary entry?
  if the result = "Not found" then go back
  else unlock screen with dissolve -- display glossary
     -- entry
end mouseUp
```

NOTE

See also the clickChunk, clickLoc, and clickLine functions in this chapter and the textStyle property in Chapter 12, “Properties.”

ClickV

SYNTAX

the clickV
clickV()

EXAMPLE

put the clickV into card field "verticalOffset"
CHAPTER 11
Functions

DESCRIPTION
The `clickV` function returns an integer that represents the number of vertical pixels from the top of the card window to the place where the mouse was last clicked.

NOTE
See also the `clickH` function in this chapter.

CommandKey

SYNTAX

```
the commandKey
commandKey()
```

EXAMPLE

```
if the commandKey is up then put "Wow" into msg box
```

DESCRIPTION
The `commandKey` function returns the constant `up` if the Command key is not pressed or `down` if it is pressed.

NOTES
The `commandKey` function name can be abbreviated `cmdKey`.
See also the `optionKey` and `shiftKey` functions, later in this chapter.
CHAPTER 11

Functions

Compound

SYNTAX

compound (rate, periods)

Rate and periods are expressions that yield numbers.

EXAMPLES

put futureValue/compound(.10,12) into presentValue
put presentValue*compound(.10,12) into futureValue

DESCRIPTION

The compound function is used to compute the present or future value of a compound interest-bearing account. Rate represents the interest rate per period, and periods is the number of periods over which the value is calculated. The formula for compound is

compound (rate, periods) = (1 + rate) periods

The compound function is more accurate than computing the formula expression above using standard arithmetic operations and exponentiation, especially when rate is small.

SCRIPT

The following example handler calculates the value in one year of an account earning 7½ percent interest compounded monthly:

on calcInterest
    ask "Enter the beginning balance:" with empty
    set numberFormat to ".00" -- dollars and cents format
    put "Value in 1 year $" & it * compound(.075/12,12)
end calcInterest
CHAPTER 11

Functions

NOTE

See also the annuity function, earlier in this chapter.

Cos

SYNTAX

the cos of factor
cos(expression)

Factor and expression yield numbers.

EXAMPLE

put the cos of 2 -- puts -.416147 into the Message box

DESCRIPTION

The cos function returns the cosine of the angle that is passed to it. The angle must be expressed in radians.

NOTE

Radians can be converted to degrees by multiplying by 180 and dividing the result by the value of the constant pi.

Date

SYNTAX

the [adjective] date
date()

Adjective is long, short, or abbreviated (or abbrev or abbr); the default adjective is short.
EXAMPLE

put last word of the long date into background field "Year"

DESCRIPTION

The date function returns a string representing the current date set in your Macintosh. There are three forms of the date function. Here are examples of the format used by each:

the short date 7/20/93
the long date Tuesday, October 7, 1989
the abbrev date Tue, Oct 20, 1992

SCRIPT

The following example handler puts the current date into a field when another field (whose script contains the handler) is changed:

on closeField
    put the long date into field "lastUpdate"
end closeField

NOTES

The format of the date is initially specified by the international resources in the System file. These resources can and are altered for the purpose of customization or localization, so that a French system, for example, can display dates using French names for months and days of the week in the standard French formats.

Therefore, though the date function always returns the same basic data—the current date—the format of the data is not fixed. This issue is important for anyone who wants to build Stackware that works anywhere without modification.

You cannot assume that the long date always returns a date in this format: <day of week>, <month> <day>, <year>. If your stack is used on a Swedish system, a script that assumes that item 1 of the long date is the day of the week will not work since the Swedish format is not delimited by commas and has this format: <day of week> < day> <month> <year>. 
Make sure you convert and store all dates in the invariant `dateItem` format before doing calculations to prevent problems that are due to different local date formats. (See the `convert` command in Chapter 10.)

### Destination

**SYNTAX**

```plaintext
the destination
destination()
```

**EXAMPLE**

```plaintext
on closeStack
    global stacksInMySuite
    if the destination is in stacksInMySuite then
        -- don't cleanup
    else
        -- cleanup: remove stack in use, restore menubar
    end if
    pass closeStack
end closeStack
```

**DESCRIPTION**

Returns the name of the destination stack when HyperCard is in the process of going to another stack. The destination is available to handlers for `closeCard`, `closeBackground`, `closeStack`, and `suspendStack`. If HyperCard is not going to another stack, this function returns the pathname of the current stack.
CHAPTER 11

Functions

DiskSpace

SYNTAX

the diskSpace
diskSpace()

EXAMPLE

if the diskSpace < 100000 then answer ¬
"Your disk is getting full."

DESCRIPTION

The diskSpace function returns an integer representing the number of bytes of free space on the disk that contains the current stack.

SCRIPT

The following function handler is used by the second handler (for the writefile message) to ensure that there is enough space on a disk to write to a file on that disk:

function thereIsRoom size
    return (the diskSpace > size)
end thereIsRoom

on writefile
    global var -- the text to be saved
    put "MyFilename" into fileName
    if thereIsRoom(length of var) then
        open file fileName
        write var to file fileName
        close file fileName
    else answer "Can't write that file; the disk is full."
end writefile
CHAPTER 11
Functions

Exp

SYNTAX
the exp of factor
exp(expression)

Factor and expression yield numbers.

EXAMPLE
put the exp of 2 -- puts 7.389056 into the Message box

DESCRIPTION
The exp function returns the mathematical exponential of its argument (the constant e, which equals 2.7182818, raised to the power specified by the argument).

Exp1

SYNTAX
the expl of factor
expl(expression)

Factor and expression yield numbers.

EXAMPLE
put the expl of 2 -- puts 6.389056 into the Message box
CHAPTER 11

Functions

DESCRIPTION

The \texttt{exp1} function returns 1 less than the mathematical exponential of its argument (1 less than the result of the constant $e$ raised to the power specified by the argument). That is, it computes

$$\exp(number) - 1$$

Exp2

SYNTAX

the \texttt{exp2} of factor
\texttt{exp2(expression)}

\textit{Factor} and \textit{expression} yield numbers.

EXAMPLE

put the \texttt{exp2} of 16 -- puts 65536 into the Message box

DESCRIPTION

The \texttt{exp2} function returns the value of 2 raised to the power specified by the argument.

FoundChunk

SYNTAX

the \texttt{foundChunk}
\texttt{foundChunk()}
CHAPTER 11

Functions

EXAMPLE

put the foundChunk

DESCRIPTION

The foundChunk function returns a chunk expression describing the location of the text found in a field with the find command. For example, if field 1 contained Now is the time, the commands (placed inside a handler)

find "Now"
put the foundChunk

would put char 1 to 3 of bkgnd field 1 into the Message box.

NOTE

See also the find command in Chapter 10.

FoundField

SYNTAX

the foundField
foundField()

EXAMPLE

put the foundField

DESCRIPTION

The foundField function returns the descriptor of the field in which the text was found with the find command. The result is in a form such as card field 1.
CHAPTER 11

Functions

SCRIPT

The following handler uses foundField to put the field descriptor of a field containing the specified word in the Message box:

on getField theWord
  lock screen
  push card
  find theWord
  if the result is "not found" then
    put "It's not here"
  else
    put the number of this card into cardNum
    put "Your word was found in" && the foundField && ¬
    "of card number" && cardNum
  end if
  pop card
  unlock screen
end getField

To make the script work, put it in the stack or background script, then enter getField followed by the word you want to find in the Message box.

SYNTAX

the foundLine
foundLine()

EXAMPLE

put the foundLine
CHAPTER 11

Functions

DESCRIPTION
The foundLine function returns a chunk expression describing the line in which the beginning of the text was found with the find command. The result is in a form such as line 1 of card field 2.

FoundText

SYNTAX
the foundText
foundText()

EXAMPLE
put the foundText

DESCRIPTION
The foundText function returns the characters that are enclosed in the box after the find command has executed successfully; for example, the commands

find "Hyper"
put the foundText

would put HyperCard in the Message box if it were the word containing the matching string.

HeapSpace

SYNTAX
the heapSpace

310 FoundText
CHAPTER 11

Functions

EXAMPLE

put the heapSpace into card field "Heap O'Fun"

DESCRIPTION

The heapSpace function returns an integer representing the remaining number of bytes of heap space currently available to HyperCard. The amount of heap space determines performance-related issues, such as whether the user can use the Paint tools, or whether HyperCard can open a stack in a new window.

SCRIPT

The following handler ensures that there is enough memory available for HyperCard to open a palette:

on openPalette
    get the heapSpace
    if it is < 100 then
        answer "Not enough memory to open this palette."
    end if
end openPalette

Length

SYNTAX

the length of factor
length(expression)

Factor and expression yield text strings.
CHAPTER 11

Functions

EXAMPLES

put length("tail") into It -- yields 4
if the length of word n of field 5 > 25
then add 1 to fogIndex

DESCRIPTION

The length function returns the number of characters (including spaces, tabs,
and return characters) in the text string passed to it.

NOTE

If expression is a literal, it must appear within quotation marks. The length
function is identical in effect to the following form of the number function:

the number of characters in factor

ln

SYNTAX

the ln of factor
ln(expression)

Factor and expression yield numbers.

EXAMPLE

put the ln of 10 -- puts 2.302585 into Message box

DESCRIPTION

The ln function returns the base-e (natural) logarithm of the number passed
to it.
CHAPTER 11

Functions

Ln1

SYNTAX

the ln1 of factor
ln1(expression)

Factor and expression yield numbers.

EXAMPLE

put the ln1 of 10 -- puts 2.397895 into Message box

DESCRIPTION

The ln1 function returns the base-e (natural) logarithm of the sum of 1 plus the number passed to it. That is, it computes

\[ \ln(1 + \text{number}) \]

If number is small, ln1 of number is more accurate than ln(1+number).

Log2

SYNTAX

the log2 of factor
log2(expression)

Factor and expression yield numbers.

EXAMPLE

put the log2 of 10 -- puts 3.321928 into the Message box

DESCRIPTION

The log2 function returns the base-2 logarithm of the number passed to it.
Max

SYNTAX

\[ \text{max} \left( \text{list} \right) \]

List is a sequence of comma-separated expressions that yield numbers, or it is a single container that contains such a sequence.

EXAMPLE

\[ \text{put max}(5,10,7.3) \rightarrow \text{puts 10 into the Message box} \]

DESCRIPTION

The max function returns the highest-value number from a list of numbers passed to it. If the source of the list is a container with more than one line in it, only the first line is used.

SCRIPT

The following example handler displays the highest number in a list contained in a variable:

\[
\text{on highStock}
\text{put "12.50,10,7.95,14.76,13.70" into stockPrices}
\text{answer "The highest price for the month is:"} \ ¬
\text{&& max(stockPrices)}
\text{end highStock}
\]
CHAPTER 11

Functions

Menus

SYNTAX

the menus
menus()

EXAMPLES

put the menus into card field 2
put menus() into MyVar -- MyVar is a variable

DESCRIPTION

The \textit{menus} function returns a return-delimited list of all the menus currently in the HyperCard menu bar, including the Apple menu and any custom menus. In System 7, the list includes system menus.

NOTE

See also the \texttt{create menu} command in Chapter 10.

Min

SYNTAX

\texttt{min(list)}

\textit{List} is a sequence of comma-separated expressions that yield numbers, or it is a single container that contains such a sequence.

EXAMPLE

put \texttt{min(5,10,7.3)} -- puts 5 into the Message box
CHAPTER 11

Functions

DESCRIPTION

The min function returns the lowest-value number from a list of numbers passed to it. If the source of the list is a container with more than one line in it, only the first line is used.

SCRIPT

The following example handler displays the lowest number in a list contained in a variable:

on lowStock
    put "12.50,10,7.95,14.76,13.70" into stockPrices
    put "The lowest price for the month is:" ¬
    && min(stockPrices)
end lowStock

Mouse

SYNTAX

the mouse
mouse()

EXAMPLE

if the mouse is up then put "Press the mouse button"

DESCRIPTION

The mouse function returns the constant up if the mouse button is not pressed, down if it is pressed.
The following example handler determines whether the user has single-clicked or double-clicked the button whose script contains the handler:

```scrit
on mouseUp
    put the ticks into start
    repeat until the ticks-start > 4 -- click speed
        if the mouse is "down" then
            go last card -- put your double-click action here
            exit mouseUp
        end if
    end repeat
    go next card -- put your single-click action here
end mouseUp
```

### MouseClick

#### Syntax

```
the mouseClick

mouseClick()
```

#### Example

```
if the mouseClick then put the mouseLoc
```

#### Description

The `mouseClick` function determines if the mouse button is clicked. If no click is sensed, the `mouseClick` immediately returns the constant `false`. The `mouseClick` function returns the constant `true` when the mouse button is clicked. The `mouseClick` function does not return `true` more than one time for a given mouse click.
CHAPTER 11
Functions

SCRIPT

The following example handler demonstrates operation of the mouseClick function by informing the user whether or not it sensed a click during its execution:

```plaintext
on mouseUp
    put "Click or don't click..."
    wait 5 seconds
    if the mouseClick then
        put "You clicked."
    else
        put "You didn't click."
    end if
end mouseUp
```

MouseH

SYNTAX

the mouseH
mouseH()  

EXAMPLE

if the mouseH > 319 then put "Stop"

DESCRIPTION

The mouseH function returns an integer representing the number of horizontal pixels from the left side of the card to the current location of the pointer.
CHAPTER 11

Functions

MouseLoc

SYNTAX

the mouseLoc

mouseLoc()

EXAMPLE

show button "everReady" at the mouseLoc

DESCRIPTION

The mouseLoc function returns the point on the screen where the pointer is currently located. This point is returned as two integers separated by a comma, representing horizontal and vertical pixel offsets from the top-left corner of the card.

SCRIPT

The following example handler, in a button script, allows the user to drag the button around the screen:

on mouseDown
    repeat until the mouse is up
        set the loc of me to the mouseLoc
    end repeat
end mouseDown
CHAPTER 11

Functions

MouseV

SYNTAX

the mouseV
mouseV()

EXAMPLE

if the mouseV > 199 then put "Stop"

DESCRIPTION

The mouseV function returns an integer representing the number of vertical pixels from the top of the card to the current location of the pointer.

Number

SYNTAX

[the] number of objects
[the] number of chunks in expression
[the] number of backgrounds [in this stack]
[the] number of cards in background
[the] number of cards [in this stack]
[the] number of marked cards
[the] number of menus
[the] number of menuItems of menu
[the] number of [card|background] parts
[the] number of windows

Objects is [background] buttons, [card] fields, backgrounds, cards, or parts. Chunks is characters (or chars), words, items, or lines, and expression yields a container or text string. Background is an expression that yields the descriptor of a background in the current stack. Menu is an expression that yields a menu descriptor.
CHAPTER 11

Functions

EXAMPLES

put the number of buttons into It
put number of items of line 1 of field 2 into listSize
put the number of chars in msg into line 3 of field 2
if number of chars in myVar > 10 then put "Big" into msg
get the number of cards of bkgnd 3

DESCRIPTION

The number function returns the number of buttons, fields, or parts on the current card or on its background, the number of backgrounds or cards in the current stack, the number of chunks of a specified kind in a designated container or text string, the number of cards that are associated with a specified background, the number of marked cards in the current stack, the number of menus in the menu bar, the number of menu items in a specified menu, or the number of windows currently available to HyperCard.

SCRIPT

The following example handler uses the number function to delete all the card fields on a card, regardless of how many there are:

on deleteFields
    repeat with whichField = the number of card fields ¬
down to 1
        -- you must count down like this, not up
        delete card field whichField
    end repeat
end deleteFields

NOTES

If backgrounds is not specified with buttons, the number of card buttons is returned; if card is not specified with fields, the number of background fields is returned; if backgrounds is not specified with parts, the number of card parts is returned. If the number function is used with a chunk name, it returns the number of chunks of that kind within the designated container or other factor yielding a text string.
The factor can be a chunk expression, so you can get the number of chunks of one kind within another chunk:

the number of chars in first word of field 1

You can also use the syntax that uses parentheses with the number function—for example:

number(cards)
number(menus)
number(bkgnds)
number(fields)

Backgrounds can be specified with the abbreviation bkgnds or bgs.

See also the number and marked properties in Chapter 12, and Chapter 5, “Referring to Objects, Menus, and Windows.”

---

**NumToChar**

**SYNTAX**

the numToChar of factor
numToChar(expression)

*Factor and expression yield positive integers.*

**EXAMPLE**

put numToChar(67) into word 4 of line 9 of field ¬ "ASCII Chart" -- yields C

**DESCRIPTION**

The numToChar function returns the character whose ASCII equivalent value is that of the integer passed to it.
SCRIPT

The following example handler turns all of the lowercase letters in a field into uppercase letters:

```
on upperCase
    put card field 4 into temp
    -- variables are faster than fields
    repeat with count = 1 to the length of temp
        get character count of temp
        if charToNum of It > 96 and ¬
        charToNum of It < 123 then
            put numToChar(charToNum(It)-32) into ¬
            character count of temp
        end if
    end repeat
    put temp into card field 4
end upperCase
```

NOTE

See also the `charToNum` function, earlier in this chapter.

Offset

SYNTAX

```
offset(string1,string2)
```

`String1` and `string2` are both expressions yielding text strings.

EXAMPLES

```
put offset("hay",field 1) into the Message box
offset("a","abc") -- typed in msg, returns 1
```
CHAPTER 11
Functions

DESCRIPTION
The offset function returns the number of characters from the beginning of the string2 string to the character at which string1 begins. If string1 doesn't appear within string2, 0 is returned.

SCRIPT
The following function handler finds every occurrence of a string within a container, and it replaces every occurrence with a second string:

```plaintext
function searchAndReplace container,original,replacement
    put length of original - 1 into theEnd
    repeat until original is not in container
        -- loop until all are replaced
        put offset(original,container) into start
        -- set start to location of original
        put replacement into char start to ¬
            start + theEnd of container
    end repeat
    return container
end searchAndReplace
```

NOTE
The parameters passed to the offset function can both be arithmetic or logical (as well as text) expressions; after evaluation, the results are treated as strings.

OptionKey

SYNTAX
the optionKey
optionKey()
Example

if the optionKey is down then choose button tool

Description

The `optionKey` function returns the constant `up` if the Option key is not pressed, `down` if it is pressed.

Note

See also the `commandKey` and `shiftKey` functions in this chapter.

Param

Syntax

```
the param of factor
param(expression)
```

*Factor* and *expression* yield integers.

Example

if param(1) is empty then answer-^ 
  "The first parameter is null."

Description

The `param` function returns a parameter value from the parameter list passed to the currently executing handler. The parameter returned is the *n*th parameter, where *n* is the integer derived from *factor* or *expression*. The value of `param(0)` is the message name.
CHAPTER 11

Functions

SCRIPT

The following example handler sums the numeric arguments passed to it, regardless of how many there are:

```script
on addUp -- adds a variable number of arguments
    put 0 into total
    repeat with i = 1 to the paramCount
        add param(i) to total
    end repeat
    put total
end addUp
```

NOTE

See also the paramCount and params functions, in this chapter, and the discussion of parameter passing in Chapter 4, “Handling Messages.”

ParamCount

SYNTAX

the paramCount
paramCount()

EXAMPLE

```script
if the paramCount < 3 then ¬
    put "I need at least three arguments."
```

DESCRIPTION

The paramCount function returns the number of parameters passed to the currently executing handler.

326 ParamCount
CHAPTER 11
Functions

SCRIPT

The following example handler draws an oval differently depending on the number of parameters passed to it:

on drawOval
  if the paramCount is 2 then set lineSize to param(2)
  choose oval tool
  drag from 30,30 to 30 + param(1),30 + param(1)
  choose browse tool
  reset paint
end drawOval

NOTE

See also the param and params functions, in this chapter, and the discussion of parameter passing in Chapter 4, “Handling Messages.”

Params

SYNTAX

the params
params()

EXAMPLE

put the params into field "messageReceived"

DESCRIPTION

The params function returns the entire parameter list, including the message name, passed to the currently executing handler.
CHAPTER 11

Functions

SCRIPT

The following example handler is useful primarily for debugging, to see if the parameters passed to a handler are correct:

```
on myMessage
    put the params
    -- rest of myMessage handler goes here
end myMessage
```

NOTE

See also the `param` and `paramCount` functions, in this chapter, and the discussion of parameter passing in Chapter 4, “Handling Messages.”

Programs

SYNTAX

```
the programs
programs()
```

EXAMPLES

```
answer the programs
put programs() into field id 4
```

DESCRIPTION

The `programs` function produces a return-delimited list of all the System 7–friendly processes currently running on your machine.

NOTE

See also the `answer` command in Chapter 10 and the discussion of Apple events in Chapter 1.

328 Programs
CHAPTER 11

Functions

Random

SYNTAX

the random of factor
random(expression)

Factor and expression yield positive integers.

EXAMPLE

set the loc of button "jumpy" to random(320),random(200)

DESCRIPTION

The random function returns a random integer between 1 and the integer derived from factor or expression, inclusive. Random supports integers up to $2^{31}-2$.

SCRIPT

The following example handler draws 10 unique random numbers between 1 and 100:

on mouseUp
  put empty into randomList
  put the itemDelimiter into delim
  repeat until the number of items in randomList is 10
    get random of 100
    if (delim&it&delim) is not in (delim&randomList) then
      put it & delim after randomList
    end if
  end repeat
  -- get rid of extra item delimiter
  delete last char of randomList
  put randomList
end mouseUp
CHAPTER 11

Functions

Result

SYNTAX

the result
result()  

EXAMPLE

if the result is not empty then answer "Try again."

DESCRIPTION

The result function returns an explanatory text string if an immediately preceding close file, convert, create stack, find, go, import paint, export paint, open file, picture, read, or save command was unsuccessful. The result function returns empty if the command executed successfully. The value of the result can also be set by a return statement in a message handler or by an external command. The value of the result is reset by execution of another command and at the end of the handler.

SCRIPT

The following example handler searches for a string and displays either the string or the error message if it doesn’t find the string:

on doMenu var
  if var is "Find..." then
    global findMe
    repeat
      ask "Find what string:" with findMe
      if It is not empty then find It
      else exit doMenu -- cancel clicked
    if the result is not empty then
      put the result into findMe -- display error
    next repeat

330 Result
CHAPTER 11

Functions

else
    put It into findMe -- display string
    exit repeat
end if
end repeat
else pass doMenu
end doMenu

NOTES

It is safer to depend on the empty result of a successful execution than on the particular value of some error message, because those values could be different in future versions of HyperCard.

If any of the commands listed in the result function description are sent from the Message box and generate an error, HyperCard displays the text of the result function in a dialog box. For example, if you sent a go command using the without dialog form from the Message box and used the name of a stack that doesn’t exist, a dialog box containing “No such stack” is displayed after the command is sent. The same command sent in a handler would put the error message into the result but would not display a dialog box. If you want a dialog box displayed, you could include an answer statement in your handler that displays the result—for example:

on mouseUp
    go stack quote& clickText &quote without dialog
    put the result into goError
    if goError is not empty
        then answer goError with "OK"
end mouseUp

Chapter 9 discusses the return statement. Appendix A contains information about external commands.
CHAPTER 11
Functions

Round

SYNTAX

the round of factor
round(expression)

Factor and expression yield numbers.

EXAMPLE

put round(resultVariable) into field 1

DESCRIPTION

The round function returns the source number rounded off to the nearest integer.

Any odd integer plus exactly 0.5 rounds up; any even integer (or 0) plus exactly 0.5 rounds down. If the source number is negative, HyperCard internally removes the negative sign, rounds its absolute value, then puts the negative sign back on.

SCRIPT

The following function handler rounds off an amount to the nearest dollar:

function roundToDollar amount
    set numberFormat to ".00" -- sets dollar format
    return round(amount)
end roundToDollar
CHAPTER 11

Functions

ScreenRect

SYNTAX

the screenRect
screenRect()

EXAMPLE

put the screenRect into menuLoc

DESCRIPTION

The screenRect function returns the rectangle of the screen in which HyperCard’s menu bar is displayed; the value returned is four integers, separated by commas, representing the pixel offsets of the left, top, right, and bottom edges, respectively, from the top-left corner of the screen.

Seconds

SYNTAX

the seconds
seconds()

EXAMPLE

put (the seconds-startTime) into runTime

DESCRIPTION

The seconds function returns an integer showing the number of seconds between midnight, January 1, 1904, and the current time set in your Macintosh. The seconds function can be abbreviated secs.
CHAPTER 11

Functions

SCRIPT

The following example handler counts the number of seconds the user holds down the mouse button:

```script
on mouseDown
    put the long time into now -- what time is it now?
    convert now to seconds
    wait while the mouse is down
    -- wait until mouse is released
    put the seconds-now into msg
    -- how many seconds have elapsed?
end mouseDown
```

NOTE

See also the convert command in Chapter 10, “Commands.”

SelectedButton

SYNTAX

the selectedButton of family

```script
selectedButton(family)
```

Family specifies a card or background button family.

EXAMPLES

```script
put the selectedButton of card family 1
get the selectedButton of family 4 of card 4
```
CHAPTER 11

Functions

DESCRIPTION

The selectedButton function returns the descriptor of the currently highlighted button in the specified card or background button family, such as card button 3. If no button in the family is highlighted, selectedButton returns empty.

NOTE

To change the value returned by selectedButton for a family, set the hilite property of one of the buttons. The select command has a different effect: select button 1 chooses the Button tool and selects the specified button for editing.

SelectedChunk

SYNTAX

the selectedChunk
selectedChunk()

EXAMPLES

get selectedChunk()
put the selectedChunk

DESCRIPTION

The selectedChunk function returns a chunk expression describing the location of the selected text or the insertion point in a card or background field. For selected text, it is in the form of first character selected to last character selected of the field containing the selected text: for example, char 7 to 15 of card field 3.
CHAPTER 11

Functions

If no text is selected but the insertion point is in a field, selectedChunk returns the number of the character on either side of the insertion point in reverse order with the highest number first. For example, if the insertion point is located between characters 9 and 10 in card field 2, selectedChunk returns char 10 to 9 of card field 2. If nothing is selected and the insertion point is not in a field, it returns empty.

NOTES

If you use the selectedChunk() form, be sure to leave the parentheses empty or you will get an error message.

Changing the highlight state of a button (clicking a button with autoHilite set to true) deselects the text and causes the selectedChunk function to return empty. Many other actions, such as clicking the card, clicking a locked field, or running an idle message handler that periodically changes some part of the display, also deselect the text, so you should put the result of selectedChunk into a container before any other action takes place.

You can put the selectedChunk result into the Message box or other container from a script, but you get an empty result if you make a selectedChunk function call from the Message box. This is the correct result of the call, since typing in the Message box and pressing the Return key or Enter key deselects any selected text.

You can get and set the text properties textFont, textSize, and textStyle of the selectedChunk function. See Chapter 12, “Properties,” for more information about the text properties.

SelectedField

SYNTAX

the selectedField

selectedField()
CHAPTER 11

Functions

EXAMPLES

get selectedField()
put the selectedField

DESCRIPTION

The selectedField function returns the descriptor of the field containing the selected text or the insertion point. If the selection or insertion point is in the Message box, selectedField returns the string message box.

NOTES

Changing the highlight state of a button (clicking a button with autoHilite set to true) deselects the text and causes the selectedField function to return empty. Many other actions, such as clicking the card or clicking a locked field, also deselect the text, so you should put the result of selectedField into a container before any other action takes place.

You can put the selectedField result into the Message box or other container from a script, but you get an empty result if you make a selectedField function call from the Message box. This is the correct result of the call, since typing in the Message box and pressing the Return key or Enter key deselects any selected field.

The selectedField function does not apply to list fields.

SelectedLine

SYNTAX

the selectedLine [of button | field]
selectedLine([button | field])

Button is a pop-up style button descriptor, and field is a field descriptor.
CHAPTER 11
Functions

EXAMPLES

get selectedLine()
put the selectedLine of card field 3
put word 2 of (selectedLine(btn 1))

DESCRIPTION

For pop-up buttons, the selectedLine function returns a chunk expression describing the currently selected line of the button’s contents as a string in the form

line number of card|bkgnd button number

For fields specified explicitly, the selectedLine function returns a chunk expression describing the line containing the selected text or insertion point in the form

line number to number of card|bkgnd field number

If no field or button is specified, the selectedLine function returns a chunk expression describing the line of a field or the Message box that contains the current selection or insertion point. The expression is of the form

line number of card|bkgnd field number

NOTES

Lines in fields are defined as ending with the return character, and because text can wrap in fields, there may be several lines of text on the screen between return characters. For example, line 8 always specifies the text between the seventh and eighth return characters.

The selectedLine function can also be spelled selectedLines.

For fields other than list fields, many actions, such as clicking a button or other field, deselect the text, so you should put the result of selectedLine into a container before any other action takes place. In contrast, list fields retain their selectedLine value until the user changes it specifically.
CHAPTER 11

Functions

If no lines are selected in an explicitly specified field, selectedLine returns empty. If no field or button is specified with the function call, and there is no current selection or insertion point, selectedLine returns empty.

SCRIPT

The following handler belongs in a list field (autoHilite, lockText, and dontWrap are true; multipleLines in this field should be false). Each line in the list field contains a glossary term. Each term corresponds to a card in the glossary stack, and the terms can be located by card number in the stack (that is, the term in line 2 of the field corresponds to card 2 of the glossary stack).

on mouseUp
  put the selectedLine of me into selLine
  put word 2 of selLine into whichEntry
  select line 0 of me -- deselect
  go card whichEntry of stack "Glossary" in a new window
end mouseUp

SelectedLoc

SYNTAX

the selectedLoc

selectedLoc()

EXAMPLES

get selectedLoc()
put the selectedLoc
CHAPTER 11

Functions

DESCRIPTION

The selectedLoc function returns a point at which the selected text begins. The selectedLoc function returns the point as two comma-separated integers that represent the offset from the top-left corner of the card to the bottom (baseline) left of the selected text.

The value returned by the selectedLoc function is the same as the value returned by the TEGetPoint routine, which is described in Inside Macintosh: Text.

SelectedText

SYNTAX

the selectedText [of button | field]
selectedText ([button | field])

Button is a pop-up style button descriptor, and field is a field descriptor.

EXAMPLES

get selectedText(card field 1)
put the selectedText
get the selectedText of button "My Pop-Up Button"

DESCRIPTION

The selectedText function returns the currently selected text in a field or pop-up button specified with the function call. If no field or button is specified, selectedText returns the text of the current selection in a nonlist field or the Message box. If there is no text selected, selectedText returns empty.
NOTES

When no field or button is specified, `selectedText` returns the same result as the selection or the value of the `selectedChunk`.

For fields other than list fields, many actions, such as clicking a button or other field, deselect the text, so you should put the result of `selectedLine` into a container before any other action takes place. In contrast, list fields retain their `selectedLine` value until the user changes it specifically.

You can put the `selectedText` result into the Message box or other container from a script, but you get an empty result if you make a `selectedText` function call from the Message box. This is the correct result of the call, since typing in the Message box and pressing the Return key or Enter key deselects any selected text.

**ShiftKey**

**SYNTAX**

```plaintext
the shiftKey
shiftKey()
```

**EXAMPLE**

```plaintext
if the shiftKey is down then put ¬
    numToChar(charToNum(msg)-32) into msg
```

**DESCRIPTION**

The `shiftKey` function returns the constant `up` if the Shift key is not pressed, `down` if it is pressed.
CHAPTER 11
Functions

SCRIPT

The following handlers in the script of a field enable the user to change uppercase characters to lowercase and vice versa. The user selects a chunk of text (not including any Return characters) and presses Enter. If the Shift key is down, the script changes the case of the characters.

on enterInField
    if the shiftKey is down then
        toggleCaps the selectedChunk, the selectedText
    end if
    pass enterInField
end enterInField

on toggleCaps theChunk, theText
    if theChunk is empty or ¬
        (word 2 of theChunk > word 4 of theChunk)
    then exit toggleCaps
    repeat with i = 1 to length(theText)
        put swapcase(char i of theText) into ¬
        char i of theText
    end repeat
    do "put" && quote & theText & quote ¬
        && "into" && theChunk
end toggleCaps

function swapcase theChar
    get charToNum(theChar)
    if it ≥ 65 and it ≤ 90 then return numToChar(it+32)
    if it ≥ 97 and it ≤ 122 then return numToChar(it-32)
    return theChar
end swapCase

NOTE

See also the commandKey and optionKey functions earlier in this chapter.
CHAPTER 11

Functions

Sin

SYNTAX

the sin of factor
sin(expression)

Factor and expression yield numbers.

EXAMPLE

put the sin of 2 -- puts 0.909297 into the Message box

DESCRIPTION

The sin function returns the sine of the angle that is passed to it. The angle must be expressed in radians.

NOTE

Radians can be converted to degrees by multiplying by 180 and dividing the result by the value of the constant pi.

Sound

SYNTAX

the sound
sound()

EXAMPLE

wait until the sound is "done"
CHAPTER 11

Functions

DESCRIPTION

The `sound` function returns the name of the sound resource currently playing (such as "boing") or the string "done" if no sound is currently playing. The `sound` function enables you to synchronize sounds with other actions, because in most cases scripts continue to run while sounds play. In the event of a low-memory situation, such as when playing a large sound while a large Home stack or several other stacks are in use and HyperCard is set to the default memory allocations, HyperCard may suspend other actions until the sound is finished playing. Increasing HyperCard’s memory allocation should alleviate this problem.

SCRIPT

The following example handler repeats a series of visual effects until a tune specified by the `play` command finishes:

```hypercard
on boogie
    play "harpsichord" tempo 200 ¬
    "ce gq fe ee de ce gq fe ee ce ce gq fe ee ce" 
    repeat until the sound is "done"
    visual effect dissolve to black
    visual effect zoom open to white
    visual effect barn door close to card
    go this card
end repeat
end boogie
```

NOTES

The "done" string is returned as a literal; it’s not a HyperTalk constant like `up` or `true`.

See also the `play` command in Chapter 10.
CHAPTER 11

Functions

Sqrt

SYNTAX

the sqrt of factor
sqrt(expression)

Factor and expression yield numbers.

EXAMPLE

put the sqrt of msg
-- converts the number in msg to its square root

DESCRIPTION

The sqrt function returns the square root of the positive number passed to it. If you pass a negative number, you get the result -NAN(001), which means "not a number."

Stacks

SYNTAX

the stacks
stacks()

EXAMPLE

put the stacks into card field 2

DESCRIPTION

The stacks function returns a return-delimited list of the pathnames of all the currently open stacks. The stack returned in the first line is the active stack.
CHAPTER 11

Functions

StackSpace

SYNTAX

the stackSpace
stackspace()

EXAMPLE

put the stackSpace into howMuch

DESCRIPTION

The stackSpace function returns an integer that represents the space remaining on the Macintosh operating system stack. Stack in this case refers to an internal data structure rather than a HyperCard stack.

Sum

SYNTAX

sum(list)

List is a sequence of comma-separated expressions that yield numbers, or it is a single container that contains such a sequence.

EXAMPLE

get sum (1,2,3,4)

DESCRIPTION

The sum function returns the sum of a list of numbers passed to it. If the source of the list is a container with more than one line in it, only the first line is used.
CHAPTER 11

Functions

SystemVersion

SYNTAX

the systemVersion
systemVersion()

EXAMPLE

put the systemVersion

DESCRIPTION

The systemVersion function returns a decimal string that represents the running version of system software.

You might use this function to determine if a particular HyperCard command or handler will run correctly under that version of the software.

SCRIPT

The following code makes sure that the machine that HyperCard is running on is using System 7; if not, HyperCard posts an appropriate message:

if the systemVersion < 7
then answer "This stack requires System 7." with "Quit"
doMenu "quit Hypercard"

Tan

SYNTAX

the tan of factor
tan(expression)

Factor and expression yield numbers.

SystemVersion

347
CHAPTER 11

Functions

EXAMPLE

put the tan of 2 -- puts -2.18504 into the Message box

DESCRIPTION

The \texttt{tan} function returns the tangent of the angle that is passed to it. The angle must be expressed in radians.

NOTE

Radians can be converted to degrees by multiplying by 180 and dividing the result by the value of the constant \texttt{pi}.

Target

SYNTAX

the target

EXAMPLE

if the target is "card id 2875" then pass mouseUp

DESCRIPTION

The \texttt{target} function returns a string indicating the original recipient of the message. The string returned is one of the following:

\begin{itemize}
  \item stack "name"
  \item bkgnd of card "name"|id number
  \item card "name"|id number
  \item bkgnd field "name"|id number
  \item card field "name"|id number
  \item bkgnd button "name"|id number
  \item card button "name"|id number
\end{itemize}
CHAPTER 11

Functions

For example, the target function enables you to tell, in a mouseUp handler in a background, whether

- the mouse was clicked over a field or button (which either would have had no mouseUp handler or would have passed the message on explicitly): the target would return the button or field name, if it has one, or ID number if not
- the mouse was clicked outside the area of all buttons and fields: the target would return the card name, if it has one, or ID number if not
- the message was sent directly to the background with the send command: the target would return the background name, if it has one, or ID number if not

You can use the target in place of an object descriptor to determine any of the target’s properties:

get the short name of the target

NOTES

The send command resets the value of the target to the value of the object the message is being sent to.

If the target is a button or field, the expression target (without the) evaluates to the contents of the button or field.

Ticks

SYNTAX

the ticks
ticks()

EXAMPLE

put the ticks into clock

Ticks
CHAPTER 11

Functions

DESCRIPTION

The ticks function returns an integer representing the number of ticks (1/60 second) since the Macintosh was turned on or restarted.

SCRIPT

The following example handler measures how long it takes to go to the Help stack and find the word ticks:

```script
on mouseUp
    put the ticks into startTicks
    go "HyperCard Help"
    find "ticks"
    put (the ticks - startTicks) into howLong
    answer "It took" && howLong div 60 && "second(s) to find Help."
end mouseUp
```

SYNTAX

the [adjective] time
time()

*Adjective* can be long, short, or abbreviated (or abbrev, or abbr).

EXAMPLE

put the time into the Message box
The `time` function returns the time as a text string. The short and abbreviated forms are the same, returning the hour and minutes, such as 8:55 AM. The long time form returns seconds as well, such as 8:55:23 AM.

The following example records the time at which a field is updated:

```plaintext
on closeField
    put return & the time after card field "updateList"
end closeField
```

An adjective can’t be used to modify the form of the `time` function that uses parentheses.

The time string returned by the `time` function can be in either 24- or 12-hour format depending on the time format set in the Date and Time control panel. The time format can also be altered in the international resources of the System file. If you are going to perform calculations on the time, you should first convert it to the invariant seconds format. See the `convert` command in Chapter 10.
CHAPTER 11
Functions

DESCRIPTION
The tool function returns the name of the currently chosen tool. Possible values returned by the tool function are

- browse tool
- brush tool
- bucket tool
- button tool
- curve tool
- eraser tool
- field tool
- lasso tool
- line tool
- oval tool
- pencil tool
- polygon tool
- rectangle tool
- regular polygon tool
- round rect tool
- select tool
- spray tool
- text tool

SCRIPT
The following example handler chooses the proper tool to manipulate a button or field when you move the pointer over either object:

```
on mouseWithin -- in card, background, or stack script
  if "button" is in the target and the optionKey is down
    then choose button tool
  else if "field" is in the target and ¬
    the optionKey is down
    then choose field tool
end mouseWithin
```

NOTE
See also the choose command in Chapter 10.
CHAPTER 11

Functions

Trunc

SYNTAX

the trunc of factor
trunc(expression)

Factor and expression yield numbers.

EXAMPLE

put the trunc of someNumber into msg

DESCRIPTION

The trunc function returns the integer part of the number passed to it. Any fractional part is disregarded, regardless of sign.

SCRIPT

The following example handler draws rectangles in increasing sizes, using the trunc function to ensure that the computed values used with the drag command are integers:

on mouseUp
    push card
doMenu "New Card"
    reset paint
    choose rectangle tool
    put 50 into left
    put 150 into right
    put 50 into top
    put 150 into bottom
CHAPTER 11

Functions

repeat 5 -- the drag command only takes integers
   drag from left,top to right,bottom
   put trunc(left/1.2) into left
   put trunc(right/1.2) into right
   put trunc(top/1.2) into top
   put trunc(bottom/1.2) into bottom
end repeat
choose browse tool
end mouseUp

Value

SYNTAX
the value of factor
value(expression)

Factor and expression yield any values.

EXAMPLE
put the value of field "formula" into field "result"

DESCRIPTION
The value function evaluates the string derived from factor or expression as an expression. Note that multitoken literal expressions evaluate to themselves:

put value ("HyperCard 2.2") -- returns HyperCard 2.2
CHAPTER 11

Functions

SCRIPT

The following example handler demonstrates the value function by forcing a second level of evaluation of a variable:

on mouseUp
    put "3 + 4" into expression
    put expression -- yields "3 + 4"
    wait 2 seconds
    put value of expression -- yields 7
end mouseUp

Windows

SYNTAX

the windows
windows()

EXAMPLES

put the windows into it -- puts the windows value
-- into the variable It
windows()
put the windows into card field 1

DESCRIPTION

The windows function returns a return-delimited list containing the names of all of the windows currently available to HyperCard and the current stack. The order of the window names corresponds to the front-to-back ordering of the windows. The windows include built-in palettes (Tool and Pattern), FatBits, the Message box, open card windows, and external windows (for example, Message Watcher, Navigator palette, Scroll window, Variable Watcher).

NOTE

The windows returned in the list may not currently be visible.
This chapter describes HyperCard properties. Properties are the defining characteristics of objects, other elements such as menus and windows, and the HyperCard environment.

Object properties determine how objects look and act. Global properties control aspects of the overall HyperCard environment. Painting properties control aspects of the HyperCard painting environment, which is invoked when you choose a Paint tool. Window properties determine how card windows, the Message box, the Tools and Patterns palettes, and external windows are displayed. Menu, menu bar, and menu item properties control aspects of HyperCard menus. There are also a few properties that apply to the Message Watcher or the Variable Watcher.

This chapter includes a set of tables that list the HyperCard properties by category. Button properties, field properties, window properties, and so on are each listed in a separate table. The tables are followed by complete descriptions of all the properties in alphabetical order.

Retrieving and Setting Properties

HyperTalk lets you retrieve most properties by using the property name as a function in a script or the Message box. You must precede the property name with the word the or follow it with of if it’s an object or window property. You can’t use parentheses after the property name, as you do with built-in functions. The following example retrieves the location property of button 1 and puts it into the Message box:

```
put the loc of button 1 into msg
```
You set properties with the `set` command:

```
set loc of button 1 to 100,100
```

Some properties can’t be set, although other actions affect them. For example, the size of a stack is read-only, but it can be changed by compacting it and by adding objects.

You can get the value of most properties with the `get` command:

```
get the property of object
```

Most of the examples in this chapter do not include the syntax for the `get property` command, because it is faster to put a property directly into a container or variable (rather than using the `get` command and then the `put` command).

---

**Object Properties**

You can see the value of many object properties by looking at an object’s Info dialog box, an example of which is shown in Figure 12-1. (You bring up an object’s Info dialog box by choosing the appropriate command from the Objects menu.)

---

![Figure 12-1](An object's Info dialog box)
You can also set many properties for the current object from the Info dialog boxes. To set the properties of any object in the current stack, you use the `set` command, either in a script or in the Message box.

Different HyperCard objects have different properties. For example, fields have a property determining their text style, but cards do not.

**Stack Properties**

Stack properties pertain to any stack on any disk or file server currently accessible to your Macintosh. A stack is specified as explained in Chapter 5, “Referring to Objects, Menus, and Windows.” Settable properties of the current stack can be manipulated from a script or through the Stack Info dialog box invoked from the Objects menu.

The stack properties are listed in Table 12-1. More detailed information about each property is given later in this chapter.

**Table 12-1 Stack properties**

<table>
<thead>
<tr>
<th>Stack property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cantAbort</code></td>
<td>Controls whether or not the user can use Command-period to stop execution of scripts.</td>
</tr>
<tr>
<td><code>cantDelete</code></td>
<td>Controls whether or not the user can delete the specified stack.</td>
</tr>
<tr>
<td><code>cantModify</code></td>
<td>Controls whether or not the stack can be changed in any way.</td>
</tr>
<tr>
<td><code>cantPeek</code></td>
<td>Controls whether or not the user can look at button or field scripts with Command-Option.</td>
</tr>
<tr>
<td><code>freeSize</code></td>
<td>Determines the amount of free space of the specified stack in bytes.</td>
</tr>
<tr>
<td><code>name</code></td>
<td>Determines or changes the name of the specified stack, which is its Macintosh filename.</td>
</tr>
<tr>
<td><code>reportTemplates</code></td>
<td>Determines the report-printing templates of the stack.</td>
</tr>
<tr>
<td><code>script</code></td>
<td>Retrieves or replaces the script of the specified stack.</td>
</tr>
</tbody>
</table>

*continued*
Background Properties

Background properties pertain to any background in the current stack. The background is specified as explained in Chapter 5, “Referring to Objects, Menus, and Windows.” Background properties can be manipulated from a script or from the Message box. Properties of the current background can also be manipulated through the Bkgnd Info dialog box invoked from the Objects menu, or in the script editor window in the case of scriptingLanguage.

The background properties are listed in Table 12-2. More detailed information about each property is given later in this chapter.

Table 12-2  Background properties

<table>
<thead>
<tr>
<th>Background property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cantDelete</td>
<td>Controls whether or not the user can delete the specified background.</td>
</tr>
<tr>
<td>dontSearch</td>
<td>Determines whether or not the fields in a specified background are searched with the find command.</td>
</tr>
<tr>
<td>ID</td>
<td>Determines the permanent ID number of any background in the current stack.</td>
</tr>
<tr>
<td>name</td>
<td>Determines or changes the name of the specified background.</td>
</tr>
<tr>
<td>number</td>
<td>Determines the number of any background in the current stack.</td>
</tr>
</tbody>
</table>

continued
Card properties pertain to any card in the current stack. The card is specified as explained in Chapter 5, “Referring to Objects, Menus, and Windows.” You can manipulate card properties from a script, in the Message box, or through the Card Info dialog box invoked from the Objects menu, or in the script editor window in the case of scriptingLanguage.

The card properties are listed in Table 12-3. More detailed information about each property is given later in this chapter.

**Table 12-3 Card properties**

<table>
<thead>
<tr>
<th>Card property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cantDelete</td>
<td>Controls whether or not the user can delete the specified card.</td>
</tr>
<tr>
<td>dontSearch</td>
<td>Determines whether or not the fields in a specified card are searched with the find command.</td>
</tr>
<tr>
<td>ID</td>
<td>Determines the permanent ID number of any card in the current stack.</td>
</tr>
<tr>
<td>marked</td>
<td>Determines or changes whether a specified card is marked.</td>
</tr>
<tr>
<td>name</td>
<td>Determines or changes the name of the specified card.</td>
</tr>
</tbody>
</table>

*continued*
Field Properties

Field properties pertain to any card field or background field in the current stack. The field is specified as explained in Chapter 5, “Referring to Objects, Menus, and Windows.” You can manipulate field properties from a script or from the Message box, or through the Field Info dialog box invoked from the Objects menu, or in the script editor window in the case of scriptingLanguage. (You must have the Field tool chosen and a specific card or background field selected or have the insertion point in a field to activate the Field Info dialog box.)

The field properties are listed in Table 12-4. More detailed information about each property is given later in this chapter.
### Table 12-4  Field properties

<table>
<thead>
<tr>
<th>Field property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>autoSelect</td>
<td>Enables a field to behave as a list when its dontWrap and lockText property are also true.</td>
</tr>
<tr>
<td>autoTab</td>
<td>Determines or changes whether or not the specified nonscrolling field sends the tabKey message to the current card.</td>
</tr>
<tr>
<td>dontSearch</td>
<td>Determines or changes whether or not the specified field is searched with the find command.</td>
</tr>
<tr>
<td>dontWrap</td>
<td>Determines or changes whether or not text at the edge of the specified field automatically wraps around to the next line.</td>
</tr>
<tr>
<td>fixedLineHeight</td>
<td>Determines or changes whether or not the lines in the specified field have a fixed line height.</td>
</tr>
<tr>
<td>ID</td>
<td>Determines the permanent ID number of the specified field.</td>
</tr>
<tr>
<td>location</td>
<td>Determines or changes the location of the specified field.</td>
</tr>
<tr>
<td>lockText</td>
<td>Determines or changes whether editing of text within the specified field in the current stack is allowed or prevented.</td>
</tr>
<tr>
<td>multipleLines</td>
<td>Determines or changes whether or not the user can select multiple lines in a list field.</td>
</tr>
<tr>
<td>name</td>
<td>Determines or changes the name of the specified field.</td>
</tr>
<tr>
<td>number</td>
<td>Determines the number of the specified field.</td>
</tr>
<tr>
<td>partNumber</td>
<td>Determines or changes the number that represents the ordering of the buttons and fields within their enclosing card or background. Setting this property can have the effect of either bringing the field closer or moving it farther (behind) other buttons and fields.</td>
</tr>
</tbody>
</table>

*continued*
Table 12-4  Field properties (continued)

<table>
<thead>
<tr>
<th>Field property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rect[angle]</td>
<td>Determines or changes the location and size of the rectangle occupied by the specified field. See also Table 12-6.</td>
</tr>
<tr>
<td>script</td>
<td>Retrieves or replaces the script of the specified field.</td>
</tr>
<tr>
<td>scriptingLanguage</td>
<td>Determines or changes the scripting language of the specified field.</td>
</tr>
<tr>
<td>scroll</td>
<td>Determines or changes how much material is hidden above the top of the specified scrolling field’s rectangle.</td>
</tr>
<tr>
<td>sharedText</td>
<td>Determines or changes whether the text in the specified background field appears on each card of that background.</td>
</tr>
<tr>
<td>showLines</td>
<td>Determines or changes whether the text baselines in the specified field appear or are invisible.</td>
</tr>
<tr>
<td>style</td>
<td>Determines or changes the style of the specified field in the current stack.</td>
</tr>
<tr>
<td>textAlign</td>
<td>Determines or changes the way lines of text are aligned in the specified field.</td>
</tr>
<tr>
<td>textFont</td>
<td>Determines or changes the font in which text in the specified field appears.</td>
</tr>
<tr>
<td>textHeight</td>
<td>Determines or changes the space between baselines of text in the specified field.</td>
</tr>
<tr>
<td>textSize</td>
<td>Determines or changes the type size in which text in the specified field appears.</td>
</tr>
<tr>
<td>textStyle</td>
<td>Determines or changes the style in which text in the specified field appears.</td>
</tr>
<tr>
<td>visible</td>
<td>Determines or changes whether the specified field is shown or hidden. The default value of the visible property is true.</td>
</tr>
<tr>
<td>wideMargins</td>
<td>Determines or changes whether some extra space is included at the left and right sides of each line in the specified field (to make the text easier to read).</td>
</tr>
</tbody>
</table>
CHAPTER 12

Properties

Button Properties

Button properties pertain to any card button or background button in the current stack. The button is specified as explained in Chapter 5, “Referring to Objects, Menus, and Windows.”

You can manipulate the properties of any button in the current stack from a script or from the Message box. Additionally, you can manipulate the properties of a button on the current card or background through the Button Info dialog box invoked from the Objects menu, or in the script editor window in the case of scriptingLanguage. (You must have the Button tool and a specific card or background button selected to activate the Button Info dialog box.)

The button properties are listed in Table 12-5. More detailed information about each property is given later in this chapter.

Table 12-5  Button properties

<table>
<thead>
<tr>
<th>Button property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>autoHilite</td>
<td>Determines or changes whether the specified button highlights when that button is pressed.</td>
</tr>
<tr>
<td>enabled</td>
<td>Determines or changes whether the specified button appears and behaves in an enabled or disabled state.</td>
</tr>
<tr>
<td>family</td>
<td>Groups two or more buttons together into a family specified by the numbers 1 to 15, inclusive.</td>
</tr>
<tr>
<td>hilite</td>
<td>Determines or changes whether the specified button is highlighted (displayed in inverse video).</td>
</tr>
<tr>
<td>icon</td>
<td>Determines or changes the icon for the specified button in the current stack.</td>
</tr>
<tr>
<td>ID</td>
<td>Determines the permanent ID number of the specified button.</td>
</tr>
<tr>
<td>loc[ation]</td>
<td>Determines or changes the location of the specified button.</td>
</tr>
<tr>
<td>name</td>
<td>Determines or changes the name of the specified button.</td>
</tr>
</tbody>
</table>

continued
### Table 12-5 Button properties (continued)

<table>
<thead>
<tr>
<th>Button property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>Determines the number of the specified button.</td>
</tr>
<tr>
<td>partNumber</td>
<td>Determines or changes the number that represents the ordering of the buttons and fields within their enclosing card or background.</td>
</tr>
<tr>
<td>rect[angle]</td>
<td>Determines or changes the location and size of the rectangle occupied by the specified button. See also Table 12-6.</td>
</tr>
<tr>
<td>script</td>
<td>Retrieves or replaces the script of the specified button.</td>
</tr>
<tr>
<td>scriptingLanguage</td>
<td>Determines or changes the scripting language of the specified field.</td>
</tr>
<tr>
<td>sharedHilite</td>
<td>Determines or changes whether the specified background button is displayed highlighted on all cards of that background.</td>
</tr>
<tr>
<td>showName</td>
<td>Determines or changes whether the name of the specified button (if it has one) is displayed in its rectangle on the screen.</td>
</tr>
<tr>
<td>style</td>
<td>Determines or changes the style of the specified button in the current stack.</td>
</tr>
<tr>
<td>textAlign</td>
<td>Determines or changes the way lines of text are aligned in the specified button.</td>
</tr>
<tr>
<td>textFont</td>
<td>Determines or changes the font in which text in the specified button appears.</td>
</tr>
<tr>
<td>textHeight</td>
<td>Determines or changes the space between baselines of text in the specified button.</td>
</tr>
<tr>
<td>textSize</td>
<td>Determines or changes the type size in which text in the specified button appears.</td>
</tr>
<tr>
<td>textStyle</td>
<td>Determines or changes the style in which text in the specified button appears.</td>
</tr>
</tbody>
</table>

*continued*
CHAPTER 12

Properties

Table 12-5  Button properties (continued)

<table>
<thead>
<tr>
<th>Button property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>titleWidth</td>
<td>Determines or changes the width of the area in which a pop-up button’s name appears.</td>
</tr>
<tr>
<td>visible</td>
<td>Determines or changes whether the specified button is shown or hidden. The default value of the visible property is true.</td>
</tr>
</tbody>
</table>

Rectangle Properties

The properties described in this section pertain to the screen rectangles of the menu bar, buttons and fields, the Tools and Patterns palettes (called windows in scripts), the Scroll window, the FatBits window, card windows, stack windows, the Message box, and external windows (if the XCMD that created the window supports control of rectangle properties).

The rectangle property itself also applies to buttons, fields, windows, and the menu bar. The rectangle property is two points, the top-left and bottom-right corners of an object’s rectangle. The points are represented as four integers separated by commas: left (item 1), top (item 2), right (item 3), bottom (item 4). The rectangle properties affect these four items, one at a time or two at a time.

The rectangle properties are listed in Table 12-6. More detailed information about each property is given later in this chapter.

Table 12-6  Rectangle properties

<table>
<thead>
<tr>
<th>Rectangle property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bottom</td>
<td>Determines or changes the value of item 4 of the rectangle property when applied to the specified object or window.</td>
</tr>
<tr>
<td>bottomRight</td>
<td>Determines or changes items 3 and 4 of the value of the rectangle property when applied to the specified object or window.</td>
</tr>
</tbody>
</table>

continued
Environmental Properties

Some of the global properties, such as the `userLevel` property, can be set on the User Preferences card of the Home stack; others, such as the `lockMessages` property, can be retrieved and set only through HyperTalk. (However, the User Preferences card uses HyperTalk to set properties, and it could be extended to set any of the others.) The window properties, which pertain to the Message box and the tear-off palettes, can be set by clicking and dragging the windows themselves, as well as through HyperTalk. Painting properties, which pertain to the painting environment, can be controlled with the menus and palettes that appear when a Paint tool is selected, as well as through HyperTalk.
You use global properties to choose how particular aspects of the HyperCard environment perform. You set global properties from any script or from the Message box, and their settings pertain to all objects—if you set `userLevel` to 3, for example, it remains 3 until you reset it (although a protected stack might impose some other user level while you are in that stack).

All of the printing properties, such as `printMargins`, can be restored simultaneously to their default values with the `reset printing` command, described in Chapter 10, “Commands.”

The global properties are listed in Table 12-7. More detailed information about each property is given later in this chapter.

<table>
<thead>
<tr>
<th>Global property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>address</code></td>
<td>Determines where your HyperCard application is running. If you’re connected to a network, this property returns a string in the form “zone:Macintosh:program.” If you’re not on a network, or on a network with just one AppleTalk zone, HyperCard substitutes an asterisk (*) for the network name.</td>
</tr>
<tr>
<td><code>blindTyping</code></td>
<td>Determines or changes whether you can type messages into the Message box and send them (execute them) without having the Message box visible.</td>
</tr>
<tr>
<td><code>cursor</code></td>
<td>Changes the image that appears at the pointer location on the screen.</td>
</tr>
<tr>
<td><code>debugger</code></td>
<td>Determines or changes the name of the current HyperTalk debugger. Custom debuggers can be created as XCMDs and called with HyperTalk.</td>
</tr>
<tr>
<td><code>dialingTime</code></td>
<td>Determines how long HyperCard waits before closing the serial connection to a modem after dialing. Time units for this property are in ticks (1/60th of a second) with the default time set to 180 ticks (3 seconds).</td>
</tr>
</tbody>
</table>

*continued*
### Chapter 12

Properties

<table>
<thead>
<tr>
<th>Global property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dialingVolume</td>
<td>Determines or changes the volume of the touch tones generated through the Macintosh speaker by the dial command.</td>
</tr>
<tr>
<td>dragSpeed</td>
<td>Determines or changes how many pixels per second the pointer moves when manipulated by all subsequent drag commands.</td>
</tr>
<tr>
<td>editBkgnd</td>
<td>Determines or changes where any painting or creating of buttons or fields happens—on the current card or on its background.</td>
</tr>
<tr>
<td>environment</td>
<td>Determines the environment of the currently running HyperCard application; returns development if it is the fully enabled development version, and player if the HyperCard Player is running.</td>
</tr>
<tr>
<td>ID</td>
<td>Determines the permanent signature of HyperCard, 'WILD'.</td>
</tr>
<tr>
<td>itemDelimiter</td>
<td>Determines what delimiter is used to separate a list of items. HyperCard resets the delimiter to its default, the comma, when the computer is idle.</td>
</tr>
<tr>
<td>language</td>
<td>Determines or changes the language dialect in which HyperTalk scripts are written and displayed.</td>
</tr>
<tr>
<td>lockErrorDialogs</td>
<td>Determines or changes whether HyperCard displays an error dialog. This property is set to false at idle time, so it has no effect if you enter it through the Message box.</td>
</tr>
<tr>
<td>lockMessages</td>
<td>Determines or changes whether HyperCard sends system messages such as openCard, closeCard, and so on.</td>
</tr>
<tr>
<td>lockRecent</td>
<td>Determines or changes whether HyperCard adds miniature representations to the Recent card.</td>
</tr>
<tr>
<td>lockScreen</td>
<td>Determines or changes whether HyperCard updates the screen when you go to another card.</td>
</tr>
</tbody>
</table>

*continued*
Retrieving and Setting Properties

### Table 12-7  Global properties (continued)

<table>
<thead>
<tr>
<th>Global property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>longWindowTitles</td>
<td>Determines or changes whether HyperCard displays the long name of a stack in its title bar.</td>
</tr>
<tr>
<td>messageWatcher</td>
<td>Determines or changes the name of the current message watcher. A custom message watcher can be created as an XCMD and called with HyperTalk.</td>
</tr>
<tr>
<td>numberFormat</td>
<td>Determines or changes the precision with which the results of mathematical operations are displayed in fields and the Message box.</td>
</tr>
<tr>
<td>powerKeys</td>
<td>Determines or changes whether keyboard shortcuts for painting actions are available.</td>
</tr>
<tr>
<td>printMargins</td>
<td>Specifies the page margins to use when printing reports and expressions.</td>
</tr>
<tr>
<td>printTextAlign</td>
<td>Specifies the text alignment to use in the header of a print report and when printing variables.</td>
</tr>
<tr>
<td>printTextFont</td>
<td>Specifies the text font to use in the header of a print report and when printing variables.</td>
</tr>
<tr>
<td>printTextHeight</td>
<td>Specifies the line height to use in the header of a print report and when printing variables.</td>
</tr>
<tr>
<td>printTextSize</td>
<td>Specifies the text size to use in the header of a print report and when printing variables.</td>
</tr>
<tr>
<td>printTextStyle</td>
<td>Specifies the text style to use in the header of a print report and when printing variables.</td>
</tr>
<tr>
<td>scriptEditor</td>
<td>Determines or changes the current script editor to use. A custom script editor can be created as an XCMD and called with HyperTalk.</td>
</tr>
<tr>
<td>scriptingLanguage</td>
<td>Determines or changes the scripting language of the Message box.</td>
</tr>
<tr>
<td>scriptTextFont</td>
<td>Determines or changes which font to use in the script editor.</td>
</tr>
<tr>
<td>scriptTextSize</td>
<td>Determines or changes which font size to use in the script editor.</td>
</tr>
</tbody>
</table>

---

Continued
Painting properties are aspects of the painting environment invoked when you choose a Paint tool from the Tools palette. Most of these properties are usually manipulated from the Options and Patterns menus that appear when a Paint tool is selected. The text attributes pertain to Paint text; they are usually manipulated from the dialog box that appears when you double-click the Paint Text tool in the Tools palette or when you choose Text Style from the Edit menu. Changes to the settings made from HyperTalk are reflected on their respective palettes and menus.

All of the painting properties can be restored to their default values simultaneously with the `reset paint` command, described in Chapter 10, “Commands.”

<table>
<thead>
<tr>
<th>Global property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stacksInUse</td>
<td>Determines which stacks are currently in the message-passing hierarchy.</td>
</tr>
<tr>
<td>suspended</td>
<td>Determines whether HyperCard is currently running in the background under MultiFinder or System 7.</td>
</tr>
<tr>
<td>textArrows</td>
<td>Alters the function of the Right Arrow, Left Arrow, Up Arrow, and Down Arrow keys.</td>
</tr>
<tr>
<td>traceDelay</td>
<td>Determines the time between execution of statements while tracing.</td>
</tr>
<tr>
<td>userLevel</td>
<td>Sets or retrieves the value of the current HyperCard user level.</td>
</tr>
<tr>
<td>userModify</td>
<td>Determines or changes whether or not a user can type into fields or use Paint tools in a stack that has been write-protected.</td>
</tr>
<tr>
<td>variableWatcher</td>
<td>Determines or changes the name of the current variable watcher. Custom variable watchers can be created as XCMDs and called with HyperTalk.</td>
</tr>
<tr>
<td>version</td>
<td>Determines the version number of the HyperCard application currently running or the versions of HyperCard that created and modified a specified stack.</td>
</tr>
</tbody>
</table>
The painting properties are listed in Table 12-8. More detailed information about each property is given later in this chapter.

<table>
<thead>
<tr>
<th>Painting property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>brush</td>
<td>Determines or changes the current brush shape used by the Brush tool.</td>
</tr>
<tr>
<td>centered</td>
<td>Determines or changes the Draw Centered setting.</td>
</tr>
<tr>
<td>filled</td>
<td>Determines or changes the Draw Filled setting.</td>
</tr>
<tr>
<td>grid</td>
<td>Determines or changes the painting Grid setting.</td>
</tr>
<tr>
<td>lineSize</td>
<td>Determines or changes the thickness of the lines drawn by the line and shape tools.</td>
</tr>
<tr>
<td>multiple</td>
<td>Determines or changes the Draw Multiple setting.</td>
</tr>
<tr>
<td>multiSpace</td>
<td>Determines or changes the amount of space left between edges of the multiple images drawn by the shape tools when the multiple property is true.</td>
</tr>
<tr>
<td>pattern</td>
<td>Determines or changes the current pattern used to fill shapes and to paint with the Brush tool.</td>
</tr>
<tr>
<td>polySides</td>
<td>Determines or changes the number of sides of the polygon created by the Regular Polygon tool.</td>
</tr>
<tr>
<td>textAlign</td>
<td>Determines or changes the way characters are aligned around the insertion point as you type them with the Paint Text tool.</td>
</tr>
<tr>
<td>textFont</td>
<td>Determine or changes the font in which Paint text appears.</td>
</tr>
<tr>
<td>textHeight</td>
<td>Determines or changes the space between baselines of Paint text.</td>
</tr>
<tr>
<td>textSize</td>
<td>Determines or changes the font size in which Paint text appears on the screen.</td>
</tr>
<tr>
<td>textStyle</td>
<td>Determines or changes the style in which Paint text appears.</td>
</tr>
</tbody>
</table>
Window Properties

Window properties let you find out about and change the way that the Message box, Tools palette, Patterns palette, card window, Scroll window, Navigator window, Message Watcher window, Variable Watcher window, stack window, and external windows are displayed. The window names these properties apply to are

card window
message [box]
message [window]
msg
pattern window
tool window
window "Message Watcher"

Message can be abbreviated msg.

The properties that apply only to the Message Watcher and Variable Watcher are listed in Table 12-11.

The window properties are listed in Table 12-9. If the property only applies to a specific window, it is called out in the description column. More detailed information about each property is given later in this chapter.

<table>
<thead>
<tr>
<th>Window property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>Determines the permanent ID number of a window in the current stack.</td>
</tr>
<tr>
<td>loc[ation]</td>
<td>Determines or changes the location at which the window is displayed.</td>
</tr>
<tr>
<td>name</td>
<td>Determines the name of the specified window.</td>
</tr>
<tr>
<td>number</td>
<td>Determines the ordinal position in the window layers of any window on your screen.</td>
</tr>
</tbody>
</table>

continued
CHAPTER 12

Properties

Table 12-9 Window properties (continued)

<table>
<thead>
<tr>
<th>Window property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner of \textit{window}</td>
<td>Returns the name of the entity that created the window; this could be HyperCard or the name of an XCMD like Picture, etc. \textit{Window} is an expression yielding a valid window identifier including either the name, ID, or layer number of the window.</td>
</tr>
<tr>
<td>\textit{rect[angle]}</td>
<td>Determines or changes the size of card and stack window rectangles. See also Table 12-6.</td>
</tr>
<tr>
<td>\textit{scroll}</td>
<td>Determines or changes the scroll of the specified card picture in the card window or picture in the specified picture window.</td>
</tr>
<tr>
<td>visible</td>
<td>Determines or changes whether a window is shown or hidden on the screen.</td>
</tr>
<tr>
<td>zoomed</td>
<td>Determines or changes whether a window is set to its maximum size and centered on the screen, as when the user clicks its zoom box.</td>
</tr>
</tbody>
</table>

Menu, Menu Bar, and Menu Item Properties

The menu item properties described in this section pertain to any specified menu item created with the \texttt{put} command. You can manipulate the properties of menu items from a script or from the Message box.

You can use the \texttt{delete}, \texttt{disable}, and \texttt{enable} commands to delete, disable, or enable the Tools, Patterns, Font, and Apple menus, but you cannot alter the contents of those menus with any other HyperTalk commands. You can also enable and disable those menus with the \texttt{enabled} property. However, you cannot manipulate the menu items of these menus.

Since there is only one menu bar per computer screen, this HyperCard object does not follow the HyperCard object identifier convention where objects can be specified by name, number, and ID. However, HyperTalk now supports both the \texttt{visible} and \texttt{rectangle} properties for the menu bar. Of these properties, \texttt{visible} is the only one that is modifiable. The \texttt{rectangle} properties are useful only for determining the size of the menu bar.
The menu, menu bar, and menu item properties are listed in Table 12-10. More detailed information about each property is given later in this chapter.

Table 12-10  Menu, menu bar, and menu item properties

<table>
<thead>
<tr>
<th>Menu property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>checkMark</td>
<td>Determines or changes the current checked value of a specified menu item; a Boolean value.</td>
</tr>
<tr>
<td>commandChar</td>
<td>Determines or changes the current character to be used with the Command key as a keyboard shortcut for a specified menu item. The commandChar property can be abbreviated cmdChar.</td>
</tr>
<tr>
<td>enabled</td>
<td>Determines or changes the enabled or disabled state of a specified menu or menu item; a Boolean value.</td>
</tr>
<tr>
<td>markChar</td>
<td>Determines or changes the current character that indicates a specified menu item is checked.</td>
</tr>
<tr>
<td>menuMessage</td>
<td>Determines or changes the current message to be sent when a specified menu item is chosen.</td>
</tr>
<tr>
<td>[english] name</td>
<td>Determines and changes the language for the specified menu or menu item name. The adjective english lets you code tests for the names of menus and menu items. This is a read-only property for the Tools, Patterns, Font, and Apple menus.</td>
</tr>
<tr>
<td>textStyle</td>
<td>Determines or changes the text style of the specified menu item.</td>
</tr>
</tbody>
</table>

Message Watcher and Variable Watcher Properties

The Message Watcher and Variable Watcher properties described in this chapter pertain to the built-in Message Watcher or Variable Watcher external windows. You can manipulate their properties from a script or from the Message box. You can also manipulate some Variable Watcher properties with the mouse.

The Message Watcher and Variable Watcher properties are listed in Table 12-11. More detailed information about each property is given later in this chapter.
The rest of this chapter contains all of the HyperCard properties in alphabetical order for easy reference. The first line of each description tells which objects (stack, background, card, field, button, rectangle) or elements (menu item, Message Watcher, or Variable Watcher) or environment (global, window, or painting) the property applies to.

Some of the syntax statements and examples in this chapter use the soft return (¬) character to continue long statements onto the next line. The soft return is used here because of the line-length limitations of the page format used in this chapter.

HyperCard Property Descriptions

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hBarLoc</td>
<td>Determines or changes the position of the horizontal bar in the Variable Watcher window.</td>
</tr>
<tr>
<td>hideIdle</td>
<td>Determines or changes whether the “Hide idle” checkbox is checked in the Message Watcher window.</td>
</tr>
<tr>
<td>hideUnused</td>
<td>Determines or changes whether the “Hide unused messages” checkbox is checked in the Message Watcher window.</td>
</tr>
<tr>
<td>rect</td>
<td>Determines or changes the size of the Variable Watcher window. Read-only property for the Message Watcher window.</td>
</tr>
<tr>
<td>vBarLoc</td>
<td>Determines or changes the position of the vertical bar in the Variable Watcher window.</td>
</tr>
</tbody>
</table>
Address

APPLIES TO
Global environment

SYNTAX
put the address [of HyperCard]
get address [of HyperCard]

EXAMPLES
answer the address
put the address into HCPathname

DESCRIPTION
You use the address property to ascertain the path of the currently executing HyperCard program. It returns the program path of your copy of HyperCard in the format “zone:Mac:HyperCard.” For instance, if you’re running HyperCard on a computer named Quill on a network called HyperText, the statement

```
put the address
```

yields

```
HyperText:Quill:HyperCard
```

If your computer is not on a network or the network only has one zone, the address property returns “*:MyMac:HyperCard”. If your computer is also not named, it returns “*:::HyperCard.” This property works only when you are running under System 7.

NOTE
The address property is read-only, and it works only when HyperCard is running under System 7.
CHAPTER 12

Properties

AutoHilite

APPLIES TO

Buttons

SYNTAX

set [the] autoHilite of button to boolean

Button is an expression that yields a button descriptor. Boolean is an expression that yields either true or false.

EXAMPLE

set autoHilite of button 6 to true

DESCRIPTION

You use the autoHilite property to determine or change whether the specified button highlights when that button is pressed.

NOTES

If a button is a member of a button family, then the press of the mouse button in the button’s rectangle not only highlights that button and sets its hilite property to true but also sets the hilite property of the rest of the buttons in the button family to false.

The effect is that the button is highlighted (displayed in inverse video) when the user clicks it, thus giving visual feedback for the click action. If the button is part of a button family, clicking one of the button rectangles unhighlights the rest of the buttons in that family when the mouse button is released.

The default value of autoHilite is false.

The autoHilite property can be set to true or false from a script or by clicking the Auto Hilite checkbox in the Button Info dialog box.

See also the hilite and sharedHilite properties, later in this chapter.
CHAPTER 12

Properties

AutoSelect

APPLIES TO
Fields

SYNTAX
set [the] autoSelect of field to boolean

Field is an expression that yields a field descriptor. Boolean is an expression that yields either true or false.

EXAMPLE
set autoSelect of field "myListField" to true

DESCRIPTION
You use the autoSelect property to determine or change whether the specified field behaves as a list field.

NOTES
You can use the autoSelect property in conjunction with the lockText and dontWrap properties to make a field behave as a list. That is, if autoSelect, dontWrap, and lockText are true, when the user clicks on a line of text in the field, the entire line is selected (and therefore appears highlighted). If the multipleLines property is also true, the user can select multiple lines in the list field by holding down the Shift key while clicking or by dragging the mouse.

You can determine which lines the user selects in a list field using the selectedLine function. You can examine the contents of the lines the user selects in a list field using the selectedText function. You can select one or more lines in a list field from a script using the select command.

The autoSelect property for a button or field can be set to true or false from a script or by clicking the Auto Select checkbox in the Field Info dialog box. When you set a field’s autoSelect property to true, HyperCard auto-
AutoTab automatically sets the field’s `dontWrap` property to `true`. When you set a field’s `dontWrap` property to `false`, HyperCard automatically sets the field’s `autoSelect` property to `false`.

See also the `dontWrap`, `lockText`, and `multipleLines` properties, later in this chapter.

---

**AutoTab**

**APPLIES TO**

Fields

**SYNTAX**

```
set [the] autoTab of field to boolean
```

*Field* is an expression that yields a nonscrolling background or card field descriptor. *Boolean* is an expression that yields either `true` or `false`.

**EXAMPLE**

```
set autoTab of field 6 to true
```

**DESCRIPTION**

You use the `autoTab` property to determine or change whether the specified nonscrolling field will send the `tabKey` message to the current card. When the `autoTab` property is `true`, pressing Return with the insertion point in the last line of that field moves the insertion point to the next field on that card.

The normal tabbing order for fields is as follows: if the field you are leaving is a card field, the insertion point moves to the next higher-numbered card field or the lowest-numbered background field if no higher-numbered card field exists; if the field you are leaving is a background field, the insertion point moves to the next higher-numbered background field or to the lowest-numbered card field if no higher-numbered background field exists.
NOTE

The autoTab property can be changed from a script or by clicking the “Auto tab” checkbox in the Field Info dialog box.

BlindTyping

APPLIES TO

Global environment

SYNTAX

set blindTyping to boolean

Boolean is an expression that yields either true or false.

EXAMPLES

set blindTyping to true
set blindTyping to false
put the blindTyping -- puts current value into message box

DESCRIPTION

You use the blindTyping property to determine or change whether you can type messages into the Message box and send them (execute them) without having the Message box visible. Blind typing is available only if the user level is set to 5 (Scripting) and is usually set with a checkbox on the User Preferences card of the Home stack.

NOTES

The blindTyping property is set to the value saved on the User Preferences card of the standard Home stack by a startup handler in that stack.

If you try to type into the Message box when it’s hidden and blindTyping is false, HyperCard makes the computer beep.
CHAPTER 12

Properties

Bottom

APPLIES TO

Buttons, fields, windows

SYNTAX

set [the] bottom of object to number

Object yields one of the following:

a valid button descriptor in the current stack
a valid field descriptor in the current stack
message [box] or message [window] or window "message"
pattern window or window "patterns" (the Patterns palette)
tool window or window "tools" (the Tools palette)
window "navigator" (the Navigator palette)
scroll window or window "scroll"
window "Fatbits"
message watcher or window "message watcher"
variable watcher or window "variable watcher"
card window
window stackName
menubar
window "Fatbits"
window stackName
scroll window or window "scroll"

Number is an expression that yields an integer representing the vertical offset in pixels from the top of the card window to the bottom of the specified object. When the specified object is a card window, the offset measures from the top of the screen. StackName is an expression that yields the name of an open stack.
EXAMPLES

set bottom of button "Mover" to 64
put bottom of card button 4
put the bottom of this card window
set bottom of message box to 350

DESCRIPTION

You use the bottom property to determine or change the value of item 4 of the rectangle property (left, top, right, bottom) when applied to the specified object or window.

NOTES

Message can be abbreviated msg.

See also the rectangle property, later in this chapter.

BottomRight

APPLIES TO

Buttons, fields, windows

SYNTAX

set [the] bottomRight of object to point

Object yields one of the following:

a valid button descriptor in the current stack
a valid field descriptor in the current stack
message [box] or message [window] or window "message"
pattern window or window "patterns" (the Patterns palette)
CHAPTER 12

Properties

tool window or window "tools" (the Tools palette)
window "navigator" (the Navigator palette)
message watcher or window "message watcher"
variable watcher or window "variable watcher"
card window
window stackName
scroll window or window "scroll"
window "Fatbits"
menubar

*Point* is an expression that yields a list of two integers separated by a comma. *Point* represents the horizontal and vertical offsets, respectively, in pixels from the top-left corner of the card to the bottom-right corner of the specified object. *StackName* is an expression that yields the name of an open stack.

EXAMPLES

```
set bottomRight of bkgnd button id 23 to 64,100
put bottomRight of button "Mover"
put the bottomRight of window "Tools"
set bottomRight of message box to 250,30
```

DESCRIPTION

You use the **bottomRight** property to determine or change the value of items 3 and 4 of the *rectangle* property (left, top, right, bottom) when applied to the specified object or window. The *bottomRight* property can be abbreviated *botRight*.

NOTES

*Message* can be abbreviated *msg*.

See also the *rectangle* property, later in this chapter.
CHAPTER 12

Properties

Brush

APPLIES TO

Painting environment

SYNTAX

set [the] brush to number

*Number* is an expression that yields one of the numbers for the brush shapes shown in Figure 12-2.

EXAMPLE

set brush to 5

DESCRIPTION

You use the *brush* property to determine or to change the current brush shape used by the Brush tool. It’s normally manipulated from the Brush Shape dialog box (shown in Figure 12-2) invoked by choosing Brush Shape from the Options menu or by double-clicking the Brush tool.

The value of the *brush* property can be any integer from 1 to 32, each representing a brush shape from the Brush Shape dialog box. If you set the value of *brush* to a number lower than 1 or higher than 32, it automatically reverts to 1 or 32, respectively. The default *brush* setting is 7.
**C H A P T E R  1 2**

Properties

---

**Figure 12-2** Brush Shape dialog box and property values

![Brush Shape dialog box and property values](image)

---

**CantAbort**

---

**APPLIES TO**

Stacks

**SYNTAX**

```plaintext
set cantAbort of stack stackName to boolean
```

*StackName* is an expression that yields a stack name. *Boolean* is an expression that yields either true or false.
EXAMPLES

- set cantAbort of this stack to true
- set cantAbort of stack "Shoes and socks" to false

DESCRIPTION

The cantAbort property pertains to any stack accessible to your Macintosh. It controls whether or not you can use Command-period to stop execution of a script. This property checks or unchecks the Can’t Abort option in the Protect Stack dialog box.

NOTE

The cantAbort property should be used with caution.

CantDelete

APPLIES TO

- Backgrounds, cards, stacks

SYNTAX

- set cantDelete of object to boolean

Object is an expression that yields a valid background, card, or stack descriptor. Boolean is an expression that yields either true or false.

EXAMPLES

- set cantDelete of first card to true
- set cantDelete of this bkgnd to true
- set cantDelete of this stack to false

388  CantDelete
The `cantDelete` property pertains to any background, card, or stack accessible to your Macintosh. It controls whether or not the user can delete the specified object. The default value of `cantDelete` is false.

For backgrounds and cards, this property checks or unchecks the Can’t Delete option in the object Info dialog box of the specified object.

For stacks, this property checks or unchecks the Can’t Delete Stack option in the Protect Stack dialog box. When the `cantDelete` property for a stack is set to true, the Delete Stack command in the File menu is unavailable. (If the user has checked Can’t Delete Stack, however, and a script sets `cantModify` to true and then false, Can’t Delete Stack is left checked.)

The `cantDelete` property of a stack is automatically set when the user sets the `cantModify` property.

### Syntax

```
set cantModify of stack to boolean
```

*Stack* is an expression that yields a valid stack descriptor. *Boolean* is an expression that yields either true or false.

### Example

```
set cantModify of this stack to true
```
CHAPTER 12

Properties

DESCRIPTION
The `cantModify` property controls whether or not the stack can be changed in any way. This property checks or unchecks the Can’t Modify Stack option and the Can’t Delete Stack option in the Protect Stack dialog box. When the `cantModify` property for a stack is set to `true`, the Compact Stack command in the File menu is unavailable. (If the user has checked Can’t Delete Stack, however, and a script sets `cantModify` to `true` and then `false`, Can’t Delete Stack is left checked.)

NOTES
When you set `cantModify` for a stack from a script, you override whatever the user last set manually in the Protect Stack dialog box. This works in reverse as well. The user can override the script by resetting the value in the Protect Stack dialog box. Setting `cantModify` to `false` does not, however, override protection provided by media that are write-protected in other ways.

See also the `cantDelete` and `userModify` properties in this chapter.

CantPeek

APPLIES TO
Stacks

SYNTAX
`set cantPeek of stack stackName to boolean`

`StackName` is an expression that yields a stack name. `Boolean` is an expression that yields either `true` or `false`.

EXAMPLES
`set cantPeek of this stack to true`
`set cantPeek of stack "TreeFrogs" to false`
DESCRIPTION
The canPeek property pertains to any stack accessible to your Macintosh. It controls whether or not you can view button outlines by pressing Command-Option, view field outlines by pressing Shift-Command-Option, or pop open scripts by clicking the mouse button while peeking. This property also checks or unchecks the Can’t Peek option in the Protect Stack dialog box.

Centered

APPLIES TO
Painting environment

SYNTAX
set [the] centered to boolean

Boolean is an expression that yields either true or false.

EXAMPLES
set centered to true
set centered to false

DESCRIPTION
You use the centered property to determine or to change the Draw Centered setting. When centered is set to true, shapes are drawn from the center rather than the corner. The default value of the centered property is false.

NOTE
You can also set the centered property by choosing Draw Centered on the Options menu.
CHAPTER 12
Properties

CheckMark

APPLIES TO
Menu items

SYNTAX

set [the] checkMark of menuItem of menu to boolean

MenuItem is an expression that yields a menu item descriptor that is in the
menu menu. Menu is an expression that yields a menu descriptor. Boolean is an
expression that yields either true or false.

EXAMPLES

set checkMark of menuItem "Get Back" of menu "Direction" to false

put the checkMark of menuItem "Get Back" of menu "Direction"

if the checkMark of menuItem "Get Back" of menu "Direction" is true then
disable menuItem "Get Back" of menu "Direction"
end if

DESCRIPTION

You use the checkMark property to set or determine whether the checkmark
character for a menu item is currently displayed. When the checkMark
property is set to true, a checkmark character appears to the left of the
menu item.

HyperCard does not automatically check or uncheck custom menu items each
time a menu item is chosen, as it does for its own standard menus. Once an
added menu item with a checkmark is chosen, the checkmark remains next to
the menu item regardless of whether it is chosen again. You need to create
handlers to keep track of the checked and unchecked menu items.
CHAPTER 12

Properties

If you try to set or determine the checkMark property for a menu item that does not exist, HyperCard displays a “No such menu item” dialog box.

NOTE

See also the create menu and put commands in Chapter 10, “Commands,” and the markChar property in this chapter.

CommandChar

APPLIES TO

Menu items

SYNTAX

set [the] commandChar of menuItem of menu to char

menuItem is an expression that yields a menu item descriptor that is in the menu menu. Menu is an expression that yields a descriptor menu. Char is the character you want to use in combination with the Command key as the keyboard equivalent of the specified menu item.

EXAMPLES

set the commandChar of menuItem "Get Back" of menu "Direction" to "D"
put the commandChar of menuItem "Get Back" of menu "Direction"

DESCRIPTION

You use the commandChar property to set or determine the character to be used in combination with the Command key as the keyboard equivalent for a specified menu item. The character is displayed to the right side of the specified menu item along with the Command key symbol, \&.
CHAPTER 12

Properties

Menu items usually send messages when invoked. The menu item message is specified with the menuMessage property, which is described later in this chapter.

CommandChar can be abbreviated cmdChar.

NOTES

If you try to set or determine the commandChar property for a menuItem that does not exist, HyperCard displays a “No such menu item” dialog box.

Command-key equivalents do not work for menu items that have been deleted or disabled. However, with HyperCard’s standard menu items, you can still send a doMenu command to invoke a menu item action even when the menu item has been deleted.

See also the menuMessage property later in this chapter and the create menu and put commands in Chapter 10, “Commands.”

Cursor

APPLIES TO

Global environment

SYNTAX

set cursor to cursorType

CursorType yields one of the cursor names or numbers listed in the description.

EXAMPLES

set cursor to 4
set cursor to plus
DESCRIPTION

The cursor property determines the image that appears at the pointer location on the screen. The cursor setting is the ID number or name of a Macintosh 'CURS' resource, which must be available in the HyperCard file itself or in the current stack file. 'CURS' resources can be installed, removed, and created with a Macintosh resource editor.

HyperCard resets the cursor to the one for the current tool at idle, when no other action is happening. Each available cursor has a name, and some of them also have a number:

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>I-beam</td>
</tr>
<tr>
<td>2</td>
<td>cross</td>
</tr>
<tr>
<td>3</td>
<td>plus</td>
</tr>
<tr>
<td>4</td>
<td>watch</td>
</tr>
<tr>
<td></td>
<td>hand</td>
</tr>
<tr>
<td></td>
<td>arrow</td>
</tr>
<tr>
<td></td>
<td>busy</td>
</tr>
<tr>
<td></td>
<td>none</td>
</tr>
</tbody>
</table>

SCRIPT

The following handler spins the busy cursor two full revolutions. The busy cursor looks like a beach ball. Each time it is set, it turns 45° clockwise:

```plaintext
on spinMe
    repeat 16 times
        set the cursor to busy
        wait 4 ticks
    end repeat
end spinMe
```

NOTE

You can’t get the cursor property or use it as a function; you can only set it.
CHAPTER 12

Properties

Debugger

APPLIES TO

Global environment

SYNTAX

set [the] debugger to debuggerName

DebuggerName is an expression that yields the name of the HyperCard debugger or a custom XCMD debugger.

EXAMPLES

set the debugger to "ScriptEditor"
set the debugger to "MyDebugger"
put the debugger

DESCRIPTION

You use the debugger property to set or determine the current debugger to use when debugging HyperTalk scripts. The default HyperTalk debugger is a built-in XCMD named ScriptEditor.

If you set a name for the debugger that HyperCard can’t find, it uses the built-in HyperTalk debugger.

NOTE

See the description of the HyperTalk debugger in Chapter 3, and see Appendix A, “External Commands and Functions,” for more information about custom debuggers.
DialingTime

**APPLIES TO**

Global environment

**SYNTAX**

```
set [the] dialingTime to numberOfTicks
```

*NumberOfTicks* is a positive integer representing ticks, or sixtieths of a second; the default value is 180 (3 seconds).

**EXAMPLE**

```
set the dialingTime to 300 -- wait 5 seconds
```

**DESCRIPTION**

This property is used to designate how long HyperCard waits before closing the serial connection to the modem after dialing.

**NOTE**

See also the *dial* command in Chapter 10, “Commands.”
CHAPTER 12
Properties

DialingVolume

APPLIES TO
Global environment

SYNTAX
set [the] dialingVolume to volume

Volume is an integer from 0 to 7, inclusive; the default value is 7.

EXAMPLE
if the dialingVolume is 7 -- too loud
then set the dialingVolume to 4

DESCRIPTION
This property is used to control the volume of the touch tones generated through the Macintosh speaker by the dial command.

DontSearch

APPLIES TO
Backgrounds, cards, fields

SYNTAX
set [the] dontSearch of object to boolean

Object is an expression that yields any valid background, card, or field descriptor. Boolean is an expression that yields either true or false.
EXAMPLES

set dontSearch of bknd 4 to true
put the dontSearch of bknd 3 into msg

if the short name of bknd field 2 of this cd is "Secrets"
then set dontSearch of bknd field 2 to true

DESCRIPTION

You use the dontSearch property to set or determine whether or not the find command searches the specified background, card, or field in the current stack. When the dontSearch property of an object is set to true, the find command doesn't search that object. The default value for the dontSearch property is false.

When the dontSearch property of a background is set to true, the find command ignores all card or background fields on all of the cards of the specified background.

NOTE

You can set the dontSearch property from a script or by clicking the Don’t Search checkbox in the object’s Info dialog box.

DontWrap

APPLIES TO

Fields

SYNTAX

set [the] dontWrap of field to boolean

Field is an expression that yields any valid field descriptor. Boolean is an expression that yields either true or false.
CHAPTER 12

Properties

EXAMPLE

set the dontWrap of bkgnd fld 4 to true

DESCRIPTION

You use the dontWrap property to determine or change the dontWrap value for a field in the current stack. When the dontWrap property of a field is set to true, the text in the specified field does not wrap around to the next line at the boundary of the field. The default value for the dontWrap property is false (wrap at the boundary of the field).

NOTES

You can also set the dontWrap property by clicking the Don’t Wrap checkbox in the Field Info dialog box. When you set a field’s autoSelect property to true, HyperCard automatically sets dontWrap to true. When you set a field’s dontWrap property to false, HyperCard automatically sets autoSelect to false.

DragSpeed

APPLIES TO

Global environment

SYNTAX

set dragSpeed to number

Number is an expression that yields a positive integer; 1 is the slowest possible speed.

EXAMPLE

set dragSpeed to 120
CHAPTER 12

Properties

DESCRIPTION

The `dragSpeed` property determines how many pixels per second the pointer moves when manipulated by all subsequent `drag` commands.

SCRIPT

The following handler, placed in a button’s script, creates a new card to draw on, sets a slow `dragSpeed` value, and slowly draws a diamond shape. It then fills the diamond with a pattern, waits a short time, deletes the card, and sends you back to the card where you started. You can change the `dragSpeed` property to drag faster by increasing the `dragSpeed` value, or slower by decreasing the value.

```plaintext
on mouseUp
    push card
    doMenu "New Card"
    set dragSpeed to 60
    choose line tool
    drag from 100,50 to 50,100
    drag from 50,100 to 100,150
    drag from 100,50 to 150,100
    drag from 100,150 to 150,100
    choose bucket tool
    set pattern to 10
    click at 100,100
    reset paint
    wait 25 ticks
    doMenu "Delete Card"
    choose browse tool
    pop card
end mouseUp
```

`DragSpeed` affects all of the Paint tools except the Bucket and Text tools. At idle time, HyperCard resets the `dragSpeed` property to 0. In this case, a value of 0 represents the fastest possible speed.
CHAPTER 12

Properties

EditBkgnd

APPLIES TO
Global environment

SYNTAX

```
set editBkgnd to boolean
```

*Boolean* is an expression that yields either *true* or *false*.

EXAMPLES

```
set editBkgnd to true
set editBkgnd to false
```

DESCRIPTION

The *editBkgnd* property determines where any painting or creating of buttons or fields happens—on the current card (*false*) or on its background (*true*). It's usually set with the Edit menu and is available only when the user level is Painting (3) or higher. The default setting is *false* (editing on the card).

Enabled

APPLIES TO
Menus, menu items, and buttons
CHAPTER 12

Properties

SYNTAX

set [the] enabled of [menuItem of] menu to boolean
set [the] enabled of button to boolean

MenuItem is an expression that yields the descriptor of a menu item that is in the specified menu. Menu is an expression that yields a menu descriptor. Button is an expression that yields a button descriptor. Boolean is either true or false.

EXAMPLES

set the enabled of button "Go For It" to true
set enabled of menuItem 4 of menu "Utilities" to false
the enabled of menu "Home"
the enabled of menuItem "Repeat" of menu "Control"
put the enabled of the fifth menu

DESCRIPTION

You use the enabled property to set or determine the state (either enabled or disabled) of a specified menu, menu item, or button. When you create any of these, the default for the enabled property is true and the menu, menu item, or button appears in solid outline (active). If you set enabled to false, the object is dimmed (disabled).

When the enabled of a button is false, it does not receive mouseDown, mouseStillDown, mouseUp, or mouseDoubleClick messages. However, the button continues to receive mouseEnter, mouseWithin, and mouseLeave messages.

NOTES

If you try to set or determine the enabled property for a menu, menu item, or button that does not exist, HyperCard displays a dialog box informing you of your error.

Command-key equivalents do not work on custom menu items that have been disabled.

See also the menuMsg property in this chapter and the create menu, disable, enable, and put commands in Chapter 10, “Commands.”
CHAPTER 12

Properties

Environment

APPLIES TO
Global environment

SYNTAX
put [the] environment

EXAMPLE
if the environment is "development" then set userLevel to 5

DESCRIPTION
The environment property returns development if the currently running version of HyperCard is the fully enabled development version, and player if the HyperCard Player is running.

Family

APPLIES TO
Buttons

SYNTAX
set [the] family of buttonName to number
put [the] family of buttonName

ButtonName is an expression that specifies either a background or card button descriptor; if you don’t specify whether the owner of a family is a card or background, the default owner is the card. Number is a positive integer between 1 and 15, inclusive, which represents the family number of a group of buttons; the number 0 indicates that the specified button does not belong to a family.
CHAPTER 12

Properties

EXAMPLES

set the family of button 6 to 7
set the family of button "Home" to 0 --no family

SCRIPT

The following example handler sets up a family of radio buttons so they automatically function properly, with only one highlighted at a time:

on setFamily
    repeat with i = 1 to 5
        set the family of card button i to 2
    end repeat
end setFamily

DESCRIPTION

HyperCard uses the family property to group related buttons of a card or background. This grouping provides a convenient means to make sure that only one button of a group is highlighted at one time. When someone clicks one of the buttons in a family, then that button's hilite property is set to true and the hilite property of any previously highlighted button in that family is automatically set to false.

You can assign any number of buttons to a family and can have up to 15 families of buttons on any card or background. You can use the set command to assign a button to a family from a script, or you can do it manually by using the Family pop-up menu in each button's Button Info dialog box.

NOTES

Buttons can be members of either a family of background buttons or a family of card buttons but cannot belong to both families. A group of card buttons of family 6 are totally unrelated to the family 6 background buttons.

You can also use the Family pop-up menu in the Button Info dialog box to assign a family to a button. When you assign a button to a family, HyperCard sets its autoHilite property to true. HyperCard preserves the state of the
autoHilite property of a button existing prior to assigning a family; if you later select None in the Family pop-up menu, HyperCard restores the former autoHilite state.

See also the sharedHilite and autoHilite properties in this chapter.

Filled

APPLIES TO

Painting environment

SYNTAX

set [the] filled to boolean

Boolean is an expression that yields either true or false.

EXAMPLES

set filled to true
set filled to false

DESCRIPTION

You use the filled property to determine or to change the Draw Filled setting. When filled is true, the current pattern on the Patterns palette is used to fill shapes as they are drawn. The default value of the filled property is false.

NOTE

You can also set the filled property by choosing Draw Filled from the Options menu.
CHAPTER 12

Properties

**FixedLineHeight**

**APPLIES TO**

Fields

**SYNTAX**

```
set [the] fixedLineHeight of field to boolean
```

*Field* is any expression that yields the descriptor of a field. *Boolean* is an expression that yields either *true* or *false*.

**EXAMPLES**

```
set the fixedLineHeight of field 6 to true
get fixedLineHeight of bkgnd field 3

if fixedLineHeight of field 6 is false then
   put "fixedLineHeight is false"
end if
```

**DESCRIPTION**

You use the *fixedLineHeight* property to determine or specify whether a field has fixed line spacing when the text is of different sizes. If widely varying sizes of text are going to be used in a field, the value of the *fixedLineHeight* property needs to be set to *false*. The default setting of *fixedLineHeight* is *false*.

**NOTES**

You can also change the value of the *fixedLineHeight* property by clicking the Fixed Line Height checkbox in the Field Info dialog box.

The *fixedLineHeight* property is *true* for fields created with versions of HyperCard earlier than 2.0. When the same stacks are converted to the HyperCard 2.0 format, *fixedLineHeight* remains *true*.
The `fixedLineHeight` property is set to `true` when `showLines` is set to `true`. If `fixedLineHeight` is set to `false`, `showLines` is also set to `false`. See the `textSize` property, later in this chapter, for more information on how to set different sizes of text in fields.

### FreeSize

**APPLIES TO**

Stacks

**SYNTAX**

```plaintext
put [the] freeSize of stack stackName [into container]
```

*StackName* is an expression that yields any stack name currently available to HyperCard, and *container* is any field, variable, the selection, or the Message box.

**EXAMPLE**

```plaintext
put freeSize of stack "dogfeathers" into field "Size"
```

**DESCRIPTION**

You use the `freeSize` property to determine the amount of free space of the specified stack in bytes. (Free space changes in a stack each time you add or delete an object.)

**SCRIPT**

The following handler compacts a stack based on a specified `freeSize` value:

```plaintext
on closeStack
    if the freeSize of this stack > 24000
        then doMenu "Compact Stack"
    end if
end closeStack
```
CHAPTER 12

Properties

NOTE

The `freeSize` property can be changed only by choosing Compact Stack from the File menu (or executing the HyperTalk command `doMenu Compact Stack`), which changes its value to 0, or by editing the stack.

Grid

APPLIES TO

Painting environment

SYNTAX

`set [the] grid to boolean`

`Boolean` is an expression that yields either `true` or `false`.

EXAMPLES

`set grid to true`
`set grid to false`

DESCRIPTION

You use the `grid` property to determine or to change the painting grid setting. When the value of `grid` is `true`, movement of the Rectangle, Round Rect, Oval, and Polygon Paint tools is constrained to 8-pixel intervals. The default value of the `grid` property is `false`.

NOTE

You can set the `grid` property from a script or by choosing Grid from the Options menu.
CHAPTER 12

Properties

HBarLoc

APPLIES TO
Variable watcher windows

SYNTAX
set [the] hBarLoc of window "Variable Watcher" to number

Number is an expression that yields a positive integer that represents the offset in pixels from the bottom of the Variable Watcher window title bar to the horizontal bar in the window.

EXAMPLES
set the hBarLoc of window "Variable Watcher" to 123
put the hBarLoc of window "Variable Watcher"

DESCRIPTION
You use the hBarLoc property to determine or to change the current position of the horizontal bar in the Variable Watcher window. The horizontal bar determines how many of the variable name and value fields are visible in the Variable Watcher window.

NOTES
The built-in Variable Watcher window is a HyperCard XCMD. It can be replaced with a custom variable watcher XCMD by setting the variableWatcher property to the name of a variable watcher XCMD.

Custom variable watcher XCMDs can respond to or ignore the hBarLoc property.

For more information about creating and calling a custom variable watcher XCMD, see Appendix A, “External Commands and Functions.”

See also the description of the Variable Watcher in Chapter 3, “The Scripting Environment,” and the rect, variableWatcher, and vBarLoc properties, later in this chapter.
CHAPTER 12

Properties

Height

APPLIES TO

Buttons, cards, fields, menu bar, windows

SYNTAX

set [the] height of object to number
put [the] height of object

Object is an expression that yields any valid button, card, field, or window descriptor, or the word menubar.

Number is an expression that yields a positive integer. Number represents the total number of pixels in the vertical height of the specified object.

EXAMPLES

set the height of bkgnd button 2 to 60
set the height of bkgnd field "phoneList" to 220
put the height of window "Home"
set the height of cd window to height of cd window div 2

DESCRIPTION

You use the height property to determine or change the vertical distance in pixels occupied by the rectangle of the specified button, field, or window. You can change the height of a button, field, or window rectangle with the set command.

When you set the height of a button, field, or window, its location property (center coordinate) remains the same.

NOTE

See also the rectangle property, later in this chapter.
CHAPTER 12
Properties

**Hid Idle**

**APPLIES TO**
Message watcher windows

**SYNTAX**

```plaintext
set [the] hideIdle of window messageWatcher to boolean
```

*MessageWatcher* is an expression that yields the name of a message watcher window. *Boolean* is an expression that yields either *true* or *false*.

**EXAMPLES**

```plaintext
set the hideIdle of window "Message Watcher" to true
put hideIdle of window "Message Watcher"
```

**DESCRIPTION**
You use the *hideIdle* property to determine or change whether the “Hide idle” checkbox is checked in a message watcher window.

The name of the message watcher window can be either the built-in HyperCard Message Watcher, *window "Message Watcher"*, or the name of a custom message watcher window that supports a “Hide idle” checkbox.

**NOTES**
See also the description of the Message Watcher in Chapter 3, “The Scripting Environment.” For more information about creating a custom message watcher XCMD, see Appendix A, “External Commands and Functions.”

See also the *hideUnused* and *messageWatcher* properties, later in this chapter.
CHAPTER 12

Properties

HideUnused

APPLIES TO
Message watcher windows

SYNTAX

set [the] hideUnused of window messageWindow to boolean

MessageWindow is an expression that yields the name of a message watcher window. Boolean is an expression that yields either true or false.

EXAMPLES

set the hideUnused of window "Message Watcher" to true
put hideUnused of window "Message Watcher"

DESCRIPTION

You use the hideUnused property to determine or change whether the “Hide unused messages” checkbox is checked in a message watcher window.

The name of the message watcher window can be either the built-in HyperCard Message Watcher, window "Message Watcher", or the name of a custom message watcher window that supports the “Hide unused messages” checkbox.

NOTE

See also the description of the Message Watcher in Chapter 3, “The Scripting Environment.” For more information about creating a custom message watcher XCMD, see Appendix A, “External Commands and Functions.”

See also the hideIdle and messageWatcher properties in this chapter.
CHAPTER 12

Properties

Hilite

APPLIES TO

Buttons

SYNTAX

set [the] hilite of button to boolean

Button is an expression that yields a background button or card button descriptor. Boolean is an expression that yields either true or false.

EXAMPLES

set hilite of button "You're on" to true
put the hilite of bkgnd button 3

DESCRIPTION

You use the hilite property to determine or to change whether the specified button is highlighted (displayed in inverse video). The default value of the hilite property is false. To see what highlighting for the various button styles looks like, see the HyperCard Reference.

NOTES

The hilite property can be changed using the set command, either from a script or from the Message box, or, if the autoHilite property is true, by pressing the button. In that case, for all styles of buttons except checkboxes and radio buttons, the hilite property becomes true when you press the button, and it becomes false when you release it.

For checkboxes and radio buttons with their autoHilite property set to true, the hilite property toggles to its opposite state when the button is clicked and stays that way until it is clicked again. That is, when a checkbox is highlighted, it appears with an “X” checkmark in its box; when it’s not highlighted, the checkmark does not appear. If autoHilite is true, an unselected checkbox displays an “X” when you click it; if you click it again,
the “X” disappears. The appearance of the checkmark correlates to the state of the button’s hilite property. The situation is similar for radio buttons, except that the true highlighted state is indicated by a solid dot inside the button’s circle.

See also the description of the autoHilite, family, and sharedHilite properties in this chapter.

### Icon

**APPLIES TO**

Buttons

**SYNTAX**

```
set [the] icon of button to designator
```

*Button* is an expression that yields a background or card button descriptor. *Designator* yields the ID number of an available icon resource or the name of an icon (if it has one).

**EXAMPLES**

```
set icon of button "Instant" to 5005
set icon of button "Instant" to "DoGood"
put the icon of button "Instant"
```

**DESCRIPTION**

You use the icon property to determine or to change the icon, if any, that is displayed with the specified button. If a button has no icon, the icon property is 0. An icon is identified by its ID number or by its name, if it has one.

Icons are small images that exist as Macintosh files and are editable with the HyperCard icon editor. For an icon to be displayed on a button, its resource must be available in the current stack, another stack in the hierarchy, or the HyperCard application.
The icon can also be changed by clicking the Icon button in the Button Info dialog box, which brings up another dialog box that displays the available icons graphically. When you click an icon in the icon display dialog box, the ID and name are displayed in the upper-left corner. The icon name is displayed in the dialog box only if the icon has a name.

If you use the `put` command with the `icon` property, you get the ID number of the icon, not the name. You cannot retrieve an icon name for a button.

**ID**

**APPLIES TO**

Backgrounds, buttons, cards, fields, menus, windows, HyperCard

**SYNTAX**

```
put [the] [adjective] ID of object | windowName | HyperCard ¬
[into container]
```

*Adjective* is one of the *long, short, and abbreviated* modifiers as described in the section “Object ID Numbers” in Chapter 5. *Object* is an expression that yields any valid background, button, card, or field descriptor. *WindowName* yields a valid window descriptor. A *container* is the selection, a field, a variable, or the Message box.

**EXAMPLES**

```
put the ID of HyperCard
put the long ID of bkgrd 3
if the ID of bkgrd 1 is 2282 then answer "Welcome Home"
put the short ID of card 35
put the id of field 1 into msg
put the ID of bkgrd button 3 into field "Button IDs"
put the short ID of card 35 after line 2 of field 2
put the id of window "DogPicture"
```
CHAPTER 12

Properties

DESCRIPTION
You use the ID property to determine the permanent ID number of any background, button, card, field, or window in the current stack.

You can also use the ID property to determine the application signature of HyperCard. Unless the current stack is running under a stand-alone application whose application signature has been modified, the ID property of HyperCard will contain WILD.

You can’t use the set command to change the ID of any object.

SCRIPT
The following script uses the name and ID properties to produce a list of button names and IDs. You need to create a field for the button name and ID list.

on mouseUp
  put empty into field "MyField"
  repeat with nums = 1 to the number of buttons
    put "Button name" && quote & short name of button ¬
    nums & quote && "has id number" && id of button ¬
    nums & return after field "MyField"
  end repeat
end mouseUp

You can place the script in any button or field, then click the button or field to get a list of all the buttons names and IDs on the current card. If you put the script into the same field you want to put the list into, be sure to set the lockText property of that field to true, so that the field receives the mouseUp message. Also, change all references to the field descriptor field "MyField" to match the descriptor for the field you created for the list. If the field created for the list is the same field the script is in, you can use me in place of field "MyField".

NOTES
When HyperCard retrieves the ID of a window, field, or button, it ignores any adjectives and always reports the ID as an unlabeled number. For instance, if
you execute the following command line from the Message Box, you will get a number result like the one shown on the line following it:

```plaintext
put the long id of window "Home"
10883218
```

If you ask for a long ID of a card, HyperCard returns a labeled response, as shown in this example:

```plaintext
put the long id of cd 1
```

The response returned is

```plaintext
card id 3916 of stack "oh dear:Desktop Folder:Home"
```

You should also be aware that a window must already exist before you call for its ID. Many windows go out of existence when you click their close boxes and are created by XCMDs each time you call for them.

---

**ItemDelimiter**

**APPLIES TO**

Global environment

**SYNTAX**

```plaintext
the itemDelimiter
set [the] itemDelimiter to character
```

*Character* yields an ASCII character or a constant that represents an ASCII character.

**EXAMPLES**

```plaintext
if the itemDelimiter <> comma
then set the itemDelimiter to comma

set itemDelimiter to "#"
```
CHAPTER 12

Properties

DESCRIPTION
You use the itemDelimiter property to change the character that delimits items in a list. The default value is comma, and, if changed, the value will revert to comma on idle.

Changing the item delimiter has no effect on comma-delimited HyperCard structures such as dateItems, location, and rectangle.

SCRIPT
The following card script’s function handler returns the pathname of the current stack without the stack name (useful when you need to refer to other files at the same level). You can call this function handler by typing shortPath() in the Message box and then pressing Return.

```plaintext
function shortPath -- Card handler
    -- Save old item delimiter value for resetting later
    put the itemDelimiter into oldDelimiter
    put the value of word 2 of the long name ¬
    of this stack into longName
    -- saves: Volume:Stacks Folder:This Stack
    -- reset item delimiter
    set itemDelimiter to colon
    delete last item of longName
    -- 'Volume:Stacks Folder' goes in longName
    -- Reset delimiter
    set itemDelimiter to oldDelimiter
    return longName & colon
end shortPath
```

The following part of a script is useful for retrieving the name of a program from a colon-delimited list on machines running system software version 7.0 or later:

```plaintext
set the itemDelimiter to ":"
answer program "Select a program"
get the last item of it
```
CHAPTER 12

Properties

Language

APPLIES TO
Global environment

SYNTAX
set language to languageName

$languageName$ is a text string that yields English or a language for which there is a HyperTalk translator resource.

EXAMPLE
set language to French

DESCRIPTION
You use the language property to choose a HyperTalk translator, which is a code resource that translates between HyperTalk and a foreign-language version of HyperTalk. If the language property is not English, when the user invokes the script editor to view a script, HyperCard translates it to the specified language. When the user closes the script, HyperCard translates it back to English HyperTalk before storing it with its object.

NOTES
The language property refers only to the HyperTalk scripting language; it has no effect on scripts written in other scripting languages.

The languages available depend on the script translator resources available in the current stack, another stack in the hierarchy, or the HyperCard application. The default setting is English, and it's always available.
CHAPTER 12

Properties

Left

APPLIES TO

Buttons, fields, menu bar, windows

SYNTAX

set [the] left of object to number

*Object* yields one of the following:

- a valid button descriptor in the current stack
- a valid field descriptor in the current stack
- message [box] or message [window] or window "message"
- pattern window or window "patterns" (the Patterns palette)
- tool window or window "tools" (the Tools palette)
- window "navigator" (the Navigator palette)
- scroll window or window "scroll"
- window "Fatbits"
- message watcher or window "message watcher"
- variable watcher or window "variable watcher"
- card window
- window stackName

*Number* yields an integer that is the horizontal offset in pixels from the left side of the card to the left side of the object. When the object is the card window, the offset is relative to the left side of the screen. *StackName* is an expression that yields the name of an open stack.

EXAMPLES

set left of button 2 to 34
put left of button 2
put the left of card field 3
set left of tool window to 65
CHAPTER 12

Properties

DESCRIPTION
You use the left property to determine or change the value of item 1 of the rectangle property (left, top, right, bottom) when applied to the specified object or window. The left property of an object can also be set to a value off the screen. Setting the left property of an object to a value off the screen makes the object seem hidden.

NOTES
The left of the menu bar is a read-only property.
Message can be abbreviated msg.
See also the rectangle property, later in this chapter.

LineSize

APPLIES TO
Painting environment

SYNTAX
set [the] lineSize to number

Number yields a positive integer that is the total number of pixels in a line’s width. It can be 1, 2, 3, 4, 6, or 8.

EXAMPLE
set lineSize to 8

DESCRIPTION
You use the lineSize property to determine or to change the thickness of the lines drawn by the Line and Shape tools. The default value of the lineSize property is 1. If you set the value of lineSize to a number lower than 1 or higher than 8, it automatically reverts to 1 or 8, respectively.
NOTE

You can also set the lineSize property by choosing Line Size from the Options menu or double-clicking the Line tool.

Location

APPLIES TO

Buttons, fields, menu bar, windows

SYNTAX

set location of object to point

Object yields one of the following:

- a valid button descriptor in the current stack
- a valid field descriptor in the current stack
- message [box] or message [window] or window "message"
- pattern window or window "patterns" (the Patterns palette)
- tool window or window "tools" (the Tools palette)
- window "navigator" (the Navigator palette)
- scroll window or window "scroll"
- window "Fatbits"
- message watcher or window "message watcher"
- variable watcher or window "variable watcher"
- card window
- window stackName
- windows created with the picture command

Point is an expression that yields two integers separated by a comma.
StackName is an expression that yields the name of a stack in quotation marks.
EXAMPLES

set loc of tool window to "100,100"
put the loc of field 3
put the loc of pattern window
set loc of msg to 30,150
set the loc of card window to 48,90
set the loc of window "Navigator" to "45,60"
set the loc of scroll window to "165,45"

DESCRIPTION

The location property sets or retrieves the location at which the window or object is displayed.

The point represents the horizontal and vertical offsets, respectively, in pixels from the top-left corner of the card to the center of a resizable window (a button or field) or the top-left corner of a nonresizable window (Tools palette, Patterns palette, Message box, Navigator palette, or Scroll window), disregarding the drag bar at the top of the window. The value for point must be within quotation marks for the Navigator palette.

The point for a card window represents the horizontal and vertical offsets, respectively, in pixels from the top-left corner of the screen to the top-left corner of the card window.

The point for windows created with the picture command represents the horizontal and vertical offsets, respectively, in pixels from the top-left corner of the current card window to the top-left corner of the picture window. The value for point must be within quotation marks.

NOTES

The location of the menu bar is a read-only property.

If you always put the value of point within quotation marks, it works with all of the HyperCard objects and elements for which you can set the location. An example that places quoted values in variables is shown under the multiSpace property, later in this chapter.

If you want to move a card on the screen, you set the location property for the card window of the current stack, not the location of the card.
CHAPTER 12

Properties

The number that represents the horizontal offset for the card window is shifted to the closest multiple of 16 regardless of how you set it. For example, the statement `set the loc of cd window to 50,90` would result in the card location of `48,90`. It would shift to the next 16 pixels when the horizontal value of the `location` property reached the halfway point to the next higher or lower 16 pixels. For example, a horizontal value of 38 would shift the card window left to a horizontal offset of 32.

When you move a card window with the `location` property, the system message `moveWindow` is sent. The `moveWindow` message is also sent when you drag the window to a new location, zoom it in or out with the zoom box, causing the `location` property to change, or show the card window at a new location with the `show` command.

See also the `rectangle` property later in this chapter; the `palette`, `picture`, and `show` commands in Chapter 10, “Commands”; and the `moveWindow` system message in Chapter 8, “System Messages.”

### LockErrorDialogs

**APPLIES TO**

Global environment

**SYNTAX**

```
set lockErrorDialogs to boolean
```

`Boolean` is an expression that yields either `true` or `false`.

**EXAMPLE**

```
set lockErrorDialogs to true
```

**DESCRIPTION**

You use the `lockErrorDialogs` property to prevent HyperCard from displaying error dialogs when a script causes an error. This property is set to `false` at idle time, so it has no effect if you enter it through the Message box.
NOTES

When the lockErrorDialogs property is set to true, HyperCard sends the message errorDialog errorMessage to the current card instead of displaying the error dialog.

Errors produced through the Message box still get dialogs, regardless of the setting of lockErrorDialogs.

LockMessages

APPLIES TO

Global environment

SYNTAX

set lockMessages to boolean

Boolean is an expression that yields either true or false.

EXAMPLES

set lockMessages to true
set lockMessages to false

DESCRIPTION

You use the lockMessages property to prevent HyperCard from sending all open, close, suspend, and resume system messages. The default setting is false. HyperCard resets lockMessages to false at idle time (in effect, at the end of all pending handlers).

NOTE

Setting the lockMessages property to true speeds up execution of scripts in which you go to cards, and those in which you create and delete cards, backgrounds, and stacks, because it prevents HyperCard from sending
messages such as openCard, closeCard, and so on. The lockMessages property does not affect new and delete system messages such as newCard and deleteField. Setting the lockMessages property to true also prevents execution of handlers invoked by system messages, which may be used to set up an environment—hiding the Message box, and so on. It’s particularly useful when you want to go to a card to retrieve or write some information, but you don’t want to stay there.

**LockRecent**

**APPLIES TO**

Global environment

**SYNTAX**

```
set lockRecent to boolean
```

*Boolean* is an expression that yields either true or false.

**EXAMPLE**

```
set lockRecent to true
```

**DESCRIPTION**

You use the lockRecent property to prevent HyperCard from adding miniature representations to the Recent Cards dialog box. (The Recent Cards dialog box is invoked by Command-R or by choosing Recent from the Go menu.)

The default setting is false. HyperCard resets lockRecent to false at idle time (in effect, at the end of all pending handlers).
LockRecent is set to true when the lockScreen property is set to true regardless of the current setting of lockRecent. Setting the lockRecent property to true speeds up execution of scripts in which you go to cards. See also the next property, lockScreen.

LockScreen

APPLIES TO
Global environment

SYNTAX

set lockScreen to boolean

Boolean is an expression that yields either true or false.

EXAMPLE

set lockScreen to true

DESCRIPTION

You use the lockScreen property to prevent HyperCard from updating the screen when you go to another card.

The default setting is false. HyperCard resets lockScreen to false at idle time (in effect, at the end of all pending handlers).

NOTES

Setting the lockScreen property to true enables you to open different cards without displaying them on the screen, and it speeds up execution of scripts in which you go to cards. For example, you can lock the screen, then go to another
CHAPTER 12

Properties

card to read information out of a field, then return to the first card without
having the second card appear to the user.

To ensure that the display is unlocked, each `set lockScreen to true`
must be balanced with a `set lockScreen to false`.

See also the `lock` and `unlock` commands in Chapter 10.

LockText

APPLIES TO

Fields

SYNTAX

```
set [the] lockText of field to boolean
```

*Field* is an expression that yields any valid card field or background field
descriptor. *Boolean* is an expression that yields either *true* or *false*.

EXAMPLES

```
set the lockText of field "Glossary" to true
set the lockText of field "Glossary" to false
```

DESCRIPTION

You use the `lockText` property to prevent or allow editing of text within a
field in the current stack.

When the Browse tool is selected and the pointer is moved over an unlocked
field, the pointer changes to an I-beam; clicking then lets you edit the text in
the field. If the field is locked (`lockText is true`), the cursor doesn’t change,
and the text cannot be edited. A locked field also receives the messages
`mouseDown`, `mouseStillDown`, and `mouseUp` when you click it. The default
value of `lockText` is `false`. 
NOTES

You can also change this property by clicking the Lock Text checkbox in the Field Info dialog box.

When the cantModify property for the current stack is true and the userModify property is false, no changes can be made in a field. When cantModify is true, userModify is true, and lockText is false, any editing done in a field is lost when the user moves to another card.

**LongWindowTitles**

**APPLIES TO**

Global environment

**SYNTAX**

```
set [the] longWindowTitles to boolean
```

_Boolean_ is an expression that yields either _true_ or _false_.

**EXAMPLES**

```
set longWindowTitles to true
set the longWindowTitles to false
```

**DESCRIPTION**

You use the _longWindowTitles_ property to determine whether the stack name or the full pathname appears in the title bar of the card window.

The default value of _longWindowTitles_ is _false_ and shows only the stack name in the title bar. If _longWindowTitles_ is set to _true_, the full pathname of the stack is shown in the card window title bar.

**NOTE**

See also Chapter 5, “Referring to Objects, Menus, and Windows,” for more information about stack names and the full pathname of a stack.
CHAPTER 12

Properties

MarkChar

APPLIES TO
Menu items

SYNTAX
set [the] markChar of menuItem of menu to char

menuItem is an expression that yields a menu item descriptor that is in the menu menu. Menu is an expression that yields a menu descriptor. Char is an expression that yields the checkmark character you want to display in the menu to the left side of the specified menu item when it is chosen.

EXAMPLES
set markChar of menuItem "Blue" of menu "Colors" to >
put the markChar of menuItem 4 of the sixth menu

DESCRIPTION
You use the markChar property to set or determine the checkmark character for a menu item. When the markChar of a menu item is not empty, the menu item's checkMark property is true. To remove a checkmark character from a menu item, set its markChar property to empty or set its checkMark property to false.

HyperCard does not automatically check or uncheck custom menu items each time a menu item is chosen, as it does for its own standard menus. Once an added menu item with a checkmark is chosen, the checkmark remains next to the menu item regardless of whether it is chosen again. You need to create handlers to keep track of checked and unchecked custom menu items.

If you try to set or determine the markChar property of a menu item that does not exist, HyperCard displays a “No such menu item” dialog box.

NOTE
See also the put command in Chapter 10, “Commands,” and the checkMark property, earlier in this chapter.
CHAPTER 12

Properties

Marked

APPLIES TO
Cards

SYNTAX

set [the] marked of card to boolean

*Card* is an expression that yields the descriptor of any card within the current stack. *Boolean* is an expression that yields either *true* or *false*.

EXAMPLES

put [the] marked of card 35
set marked of card 2 to true
if the marked of this card is true then doSomething

DESCRIPTION

You use the *marked* property to set or determine the marked state of any card in the current or specified stack. The default value of *marked* is *false*. A marked card or group of marked cards can be referred to in complicated searches and when printing.

NOTES

You can also set the *marked* property with the Card Marked option in the Card Info dialog box.

See also the *mark* and *unmark* commands in Chapter 10, "Commands."
CHAPTER 12

Properties

**MenuMessage**

**APPLIES TO**

Menu items

**SYNTAX**

```
set [the] menuMessage of menuItem of menu to message
```

*MenuItem* is an expression that yields a menu item descriptor that is in the menu *menu*. *Menu* is an expression that yields a menu descriptor. *Message* is an expression that yields a message to be sent by the menu item.

**EXAMPLES**

```
set menuMessage of menuItem 7 of menu 10 to doMenu "Prev"
put the menuMessage of menuItem "Next" of menu "Pictures"
put "Maple" after menu "Syrups" with menuMsg "go card 4"
```

**DESCRIPTION**

You use the *menuMessage* property to specify the message sent to the current card by a specified menu item. You can also specify a menu message for a menu item when you create a menu item with the *put* command.

You may respond to the choosing of menu items in your scripts in two ways. Whenever a menu item is chosen, HyperCard sends a *doMenu* message. You can respond to the choosing of menu items by writing *doMenu* handlers in your scripts (this is the first way). If the *doMenu* message is not intercepted by any script, HyperCard checks the menu item's *menuMessage* property. If *menuMessage* is not empty, HyperCard sends that message to the current card. Therefore you can respond to the choosing of menu items by assigning them menu messages (this is the second way). If *menuMessage* is empty, HyperCard checks whether the item is one of its standard menu items. If it is a standard menu item, the standard behavior of that menu item occurs.
Custom menu items with the same name as a standard HyperCard menu item inherit the standard behavior of the HyperCard menu item. For example, if you add Background to a menu called Special, choosing it has the same effect as choosing the standard Background command from the Edit menu, unless you assign a custom menu message or intercept the `doMenu` message.

If you try to set or determine the `menuMessage` property of a menu item that does not exist, HyperCard displays a “No such menu item” dialog box.

**NOTES**

In the case of HyperCard’s standard menu items, the `doMenu` command works even when the item is deleted. For example, if the following handler is executed, HyperCard exits to the Finder:

```plaintext
on mouseDown
  delete menuItem "Quit HyperCard" from menu "File"
  doMenu "Quit HyperCard"
end mouseDown
```

The `menuMessage` property returns values for standard HyperCard menus only if they have been changed from the default HyperCard menu messages.

`MenuMessage` can be abbreviate `menuMsg`.

You can’t set the `menuMessage` property of menu items in the Tools, Patterns, Font, or Apple menus, except for the `menuMessage` property of the About HyperCard menu item in the Apple menu, which you can set.

See also the `create menu`, `delete`, `doMenu`, and `put` commands in Chapter 10, “Commands.”
CHAPTER 12

Properties

MessageWatcher

APPLIES TO
Global environment

SYNTAX

set [the] messageWatcher to \textit{name}

\textit{Name} is an expression that yields a valid message watcher XCMD name.

EXAMPLES

set messageWatcher to "MyWatcher"
put the messageWatcher

DESCRIPTION

You use the \textit{messageWatcher} property to determine or to change the current message watcher. The default value is \textit{messageWatcher}, the built-in message watcher XCMD. You display the current message watcher with the \textit{show} command or by setting the \textit{visible} property of the message watcher window to \textit{true}.

The built-in message watcher is a HyperCard XCMD. It can be replaced with a custom message watcher XCMD by setting the \textit{messageWatcher} property to the name of a custom message watcher XCMD.

NOTES

See also the description of the Message Watcher in Chapter 3, “The Scripting Environment.”

For more information about creating a custom message watcher XCMD, see Appendix A, “External Commands and Functions.”
CHAPTER 12

Properties

Multiple

APPLIES TO

Painting environment

SYNTAX

set [the] multiple to boolean

Boolean is an expression that yields either true or false.

EXAMPLES

set multiple to true
set multiple to false

DESCRIPTION

You use the multiple property to determine or to change the Draw Multiple setting. When multiple is true, multiple images are drawn as you drag a shape tool.

Tools affected by the multiple property are the Line, Rectangle, Rounded Rectangle, Oval, and Regular Polygon tools. The default value of the multiple property is false.

NOTE

You can also set the multiple property by choosing Draw Multiple from the Options menu. Setting multiple to true puts a checkmark next to the Draw Multiple item in the Options menu. See also the multiSpace property, described later in this chapter.
MultipleLines

APPLIES TO
Fields

SYNTAX

set [the] multipleLines of field to boolean

Field is an expression that yields any valid field descriptor. Boolean is an expression that yields either true or false.

EXAMPLE

set the multipleLines of card field 1 to true

DESCRIPTION
You use the multipleLines property to determine or change whether multiple-line selections are allowed in the field when it is configured as a list field (that is, when its autoHilite, lockText, and dontWrap properties are true).

NOTES
You can set the multipleLines property from a script or by clicking the Multiple Lines checkbox in the Field Info dialog box.
CHAPTER 12
Properties

MultiSpace

APPLIES TO
Painting environment

SYNTAX

```
set [the] multiSpace to number
```

*Number* is an expression that yields any positive integer.

EXAMPLE

```
set multiSpace to 12
```

DESCRIPTION

You use the *multiSpace* property to determine or to change the amount of space left between edges of the multiple images drawn by the shape tools when the *multiple* property is *true*. The default value of the *multiSpace* property is 1.

SCRIPT

The following script uses the *multiSpace* and *multiple* properties to create an interesting image. It also cleans up after it is completed.

```plaintext
on roundyRound
  doMenu "New Card"
  reset paint
  choose oval tool
  set multiple to true
  set dragSpeed to 200
  set multiSpace to 15
  put "60,25" into upperLeft
  put "260,175" into botRight
```

438 MultiSpace
CHAPTER 12

Properties

put "260,25" into upperRight
put "60,175" into botLeft
drag from upperLeft to botRight with shiftKey
drag from botRight to upperLeft with shiftKey
drag from botLeft to upperRight with shiftKey
drag from upperRight to botLeft with shiftKey
wait 60 ticks
set lockScreen to true
choose browse tool
play boing
doMenu "Delete Card"
doMenu "Back"
set lockScreen to false
end roundyRound

Name

APPLIES TO

Backgrounds, buttons, cards, fields, HyperCard, menus, menu items, stacks

SYNTAX

set [the] name of object to objectName
set [the] name of menuItem of menu to itemName
[the] [english] name of menuNumber

Object is an expression that yields a valid background, button, card, field, or stack descriptor. ObjectName is an expression that yields any valid HyperCard object name. The object name can be a maximum of 31 characters.

MenuItem is an expression that yields a menu item descriptor that is in the menu menu. Menu is an expression that yields a menu descriptor. ItemName is an expression that yields the new text to replace the current menu item name. MenuNumber is an expression that yields the number form of a menu descriptor.
CHAPTER 12
Properties

EXAMPLES
put the english name of menu 1
put the english name of menuitem 2 of menu 3
set name of this stack to "TooHip"
put the long name of this stack into field 2
set name of this bkgnd to "TrueGrit"
put the long name of this background into field 3
put the long name of field 3 into msg
set name of menuItem "Dogs" of menu "Animals" to "Canines"
put the name of the third menuItem of menu "Direction"
put the name of menu 8
put the name of the third menu

DESCRIPTION
You use the name property to determine or to change the name of the specified background, button, card, field, stack, or menu item. You can use the name property to determine the name of a menu, but not to set the name. A stack name must be a valid Macintosh filename.

When you use the name of menuNumber form of the name property to get the name of a menu, the descriptor for the menu is an expression that yields a valid number of one of the current HyperCard menus or custom menus in the menu bar. Menus are numbered from left to right, starting with number 1 for the Apple menu.

Using the adjective english in conjunction with the name property ensures that you can correctly refer to menu items even after they have been localized.

NOTES
If you try to retrieve an object’s name when it has none, HyperCard returns its ID number.

If you try to modify or determine the name property of a menu item that does not exist, HyperCard displays a “No such menu item” dialog box.

If you try to modify or determine the name property of a menu that does not exist, HyperCard displays a “No such menu” dialog box.
You can set the name of the About HyperCard menu item in the Apple menu to a different value.
See also the put command in Chapter 10, “Commands.”

Number

APPLIES TO
Backgrounds, buttons, cards, fields, windows

SYNTAX
put [the] number of object [into container]

Object is an expression that yields any valid background, button, card, or field descriptor. Container is the selection, any field, a variable, or the Message box.

EXAMPLES
if the number of this bkgnd is 2 then go next card
put number of last card into msg

DESCRIPTION
You use the number property to determine the number of any background, button, card, or field in the current stack.
You can’t set the number of a background, button, card, or field; the number of an object changes when you add or delete a background, a button, a card, or a field. The number of a field or button may also change if you change its partNumber property.

NOTE
See also the number function in Chapter 11, which returns the count (how many) of various elements, not the descriptor number of an individual object.
NumberFormat

APPLIES TO
Global environment

SYNTAX
set numberFormat to formatType

FormatType is an expression that yields the format (within quotation marks) that is to be used for the display of numbers.

EXAMPLES
set numberFormat to "00.##" -- display 02.21
set numberFormat to "0" -- display 2 for the same value
set numberFormat to "0." -- display 2.2

DESCRIPTION
The numberFormat property determines the precision with which the results of mathematical operations are displayed in fields and the Message box. Use zeros to show how many digits you want to appear, a period to show where you want the decimal point (if at all), and number signs (#) to the right of the decimal point in places where you want a trailing digit to appear, but only if it has value. Use zeros to the right of the decimal point if you always want the same number of digits to show, whether or not they have value. HyperTalk does calculations with up to 19 digits of accuracy.

HyperCard resets the numberFormat property to its default value, "0.########", at idle time (in effect, at the end of all pending handlers).
NOTE

The `numberFormat` property has no effect on how a number is displayed unless you perform a mathematical operation on it first (for details, see Chapter 6, “Values”).

Owner

APPLIES TO

Card or window

SYNTAX

```
put [the] owner of window|card
```

*Window* is the name, ID, or layer number of a window, and *card* is the name, ID, or positional number of a card in the current stack.

EXAMPLES

```
put the owner of window 8
put the owner of card "Introduction"
```

DESCRIPTION

You can ask HyperCard to return the owner of either a card or a window. For a window, this read-only property returns the name of the entity that created the window. This might be HyperCard itself (as in the case of a stack window) or the name of an XCMD like Picture, Message Watcher, or Variable Watcher. The owner of a card is the name or ID of the background that card shares.
CHAPTER 12

Properties

PartNumber

APPLIES TO

Button or field

SYNTAX

put [the] partNumber of button|field
set [the] partNumber of button|field to number

Button is an expression yielding a button identifier, field is an expression yielding a field identifier, and number is a positive integer that is less than or equal to the number of parts in the enclosing background or card.

EXAMPLES

put the partNumber of bg btn "StackKit"
put the partNumber of bg fld "Card Title"
set the partNumber of button "Apple Event Primer" to 1
-- sends that object back

DESCRIPTION

The partNumber property returns a number representing the order in which a button or field was placed in its enclosing object. For example, the order of buttons and fields within a card might be card field 1, card field 2, card button 1, card field 3. Even though the number of card button 1 is 1, it’s actually in the third position within its enclosing object.

You can use partNumber to reset the ordering of parts within a background or card. A smaller number than the object’s original partNumber sends the object back, and a larger number brings it forward.
CHAPTER 12

Properties

Pattern

APPLIES TO
Painting environment

SYNTAX
set [the] pattern to number

*Number* is an expression that yields a positive integer in the range 1 to 40, each representing a pattern in the Patterns palette. The patterns are shown in Figure 12-3.

EXAMPLES
set pattern to 12
put the pattern

DESCRIPTION
You use the *pattern* property to determine or change the current pattern used to fill shapes (including Paint text) and to paint with the Brush tool.

**Figure 12-3** Patterns palette and pattern numbers
The pattern numbers correspond to the 40 positions in the Patterns palette, not to a specific pattern.

**NOTE**

You normally set the `pattern` property from the Patterns palette. You can edit a pattern by double-clicking it on the Patterns palette.

---

**PolySides**

**APPLIES TO**

Painting environment

**SYNTAX**

```plaintext
set [the] polySides to number
```

*Number* is an expression that yields a positive integer between 3 and 50. This integer is the number of sides in the polygon.

**EXAMPLE**

```plaintext
set polySides to 12
```

**DESCRIPTION**

You use the `polySides` property to determine or to change the number of sides of the polygon created by the Regular Polygon tool. The default value is 4. If you set the value of `polySides` to a number lower than 3 or higher than 50, it automatically reverts to 3 or 50, respectively. If you choose the circle in the Polygon Sides dialog box, the setting becomes 0.

**NOTE**

You normally choose the Polygon Sides setting from a dialog box invoked by choosing Polygon Sides from the Options menu or by double-clicking the Regular Polygon tool.
CHAPTER 12

Properties

PowerKeys

APPLIES TO

Global environment

SYNTAX

set powerKeys to boolean

Boolean is an expression that yields either true or false.

EXAMPLES

set powerKeys to true
set powerKeys to false

DESCRIPTION

You use the powerKeys property to provide keyboard shortcuts for painting actions. The availability of power keys is usually set on the User Preferences card of the Home stack.

The default setting is determined at startup and resume time by the setting on the User Preferences card of the Home stack.

NOTE

Setting powerKeys to true in a script puts a checkmark next to the Power Keys command in the Options menu and changes the setting on the User Preferences card.
CHAPTER 12

Properties

PrintMargins

APPLIES TO

Global environment

SYNTAX

set [the] printMargins to rectangle

Rectangle is an expression that yields two points, reported as four positive integers separated by commas.

EXAMPLES

set the printMargins to 78,78,340,440

the printMargins -- puts current margins in Msg box

DESCRIPTION

You use the printMargins property to determine or change the current margin of the print area to be used when printing an expression. It may also affect page margins when printing a field. The default value for the printMargins property is 0,0,0,0. The value of the printMargins property represents the top-left and bottom-right corners of the printing area.

NOTE

See also the printTextAlign, printTextFont, printTextHeight, printTextSize, and printTextStyle properties described in this chapter and the print and reset printing commands in Chapter 10.
CHAPTER 12

Properties

PrintTextAlign

APPLIES TO

Global environment

SYNTAX

set [the] printTextAlign to alignment

Alignment is an expression that yields right, left, or center.

EXAMPLES

set the printTextAlign to Right
the printTextAlign -- puts current alignment in Msg box

DESCRIPTION

You use the printTextAlign property to determine or change the current alignment to be used when printing the contents of a variable or when you want to modify the text alignment in the header of a print report job. The default value for the printTextAlign property is left.

NOTE

See also the printMargins, printTextFont, printTextHeight, printTextSize, and printTextStyle properties described in this chapter and the print and reset printing commands in Chapter 10.
CHAPTER 12

Properties

PrintTextFont

APPLIES TO

Global environment

SYNTAX

set [the] printTextFont to font

Font is an expression that yields a valid font name in the current Macintosh system or the name of a font in a font resource installed in HyperCard or in the current stack.

EXAMPLES

set the printTextFont to "Palatino"
the printTextFont -- puts current font in Msg box

DESCRIPTION

You use the printTextFont property to determine or change the current font to be used when printing the contents of a variable or when you want to modify the font used in the header of a print report job. The default value for the printTextFont property is Geneva.

NOTE

See also the printMargins, printTextAlign, printTextFont, printTextSize, and printTextStyle properties described in this chapter and the print and reset printing commands in Chapter 10.
CHAPTER 12
Properties

PrintTextHeight

APPLIES TO
Global environment

SYNTAX
set [the] printTextHeight to number

Number is an expression that yields a valid line height for a font in the current Macintosh system or a font in a font resource installed in HyperCard or in the current stack.

EXAMPLES
set the printTextHeight to 16
the printTextHeight -- puts current line height in MsgBox

DESCRIPTION
You use the printTextHeight property to determine or change the space between baselines of text to be used when printing the contents of a variable or when you want to modify the line height of the text in the header of a print report job. The default value for the printTextHeight property is 13.

NOTE
See also the printMargins, printTextAlign, printTextFont, printTextSize, and printTextStyle properties described in this chapter and the print and reset printing commands in Chapter 10.
**PrintTextSize**

**APPLIES TO**

Global environment

**SYNTAX**

```
set [the] printTextSize to number
```

*Number* is an expression that yields an integer that represents a valid font size in the current Macintosh system or the size of a font in a font resource installed in HyperCard or in the current stack.

**EXAMPLES**

```
set the printTextSize to 12
the printTextSize -- puts current text size in Msg box
```

**DESCRIPTION**

You use the `printTextSize` property to determine or change the current size of the font when printing the contents of a variable or when you want to modify the size of the text in the header of a print report job. The default value for the `printTextSize` property is 10.

**SCRIPT**

This script sets some of the printing properties and prints the contents of card field 1, which is put in the variable `PJob`:

```plaintext
on printField
    put card field 1 into PJob
    set the printTextFont to "New York"
    set the printTextStyle to "Outline"
    set the printTextSize to "12"
    print PJob
end printField
```
NOTE

See also the printMargins, printTextAlign, printTextFont, printTextHeight, and printTextStyle properties described in this chapter and the print and reset printing commands in Chapter 10.

PrintTextStyle

APPLIES TO

Global environment

SYNTAX

set [the] printTextStyle to style

*Style* is an expression that yields a valid font style in the current Macintosh system or the style of a font in a font resource installed in HyperCard or in the current stack. Valid HyperCard font styles are bold, condense, extend, italic, outline, plain, and underline.

EXAMPLES

set the printTextStyle to bold
the printTextStyle -- puts current font style in Msg box

DESCRIPTION

You use the printTextStyle property to determine or change the current style of the font when printing the contents of a variable or when you want to modify the style of the text in the header of a print report job. The default value for the printTextStyle property is plain. The group style that is available in the HyperCard Style menu does not apply to printing.

NOTE

See also the printMargins, printTextAlign, printTextFont, printTextHeight, and printTextSize properties described in this chapter and the print and reset printing commands in Chapter 10.
CHAPTER 12

Properties

Rect

APPLIES TO
Variable and message watcher windows, picture windows, card windows, script windows

SYNTAX

set rect of window variableWatcher to location
set rect of window name to location

VariableWatcher is an expression that yields the name of a variable watcher window. Location is an expression that yields two points, reported as four positive integers separated by commas. Name is an expression that yields the name of a picture or stack window.

EXAMPLES

set rect of window "Variable Watcher" to "0,0,168,185"
put rect of window "MyWatcher" into msg

DESCRIPTION

The rect property is two points, reported as four integers separated by commas. You use the rect property to determine or change the size of the variable watcher window.

The points represent the rectangle’s top-left (horizontal and vertical) and bottom-right (horizontal and vertical) corner offsets in pixels, respectively, from the top-left corner of the variable watcher window. The first point is always 0,0, and the second point is the offset from the first point.

NOTES

Rect is also the abbreviated form of the rectangle property and works on all of the objects and windows that the rectangle property works on.
CHAPTER 12

Properties

Properties that work on HyperCard’s built-in external windows may not work on custom external windows. It is the responsibility of the creator of the custom window to provide support for HyperTalk external window properties. See also the variableWatcher, hBarLoc, and vBarLoc properties in this chapter.

Rectangle

APPLIES TO

Buttons, cards, fields, menu bar, windows

SYNTAX

set [the] rect[angle] of object to location

Object yields one of the following:

a valid button descriptor in the current stack
a valid field descriptor in the current stack
message [box] or message [window] or window "message"
pattern window or window "patterns" (the Patterns palette)
tool window or window "tools" (the Tools palette)
window "navigator" (the Navigator palette)
scroll window or window "scroll"
window "Fatbits"
message watcher or window "message watcher"
variable watcher or window "variable watcher"
card window
window stackName
menubar

Location is an expression that yields two points, reported as four positive integers separated by commas. StackName is the name of a stack.
CHAPTER 12

Properties

EXAMPLES

put rectangle of menubar
set the rectangle of field 4 to 23,45,68,85
put rectangle of field "Sweet" into msg
put the rect of message box -- puts h,v,h,v into Msg
set rect of card window to 64,81,576,441
set the rect of this card to 0,0,512,360

DESCRIPTION

The rectangle property is two points, reported as four integers separated by commas. You use the rectangle property to set or determine the size of buttons, fields, and windows. This property is a read-only property for the Message box, Tools palette, Patterns palette, Scroll window, and menu bar.

The points represent the rectangle’s top-left (horizontal and vertical) and bottom-right (horizontal and vertical) corner offsets in pixels, respectively, from the top-left corner of the card. The offsets for card windows and menu bar measure from the top-left corner of the screen.

You can set either of the rectangle points of a field or button beyond the boundaries of the card rectangle, putting the field or button out of view until you reset its coordinates through HyperTalk.

You can set the bottom-right corner location of a button or field to a value smaller than the top-left corner location, effectively causing the button or field to disappear. If you set a field to a size smaller than the minimum (12 by 12 pixels) but large enough to see, HyperCard resets it to the minimum size when you click it with the corresponding tool.

You can also change a button or field rectangle by dragging the top-left or bottom-right corner of the button or field with the appropriate tool selected (Button or Field).
CHAPTER 12

Properties

SCRIPT

The following example handler, placed in a button script, is invoked when you click the button. It waits until you move the pointer outside the button rectangle, then beeps when you move the pointer back inside the button rectangle:

```
on mouseUp
    wait until the mouseLoc is not within rect of me
    repeat until the mouseLoc is within rect of me
        set cursor to busy -- spin beach ball while we wait
    end repeat
    beep
end mouseUp
```

NOTES

The rectangle property can be abbreviated rect. The four integers that make up the rectangle property can also be changed individually and in various combinations. See the descriptions of bottom, bottomRight, height, left, right, top, topLeft, and width in this chapter. These are known collectively as rectangle properties and are summarized in Table 12-6.

The rectangle property cannot be set for palettes or the built-in Message Watcher.

The rectangle property of the menu bar cannot be set.

The operator within pertains to any rectangle, such as the rectangles of buttons and fields, the Tools and Patterns palettes, the Message box, and the screen on which the HyperCard menu bar is displayed. The syntax of an expression in which within is valid is the following:

```
location is [not] within rectangle
```

Location is an expression that yields a list of two integers separated by a comma, and rectangle is an expression that yields a list of four integers separated by commas.
When you resize a card window with the `rectangle` property, a `sizeWindow` system message is sent. The `sizeWindow` message is also sent when you zoom a card window by clicking the zoom box or when you change its size with the `Scroll window` (Command-E).

If the `location` property of a card window changes when you set the `rectangle` property, a `moveWindow` system message is sent. The `moveWindow` message is also sent when you drag the window to a new location, zoom it in or out, or show the window at a new location with the `show` command.

See also the `location` property, earlier in this chapter.

---

**ReportTemplates**

**APPLIES TO**

Stacks

**SYNTAX**

```
put [the] reportTemplates of stack *stackName*
```

*StackName* is an expression that yields the name of a stack.

**EXAMPLES**

```
put reportTemplates of stack "People" into field 2
get the reportTemplates of stack "Forecast"
```

**DESCRIPTION**

The `reportTemplates` property is a read-only property of stacks. It returns a return-delimited list of the names of the report templates for the specified stack.
CHAPTER 12

Properties

NOTES

Report templates are created and saved for a stack with the Print Report command in the File menu. See the HyperCard Reference for more information about creating report templates.

See also the open report printing command in Chapter 10.

Right

APPLIES TO

Buttons, fields, windows, menu bar

SYNTAX

set [the] right of object to number

Object yields one of the following:

- a valid button descriptor in the current stack
- a valid field descriptor in the current stack
- message [box] or message [window] or window "message"
- pattern window or window "patterns" (the Patterns palette)
- tool window or window "tools" (the Tools palette)
- window "navigator" (the Navigator palette)
- scroll window or window "scroll"
- window "Fatbits"
- message watcher or window "message watcher"
- variable watcher or window "variable watcher"
- card window
- window stackName
- menubar

Number yields an integer that is the horizontal offset in pixels from the left side of the card to the right side of the object. When the object is the card window, the offset is relative to the left side of the screen. StackName is an expression that yields the name of an open stack.
EXAMPLES

set right of button 2 to 165
put right of button 2
put the right of the card window
set right of pattern window to 100

DESCRIPTION

You use the right property to determine or change the value of item 3 of the rectangle property (left, top, right, bottom) when applied to the specified object or window.

NOTE

The right of the menu bar is a read-only property.

See also the rectangle property, earlier in this chapter.

Script

APPLIES TO

Backgrounds, buttons, cards, fields, stacks

SYNTAX

set [the] script of object to scriptText

Object is the current background, button, card, field, or stack or any background, button, card, field, or stack name currently available to HyperCard. ScriptText yields any valid container that contains a script, or yields a text string that is a script.
EXAMPLES

set script of field "Effect" of first card to empty
set the script of second bgnd to empty
set the script of third card to field 3
put the script of stack "home" into field "Home Script"

DESCRIPTION

You use the script property to retrieve or to replace the script of the specified object. The value of the script property is the text string composing the script of the specified stack.

When you set the script property using the set command, you replace it entirely.

NOTE

Scripts are normally edited using the HyperCard script editor described in Chapter 3, “The Scripting Environment.”

ScriptEditor

APPLIES TO

Scripting environment

SYNTAX

set [the] scriptEditor to name

Name is an expression that yields a valid script editor XCMD name.
CHAPTER 12

Properties

EXAMPLES

set the scriptEditor to "MyEditor"
the scriptEditor -- puts current script editor in Msg box
put the scriptEditor after field "Editor in Use"

DESCRIPTION

You use the scriptEditor property to determine or change the current script editor. The default value for the scriptEditor property is scriptEditor, the name of the built-in script editor.

The built-in script editor is a HyperCard XCMD. It can be replaced with a custom script editor XCMD by setting the scriptEditor property to the name of a script editor XCMD.

NOTES

See also the scriptTextFont and scriptTextSize properties described later in this chapter and the description of the scripting environment in Chapter 3.

For more information about creating a custom script editor XCMD, see Appendix A, “External Commands and Functions.”

ScriptingLanguage

APPLIES TO

Buttons, fields, parts, cards, background, stacks, global environment

SYNTAX

set the scriptingLanguage [of object] to scriptingLanguage

Object is any background, button, card, field, part, or stack name currently available to HyperCard. ScriptingLanguage is a scripting language installed on the current system.
CHAPTER 12

Properties

EXAMPLE

put the scriptingLanguage of this cd
set the scriptingLanguage to AppleScript

DESCRIPTION

The scriptingLanguage property lets you set any of the HyperCard objects to accept scripts written in the scripting language of your choice, among those available in your system. The script editor has a pop-up menu that displays the available scripting languages.

NOTE

You can use the unary operator there is a to test for the existence of a scripting language capability on the system where HyperCard is running, using the syntax

there is a scriptingLanguage scriptingLanguage

The statement returns a Boolean value.

ScriptTextFont

APPLIES TO

Scripting environment

SYNTAX

set [the] scriptTextFont to font

Font is an expression that yields a valid font name in the current Macintosh system.
CHAPTER 12

Properties

EXAMPLES

set the scriptTextFont to "Palatino"
the scriptTextFont -- puts current script editor
        -- font in the Msg box

DESCRIPTION

You use the scriptTextFont property to determine or change the current
font in the script editor. The default value for the scriptTextFont property
is monaco.

NOTES

You can also set the scriptTextFont property in the dialog box invoked by
typing se into the Message box, using a handler provided in the standard
Home stack script.

See also the scriptEditor and scriptTextSize properties described in
this chapter and the description of the scripting environment in Chapter 3.

ScriptTextSize

APPLIES TO

Scripting environment

SYNTAX

set [the] scriptTextSize to number

Number is an expression that yields an integer that represents a valid font size
in the current Macintosh system.

EXAMPLES

set the scriptTextSize to 12
the scriptTextSize -- puts current script editor
        -- font size in the Msg box
CHAPTER 12

Properties

DESCRIPTION
You use the scriptTextSize property to determine or change the current size of the font used in the script editor. The default value for the scriptTextSize property is 9.

NOTES
You can also set the scriptTextSize property in the dialog box invoked by typing se into the Message box.
See also the scriptEditor and scriptTextFont properties described in this chapter and the description of the scripting environment in Chapter 3.

Scroll (fields)

APPLIES TO
Fields

SYNTAX
set [the] scroll of scrollingField to number

ScrollingField is any valid card or background scrolling field. Number is an expression that yields an integer representing the number of pixels that have scrolled above the top of the field rectangle.

EXAMPLES
set the scroll of field "Clues" to 0
put the scroll of field 1 div the textHeight of field 1¬ into linesAbove

DESCRIPTION
You use the scroll property to determine or to change how much material is hidden above the top of a scrolling field’s rectangle. Figure 12-4 illustrates the scroll property.
The value of the scroll property is 0 if the top of the field is visible. The number of text lines to which the scroll property correlates depends on the textHeight property of the field.

NOTES

You normally control how much material is above the top of the field rectangle by clicking or dragging in the scroll bar at the right side of the field.

If you try to get or set the scroll property of a nonscrolling field, you get an error dialog box.

**Figure 12-4**  The scroll property

Scroll (windows)

APPLIES TO

Card windows, picture windows

466  Scroll (windows)
CHAPTER 12
Properties

SYNTAX

set [the] scroll of [the] card window to point
set [the] scroll of window to point

Point is an expression that yields two comma-separated positive integers that represent the point on the card or picture to be displayed at the top-left corner of the window. Window is an expression that yields a reference to a window created with the picture command or to a card window.

EXAMPLES

set the scroll of the card window to 45,60
put the scroll of card window into scrollVar
set the scroll of window "Home" to "0,100"

DESCRIPTION

You use the scroll property to determine or change the horizontal and vertical scroll (position) of the card window over the current card or the window over the current picture. This property allows you to scroll over a card that is larger than the area of the card window region. The default position of the scroll property for a card window is 0,0.

NOTES

The scroll property has no effect on card windows that are the same size as the card. Card windows can’t be larger than the card. You can reset the value of the scroll property to reposition the card window with the Scroll command in the Go menu. You cannot set the scroll property of a window that is displaying an inactive stack.

See also the rectangle property in this chapter and the picture command in Chapter 10.
CHAPTER 12

Properties

SharedHilite

APPLIES TO
Background buttons

SYNTAX

set [the] sharedHilite of button to boolean

*Button* is an expression that yields any valid background button descriptor. *Boolean* is an expression that yields either *true* or *false*.

EXAMPLES

set sharedHilite of bkgnd button "Flip card" to true
put the sharedHilite of bkgnd button 2 into msg

DESCRIPTION

You use the *sharedHilite* property to determine or to change whether the specified background button shares the same highlight state on all cards of that background. The default value for new background buttons is *true*. When *sharedHilite* is set to *true*, the background button has the same highlight state on all cards of that background.

NOTES

Background buttons have two sets of highlight states, one you see when the button's *sharedHilite* is *true* and one you see when its *sharedHilite* is *false*. If you have a background button with its *sharedHilite* set to *false*, that button on each of those cards of that background can have a different highlight state (determined by the *hilite* property). If you change the *sharedHilite* property to *true* on that background button, the highlight state of that button on all of the cards of that background is set to *false* (not highlighted). The unshared highlight states of that background button are not lost, however. The unshared highlight states are stored separately with each card and can be returned to their previous values by setting the *sharedHilite* property back to *false*. 

468 SharedHilite
Background buttons copied and pasted to other cards have the same `sharedHilite` value as the button originally copied.

You can also change the `sharedHilite` property by clicking the Shared Hilite checkbox in the Button Info dialog box. See also the descriptions of the `autoHilite` and `hilite` properties, earlier in this chapter.

**SharedText**

**APPLIES TO**

Background fields

**SYNTAX**

`set [the] sharedText of field to boolean`

*Field* is an expression that yields any valid background field descriptor. *Boolean* is an expression that yields either `true` or `false`.

**EXAMPLE**

`set the sharedText of field 3 to true`

**DESCRIPTION**

You use the `sharedText` property to determine or to change whether the text in the specified background field appears on each card of that background. When the value of `sharedText` is `true`, the text in the specified background field is shared by all cards of that background. When it is `false`, the text can be different in the specified field on all the cards of that background. The default value of the `sharedText` property for new background fields is `false`.

**NOTES**

A background field with its `sharedText` property set to `true` effectively has its `dontSearch` property set to `true`. The `find` command excludes the field from any searches.
CHAPTER 12

Properties

If you change the `sharedText` property to `true` on a background field that previously had the `sharedText` property set to `false` (unshared text), no text is displayed in that field on any of the cards of that background. The previous unshared text of that background field on each of the cards with that background is not lost, however. The unshared text is stored separately with each card and can be redisplayed in the background field of those cards with that background by setting the `sharedText` property back to `false`.

You can also change `sharedText` by clicking the Shared Text checkbox in the Field Info dialog box.

ShowLines

APPLIES TO

Fields

SYNTAX

```
set [the] showLines of field to boolean
```

*Field* is an expression that yields any valid field descriptor. *Boolean* is an expression that yields either `true` or `false`.

EXAMPLES

```
set the showLines of field four to true
put the showLines of card field 1 into msg
```

DESCRIPTION

You use the `showLines` property to determine or to change whether the text baselines in the card or background field show or not. The default value of the `showLines` property is `false` (lines don’t show).
NOTES

You can also change `showLines` by clicking in the Show Lines checkbox in the Field Info dialog box. The `showLines` property does not apply to scrolling fields.

ShowName

APPLIES TO

Buttons

SYNTAX

```
set [the] showName of button to boolean
```

*Button* is an expression that yields any valid background or card button descriptor. *Boolean* is an expression that yields either *true* or *false*.

EXAMPLES

```
set showName of button "You who" to true
put the showName of button "You who" into msg
```

DESCRIPTION

You use the `showName` property to determine or to change whether the name of the specified button (if it has one) is displayed in its rectangle on the screen. Buttons created with the New Button command have `showName` set to *true*. Buttons created by Command-dragging the button tool have their `showName` property initially set to *false*.

NOTE

You can also change this property by clicking the Show Name checkbox in the Button Info dialog box.
ShowPict

APPLIES TO
Cards, backgrounds

SYNTAX

set [the] showPict of object to boolean

*Object* is an expression that yields any valid background or card descriptor. *Boolean* is an expression that yields either *true* or *false*.

EXAMPLES

set showPict of next card to false
set the showPict of this bkgnd to false
put showPict of bkgnd 3

DESCRIPTION

You use the *showPict* property to determine or to change whether the picture on the specified card or background (if it has one) is displayed on the screen. The default value is *true* (displayed).

Setting the *showPict* property of a card to *false* is the same as hiding it with the *picture* form of the *hide* command. Setting it to *true* is the same as showing it with the *picture* form of the *show* command.

NOTES

When the *showPict* property of the current background or card is *false* and you try to use a Paint tool on it manually, a dialog box appears asking if you want to make the picture visible; clicking OK sets the *showPict* property to *true* and the picture appears. You can draw on hidden pictures from a script.

See also the *show* and *hide* commands in Chapter 10.
Size

APPLIES TO
Stacks

SYNTAX
put [the] size of stack stackName [into container]

StackName is the current stack, or an expression that yields any stack name currently available to HyperCard. Container is an expression that yields any valid container.

EXAMPLES
get size of stack "Home"
put size of stack "Home" into field "Home Size"

DESCRIPTION
You use the size property to determine the size of the specified stack in bytes.

SCRIPT
The following handler examines a stack to see if it fits on an 800 KB disk:

on closeStack
  if size of this stack > 795000
    then answer "This stack won't fit on an 800 KB disk."
  pass closeStack
end closeStack

NOTES
The minimum stack size is 4096 bytes; the theoretical maximum is 512 MB.
The size property can’t be changed with the set command; it’s changed only by adding things to and deleting things from the stack (you must then compact the stack for any deletions to affect its size).

**StacksInUse**

**APPLIES TO**

Global environment

**SYNTAX**

```
put [the] stacksInUse [into container]
```

*Container* is an expression that yields any valid container.

**EXAMPLES**

```
the stacksInUse -- puts stack list in Msg box
put the stacksInUse into field "Message Path"
```

**DESCRIPTION**

You use the stacksInUse property to determine the stacks that have been added to the current message-passing hierarchy. StacksInUse returns a return-delimited list of the stacks in the current message-passing hierarchy. The stacks are listed in the order in which they are currently placed in the hierarchy.

The stacksInUse property can’t be changed with the set command; the stacks in the current message-passing hierarchy can only be changed with the start using and stop using commands.

**NOTES**

See also the start using and stop using commands in Chapter 10 and the description of the message-passing hierarchy in Chapter 4.

If the message-passing hierarchy hasn’t been altered with the start using command, stacksInUse returns empty.
CHAPTER 12

Properties

Style

APPLIES TO

Buttons, fields

SYNTAX

set [the] style of object to objectStyle

Object is an expression that yields any valid button or field descriptor. ObjectStyle is an expression that yields one of the valid field or button styles. Button styles are transparent, opaque, rectangle, roundRect, shadow, checkBox, radioButton, standard, default, oval, and popup. Field styles are transparent, opaque, rectangle, shadow, and scrolling.

EXAMPLES

set the style of field 1 to scrolling
set style of button "You who" to roundRect
put the style of button 3 into msg
set the style of field 2 of card 4 to transparent

DESCRIPTION

You use the style property to determine or to change the style of any button or field in the current stack.

NOTE

You can also change the button or field style by using the Style pop-up menu in the Button or Field Info dialog box. Some useful peculiarities of radio buttons and checkbox buttons are described under the hilite property, in this chapter.
CHAPTER 12

Properties

Suspended

APPLIES TO
Global environment

SYNTAX
the suspended

DESCRIPTION
The suspended property returns whether or not HyperCard is currently running in the background under MultiFinder or System 7. A user can switch from HyperCard to another program while a handler is running and scripts will continue to run in the background.

Use the suspended property in a handler to alter the handler’s behavior if it’s running in the background—for example, to avoid displaying ask or answer dialog boxes.

HyperCard gives time to the system (and thus to other programs)

■ after it executes each HyperTalk statement in a handler,
■ whenever it rotates the busy cursor (during compacting, sorting, and printing),
■ during the execution of the show cards command and the wait command.

EXAMPLE

if not (the suspended) then
    -- Show dialog when not running in background
    ask file "Save as what file?"
    put it into theFileName
else
    -- We're in the background, use a default name
    put "Untitled" into theFileName
end if

Suspended
CHAPTER 12

Properties

** TextAlign**

**APPLIES TO**

Buttons, fields, painting environment

**SYNTAX**

```
set [the] textAlign [of object] to alignment
```

*Object* is an expression that yields a button or field descriptor. *Alignment* is an expression that yields one of the words *left*, *right*, and *center*.

**EXAMPLES**

```
set the textAlign of field 1 to left
set textAlign to center -- for paint text
```

**DESCRIPTION**

You use the `textAlign` property to determine or to change the way characters are aligned around the insertion point as you type them. This property applies to Paint text, button name text, and the text in fields. The default value of the `textAlign` property is *left* for fields and Paint text; the default value is *center* for buttons.

**NOTES**

For Paint text, you can also set the `textAlign` property from the Font dialog box, which is invoked by choosing Text Style from the Edit menu, by double-clicking the Paint Text tool, or by pressing Command-T when the Paint Text tool is selected.

For buttons or fields, you can also set the `textAlign` property by choosing one of the text alignment options from the Text Properties dialog box. To invoke the Text Properties dialog box, you click the Text Style button in the Button or Field Info dialog box, choose Text Style from the Edit menu, or press Command-T when a button or field is selected.

See also the `printTextAlign` property, earlier in this chapter.
CHAPTER 12

Properties

TextArrows

APPLIES TO
Global environment

SYNTAX

set textArrows to boolean

Boolean is an expression that yields either true or false.

EXAMPLES

set textArrows to true
set textArrows to false

DESCRIPTION

The textArrows property alters the function of the Right Arrow, Left Arrow, Up Arrow, and Down Arrow keys.

The default value of the textArrows property is false. When the textArrows property is false, the Right Arrow and Left Arrow keys take you to the next and previous cards in the stack, respectively, and the Up Arrow and Down Arrow keys take you forward and backward, respectively, through the cards you’ve already viewed.

When the textArrows property is true, the arrow keys move the text insertion point around in a field that you’ve opened for text editing or in the Message box if you’ve clicked it. In the Message box, the Up Arrow and Down Arrow keys move the insertion point to the beginning and end of the line of text, respectively.

NOTE

When the textArrows property is true, holding down the Option key while you press the arrow keys produces the same effect as pressing them alone when textArrows is false.
CHAPTER 12

Properties

TextFont

APPLIES TO

Buttons, fields, painting environment

SYNTAX

set [the] textFont [of chunk] of field to font
set [the] textFont [of object] to font

Chunk is any valid chunk expression. Field is an expression that yields a field descriptor. Object is an expression that yields a button or field descriptor. Font is an expression that yields one of the font names available in your Macintosh system.

EXAMPLES

set textFont of field 1 to "courier"
set the textFont of bkgnd button 3 to helvetica
set textFont to Palatino -- for paint text

DESCRIPTION

You use the textFont property to determine or to change the font in which text appears. This property applies to button name text, the text in fields, and Paint text. The default value of the textFont property is geneva for fields and Paint text; the default value for buttons is chicago.

NOTES

For Paint text, you can also set the textFont property from the Font dialog box, which is invoked by choosing Text Style from the Edit menu, by double-clicking the Paint Text tool, or by pressing Command-T while using a Paint tool.

For buttons or fields, you can also set this property by choosing one of the font names from the Font menu or from the Font dialog box. To invoke the Font...
dialog box, you click the Font button in the Field or Button Info dialog box, choose Text Style from the Edit menu, or press Command-T while a button or field is selected.

If you reset the default font for a field with the textFont property, any text that is already in that field is updated. If you try to set the textFont property to a font that doesn’t exist, HyperCard sets it to geneva.

If different fonts are in a chunk of a field, textFont returns the result mixed.

See also the selectedChunk and selectedLine functions in Chapter 11, and the printTextFont and scriptTextFont properties, earlier in this chapter.

### TextHeight

**APPLIES TO**

Buttons, fields, painting environment

**SYNTAX**

```
set [the] textHeight [of object] to number
```

*Object* is an expression that yields a button or field descriptor. *Number* is an expression that yields any positive integer.

**EXAMPLES**

```
set textHeight of field 1 to 20
set textHeight to 20 -- for paint text
```

**DESCRIPTION**

You use the textHeight property to determine or to change the space between baselines of button text, field text, and Paint text. The value of the textHeight property is in pixels.
NOTES

For Paint text, you can also set the `textHeight` property in the Line Height box of the Font dialog box, which is invoked by choosing Text Style from the Edit menu, by double-clicking the Paint Text tool, or by pressing Command-T when a Paint tool is selected.

For buttons or fields, you can also set this property by typing the line height in the Line Height box in the Font dialog box. To invoke the Font dialog box, you click the Font button in the Field or Button Info dialog box, choose Text Style from the Edit menu when the field or button is selected, or press Command-T when a button or field is selected.

Although you can set this property for a button, it is meaningless because button-name text has only one line. See also the `fixedLineHeight` and `printTextHeight` properties described earlier in this chapter.

**TextSize**

**APPLIES TO**

Buttons, fields, painting environment

**SYNTAX**

```
set [the] textSize [of chunk] of field to number
set [the] textSize [of object] to number
```

*Chunk* is any valid chunk expression. *Field* is an expression that yields a field descriptor. *Object* is an expression that yields a button or field descriptor. *Number* is an expression that yields any positive integer.

**EXAMPLES**

```
set textSize of field 1 to 18
set the textSize of word 3 of line 4 to 12
set textSize to 18 -- for paint text
```
DESCRIPTION

You use the `textSize` property to determine or to change the font size in which text appears on the screen. The `textSize` property applies to button text, text in fields, and Paint text. The value of the `textSize` property is in pixels. The default value of the `textSize` property is 12.

Although you can use any integer for `textSize`, exact sizes of fonts available look best. Fonts available are in the Macintosh system or in the font resources in the current stack, a stack in use, the Home stack, or HyperCard.

NOTES

For Paint text, you can also set the `textSize` property from the Font dialog box, which is invoked by choosing Text Style from the Edit menu, by double-clicking the Paint Text tool, or by pressing Command-T while using a Paint tool.

For buttons or fields, you can also set this property from the Style menu or by selecting one of the font sizes shown or typing directly in the size box in the Font dialog box. To invoke the Font dialog box, you click the Font button in the Field or Button Info dialog box, choose Text Style from the Edit menu while a button or field is selected, or press Command-T while a button or field is selected.

If you reset the default text size for a field with the `textSize` property, any text that is already in that field is updated.

If different sizes of text are in a text selection, `textSize` returns the result mixed.

See also the `printTextSize` and `scriptTextSize` properties, earlier in this chapter.

TextStyle (buttons, fields, painting environment)

APPLIES TO

Buttons, fields, painting environment
SYNTAX

set [the] textStyle [of chunk] of field to style
set [the] textStyle [of object] to style

Chunk is any valid chunk expression. Field is an expression that yields a field descriptor. Object is an expression that yields a button or field descriptor. Style is an expression that yields a value of plain or any combination of the following: bold, italic, underline, outline, shadow, condensed, extend, and group (separated by commas).

EXAMPLES

set textStyle to plain -- for paint text
set textStyle to bold, italic, underline -- for paint text
set textStyle of field 1 to plain
set the textStyle of line 1 of field 1 to bold, group
set the textStyle of the first card field to bold

DESCRIPTION

You use the textStyle property to determine or to change the style in which text appears. The textStyle property applies to button text, text in fields, and Paint text. Its default value is plain. If you use plain in combination with any of the other values, the other values override plain.

NOTES

You use the group text style to group characters, words, or lines together so they are seen as a unit by HyperTalk. The group style does not apply to Paint text or button text.

Group text is supported through the mouseDown and mouseUp messages that are sent to locked fields when clicked and through three functions: clickChunk, clickLine, and clickText.

Here's an example in which you might use group text. You have a field with a list containing George Washington, King George, and George Bush, and you want to display more information about the appropriate George on the screen when his name is clicked. If these three phrases are set to plain text, clicking...
“George” wouldn’t be specific enough, because HyperTalk’s `clickText` function would only return the single word George, without specifying more information about which George was clicked. If you set the style of each of the phrases George Bush, George Washington, and King George to `group`, then when the user clicks any word in the group phrase, the person’s full name is returned and can be analyzed. If the user clicks either George or Bush in the phrase George Bush, the whole phrase—not just the word the user clicked—is returned by the `clickText` function.

For Paint text, you can also set the `textStyle` property from the Font dialog box, which is invoked by choosing Text Style from the Edit menu, by double-clicking the Paint Text tool, or by pressing Command-T while using a Paint tool.

For buttons or fields, you can also set the `textStyle` property by choosing a style from the Style menu or in the Font dialog box. To invoke the Font dialog box, you click the Font button in the Field or Button Info dialog box, choose Text Style from the Edit menu, or press Command-T while a button or field is selected.

If you reset the `textStyle` property for a field, any text that is already in that field is updated to the specified style.

If different styles of text are within a text selection, `textStyle` returns the result mixed.

See also the `printTextStyle` property, earlier in this chapter, and the `clickChunk`, `clickLine`, `clickText`, `selectedChunk`, and `selectedLine` functions in Chapter 11.

**TextStyle (menu items)**

**APPLIES TO**

Menu items
CHAPTER 12

Properties

SYNTAX

set [the] textStyle of menuItem of menu to style

menuItem is an expression that yields a menu item descriptor. Menu is an expression that yields a menu descriptor. Style is an expression that yields a value of plain or any combination of the following: bold, italic, underline, outline, shadow, condensed, and extend (separated by commas).

EXAMPLES

set the textStyle of menuItem "Get Back" of ¬ menu "Direction" to "outline"
put the textStyle of menuItem "Get Back" of ¬ menu "Direction"

DESCRIPTION

You use the textStyle property to set or determine the text style of a specified menu item. The default value for the textStyle property is plain.

The textStyle property could be used with the checkMark property to indicate that a menu item has been chosen.

If you try to modify or determine the textStyle property of a menu item that does not exist, HyperCard displays a “No such menu item” dialog box.

NOTES

The text style of the menu items in the Font and Tools menu cannot be altered with the textStyle property.

See also the printTextStyle property, earlier in this chapter, and the create menu and put commands in Chapter 10, “Commands.”
CHAPTER 12

Properties

TitleWidth

APPLIES TO

Pop-up buttons

SYNTAX

set [the] titleWidth of button to number

(Button is an expression that yields a valid button descriptor. Number is the width of the title area of the button in pixels.)

EXAMPLE

set the titleWidth of last button to 65

DESCRIPTION

You use the titleWidth property to determine or change the width of the title area of a pop-up button. You can also adjust the width of the title area by using the mouse to drag the line separating the title area and the pop-up menu.

NOTE

See also the style property, earlier in this chapter.

Top

APPLIES TO

Buttons, fields, windows
CHAPTER 12

Properties

SYNTAX

set [the] top of object to number

Object yields one of the following:

a valid button descriptor in the current stack
a valid field descriptor in the current stack
message [box] or message [window] or window "message"
pattern window or window "patterns" (the Patterns palette)
tool window or window "tools" (the Tools palette)
window "navigator" (the Navigator palette)
scroll window or window "scroll"
window "Fatbits"
message watcher or window "message watcher"
variable watcher or window "variable watcher"
card window
window stackName
menubar

Number is an expression that yields an integer that is the vertical offset in pixels of the top of the specified object. StackName is an expression that yields the name of an open stack.

EXAMPLES

set top of button 2 to 65
put top of button 2
set top of tool window to 10

DESCRIPTION

You use the top property to determine or change the value of item 2 of the rectangle property (left, top, right, bottom) when applied to the specified object or window. The top property of an object can also be set to a value off the screen. Setting the top property of an object to a value off the screen may make the object seem as though it is hidden.

Top
NOTE

See also the rectangle property, earlier in this chapter.

TopLeft

APPLIES TO

Buttons, fields, windows

SYNTAX

set [the] topLeft of object to point

Object yields one of the following:

a valid button descriptor in the current stack
a valid field descriptor in the current stack
message [box] or message [window] or window "message"
pattern window or window "patterns" (the Patterns palette)
tool window or window "tools" (the Tools palette)
window "navigator" (the Navigator palette)
scroll window or window "scroll"
window "Fatbits"
message watcher or window "message watcher"
variable watcher or window "variable watcher"
card window
window stackName
menubar

Point is an expression that yields a list of two integers separated by a comma.
Point represents the horizontal and vertical offsets, respectively, in pixels from
the top-left corner of the card to the top-left corner of the specified object.
StackName is an expression that yields the name of an open stack window.
EXAMPLES

set topLeft of background button id 23 to 64, 30
put topLeft of window "scroll"
put the topLeft of tool window
set topLeft of message box to 150, 75

DESCRIPTION

You use the **topLeft** property to determine or change items 1 and 2 of the value of the **rectangle** property (left, top, right, bottom) when applied to the specified object or window. When you change the **topLeft** property of an object, the entire object moves, its width and height remaining the same.

NOTE

See also the **rectangle** property, earlier in this chapter.

TraceDelay

**APPLIES TO**

Global environment

**SYNTAX**

```
set [the] traceDelay to number
```

*Number* is an expression that yields 0 or a positive integer.

**EXAMPLES**

```
set traceDelay to 32
put the traceDelay
```
CHAPTER 12

Properties

DESCRIPTION
You use the traceDelay property to set or retrieve the value of the debugger’s trace rate. Setting the number value changes the number of ticks HyperCard waits between executing lines of HyperTalk while tracing. The default value for traceDelay is 0, the fastest trace rate.

UserLevel

APPLIES TO
Global environment

SYNTAX
set userLevel to number

Number is an expression that yields one of the valid user-level numbers, 1 through 5.

EXAMPLE
set userLevel to 5

DESCRIPTION
You use the userLevel property to set or retrieve the value of the current HyperCard user level. User levels give progressively more power to the user. The levels are 1 (Browsing), 2 (Typing), 3 (Painting), 4 (Authoring), and 5 (Scripting), as explained in the HyperCard Reference.

If you set the value of userLevel to a number lower than 1 or higher than 5, it automatically reverts to 1 or 5, respectively.

NOTES
You can invoke the Protect Stack dialog box from the File menu to impose a limit on the user level available in a stack. In that case, setting the user level
higher than the Protect Stack limit has no effect, although it generates no error
message. On leaving the protected stack, the user level in effect when the stack
was entered is restored.

If your stack script changes the value of userLevel, be sure to restore the
original value of userLevel when your stack closes.

UserModify

APPLIES TO
Global environment

SYNTAX
set userModify to boolean

Boolean is an expression that yields either true or false.

EXAMPLES
set userModify to true
set userModify to false

DESCRIPTION
The userModify property is a global property pertaining to HyperCard itself.
It controls whether or not a user can type into fields or use Paint tools on a
stack that has been write-protected. A stack is write-protected under any of the
following circumstances:

- The stack is on a CD-ROM.
- The stack is on a file server in a folder whose access privileges are set to
  Read Only.
- The Locked box is checked in the stack’s Get Info dialog box in the Finder’s
  File menu.
CHAPTER 12

Properties

- The stack is on a locked disk.
- “Can’t modify stack” is checked in the stack’s Protect Stack dialog box.

SCRIPT

The following openStack handler sets up HyperCard so that the stack can be used, even though it is locked:

```script
on openStack
    if the cantModify of this stack is true then
        set the userModify to true
    end if
    pass openStack
end openStack
```

NOTES

Changes can be made only to the level that the userLevel settings allow. Any changes made to the stack are disregarded when the stack is closed.

See also the cantDelete and cantModify properties, earlier in this chapter.

VariableWatcher

APPLIES TO

Global environment

SYNTAX

```
set [the] variableWatcher to name
```

*Name* is an expression that yields a valid variable watcher XCMD name.
EXAMPLES

set variableWatcher to "MyWatcher"
put the variableWatcher

DESCRIPTION

You use the variableWatcher property to determine or to change the current variable watcher. The default value for variableWatcher is variableWatcher, the built-in variable watcher. You display the current variable watcher with the show command or by setting the visible property of the variable watcher window to true.

The built-in variable watcher is a HyperCard XCMD. It can be replaced with a custom variable watcher XCMD by setting the variableWatcher property to the name of a custom variable watcher XCMD.

NOTES

See also the description of the Variable Watcher in Chapter 3, “The Scripting Environment.”

For more information about creating and calling a custom variable watcher XCMD, see Appendix A, “External Commands and Functions.”

VBarLoc

APPLIES TO

Variable watcher windows

SYNTAX

set [the] vBarLoc of window "variable watcher" to number

*Number* is an expression that yields a positive integer that represents the offset in pixels from the left side of the variable watcher window to the vertical bar in the window.
EXAMPLES

set the vBarLoc of window "variable watcher" to 123
put the vBarLoc of window "variable watcher"

DESCRIPTION

You use the vBarLoc property to determine or to change the current position of the vertical bar in the variable watcher window. The vertical bar separates the variable names from the actual values of the variables.

NOTES

The built-in variable watcher is a HyperCard XCMD. It can be replaced with a custom variable watcher XCMD by setting the variableWatcher property to the name of a variable watcher XCMD.

A custom variable watcher may or may not respond to the vBarLoc property. It is up to the variable watcher XCMD to provide support for variable watcher properties.

See also the description of the Variable Watcher in Chapter 3, “The Scripting Environment,” and the hBarLoc, rect, and variableWatcher properties in this chapter.

For more information about creating and calling a custom variable watcher XCMD, see Appendix A, “External Commands and Functions.”

Version

APPLIES TO

HyperCard, stacks

SYNTAX

the [long] version [of HyperCard]
the version of stack stackName

StackName is an expression that yields a stack name.
EXAMPLE

if the version > 1.0 then set textArrows to true

DESCRIPTION

The `version` property returns the version number of the HyperCard application currently running or the versions of HyperCard that created and modified a specified stack.

The `long version` returns an eight-digit number that represents the major revision number, minor revision number, and software state (development, alpha, beta, or final, plus the release number). Here are the values the numbers represent:

```
version xxyyzzrr
xx    major revision number
yy    minor revision number
zz    80 = final
      60 = beta
      40 = alpha
      20 = development
rr    release number
```

For example, 0200600E is version 2.0 beta engineering release, and 02008000 is version 2.0 final.

The `version of stackName` form returns a list of five comma-separated eight-digit numbers. The first four of these numbers are of the form described previously for the `long version`. They are, respectively the version of HyperCard used to create this stack, the version of HyperCard that last compacted this stack, the version of HyperCard that last modified the stack, and the version of HyperCard that first modified the stack. The last number is the date and time (in seconds) of the most recent save before the start of the current session. (You can use the `convert` command to change the seconds format into a date and time format.)
Visible

APPLIES TO
Buttons, fields, menu bar, windows

SYNTAX

set the visible of object to boolean

Object yields one of the following:

a valid button descriptor in the current stack
a valid field descriptor in the current stack
message [box] or message [window] or window "message"
pattern window or window "patterns" (the Patterns palette)
tool window or window "tools" (the Tools palette)
window "navigator" (the Navigator palette)
scroll window or window "scroll"
window "Fatbits"
message watcher or window "message watcher"
variable watcher or window "variable watcher"
card window
window stackName
menubar

Boolean is an expression that yields either true or false. StackName is an expression the yields the name of an open stack window.

EXAMPLES

if the visible of menubar is false
    then set the visible of menubar to true
set the visible of tool window to false
set the visible of window "variable watcher" to true
CHAPTER 12

Properties

DESCRIPTION

The `visible` property determines whether a button, field, menu bar, or window is shown or hidden on the screen.

The Tools and Patterns palettes become visible when you tear them off the menu bar; the Message box and the menu bar can be toggled between being visible and hidden by pressing Command-M and Command-Space bar, respectively.

SCRIPT

The script that follows could be used to show a hidden field that is used for making notes. Create a button and a background field with the `sharedText` property set to `false`. The script placed in the button would display a field named `Notes` with the `show` command based on the value of the `visible` property of the `Notes` field. It also hides the field if you click the button again:

```plaintext
on mouseUp -- button or field script to show a field
    if the visible of bkgnd field "Notes" then
        hide bkgnd field "Notes"
    else
        show bkgnd field "Notes"
    end if
end mouseUp

The next short script makes a field disappear after it has been made visible. The script, when placed in a locked field, sets the `visible` property of the field to `false` when the field receives a `mouseUp` message. Whenever a user clicks the visible field, it disappears:

```plaintext
on mouseUp -- field script to set visible property
    set the visible of me to not the visible of me
end mouseUp

NOTE

See also the `show` and `hide` commands in Chapter 10.
CHAPTER 12

Properties

WideMargins

APPLIES TO

Fields

SYNTAX

set [the] wideMargins of field to boolean

Field is an expression that yields a background or card field descriptor. Boolean is an expression that yields either true or false.

EXAMPLES

set wideMargins of field "just fine" to true
the wideMargins of field 1 -- puts value in Msg box

DESCRIPTION

You use the wideMargins property to specify whether some extra space is included at the left and right sides of each line in the field (to make the text easier to read). The default value of wideMargins is false.

NOTE

You can also change this property by clicking the Wide Margins checkbox in the Field Info dialog box.
CHAPTER 12

Properties

Width

APPLIES TO

Buttons, fields, cards, windows, menu bar

SYNTAX

set [the] width of object to number

Object is an expression that yields a valid button, field, or window descriptor. Number is an expression that yields a positive integer. The number value represents the total number of pixels in the horizontal width of the specified object.

EXAMPLES

set width of cd window to width of cd window div 2
-- actually shrinks the window for all cards in the current
-- stack because all cards in a stack share same window

put width of button 4 -- puts width in Msg box
the width of bkgnd field "phoneList"

DESCRIPTION

You use the width property to determine or change the horizontal distance in pixels occupied by the rectangle of the specified button, field, or window.

NOTES

The width property is read-only for the Message box and menu bar.
See also the rectangle property, earlier in this chapter.
CHAPTER 12

Properties

Zoomed

APPLIES TO

Windows

SYNTAX

Set [the] zoomed of window to boolean

Window is an expression that yields a window descriptor. Boolean is an expression that yields either true or false.

EXAMPLE

set the zoomed of window "home" to true

DESCRIPTION

You use the zoomed property to determine or change whether a window is set to its maximum size and centered on the screen, as when the user clicks its zoom box in its upper-right corner.
Appendixes
External Commands and Functions

This appendix describes the external command and function interface of HyperCard. In addition to general information about external commands and functions, this appendix contains specific information that requires a reading knowledge of 68000 assembly language, Pascal, or C to be understood. This appendix does not include information about how to write code, nor does it explain how to use a compiler or assembler to create an executable resource.

Definitions, Uses, and Examples

External commands and functions are extensions to the HyperTalk built-in command and function set. HyperCard includes interface procedures that make extending HyperTalk in this way convenient and practical for expert programmers.

XCMD and XFCN Resources

External commands (ex-commands, or XCMDs) and external functions (ex-functions, or XFCNs) are executable Macintosh code resources, written in a Macintosh programming language (such as Pascal, C, or 68000 assembly language), which are attached to the HyperCard application or a stack with a resource editor such as ResEdit. The resource type of an external command is 'XCMD', and the resource type of an external function is 'XFCN'.

An XCMD or XFCN is a compiled (or assembled) executable code module. After XCMDs or XFCNs have been created and attached to HyperCard or a stack, they’re called from HyperTalk in much the same way that built-in commands or user-defined message and function handlers are called. They also use the message-passing hierarchy in the same way.

An XCMD or XFCN resource has no header bytes; it is invoked by a jump instruction to its entry point. These resources are simpler than Macintosh drivers: they can’t have any global (or static) data, and they can’t be larger...
than 32 KB in size. (For more details about these restrictions, see “Guidelines for Writing XCMDs and XFCNs,” later in this appendix.)

For detailed information on Macintosh resources, see the chapter “Resource Manager” in Inside Macintosh: More Macintosh Toolbox, published by Addison-Wesley.

**Uses for XCMDs and XFCNs**

External commands and functions can provide access to the Macintosh Toolbox and to some of HyperCard’s own internal routines; they can provide fast processing speed for time-critical operations; and they can override built-in HyperTalk commands to provide custom solutions. XCMDs or XFCNs can be used for serial port input and output routines, custom search-and-replace routines, AppleCD SC control routines, file input and output routines, and so on.

A typical use for an XCMD would be as an interface for a driver, allowing HyperCard to control an external device such as a video disc player. Such an interface would have three parts: the driver, the XCMD, and a HyperTalk handler. The driver would be completely separate from HyperCard. (See Inside Macintosh: Devices for information about writing drivers.) The XCMD would be small; its purpose would be to convert HyperTalk messages to the appropriate driver calls. The HyperTalk handler would call the XCMD with various parameters directing it to open or close the driver or to perform a specific control call.

**Using an XCMD or XFCN**

You invoke XCMDs and XFCNs from HyperTalk using the regular message syntax and user-defined function call syntax. The message or function call is passed through the HyperCard message-passing hierarchy.
Appendix A

External Commands and Functions

Invoking XCMDs and XFCNs

You invoke an XCMD as you do a message handler. That is, you type the name of the XCMD followed by its parameters in a HyperTalk script or in the Message box. Separate the parameters (if there are more than one) with commas, and put quotation marks around parameters of more than one word. When the script executes or when you send the Message box contents by pressing Return or Enter, HyperCard sends the message through the normal message-passing hierarchy. For external commands, the Macintosh resource name correlates to the message name—the first word in the message.

Similarly, you call an XFCN in a HyperTalk statement in the same way you would a user-defined function (use parentheses after the function rather than preceding the function with the word the). Enclose any parameters within parentheses, separate them (if more than one) with commas, and put quotation marks around parameters of more than one word. If the function takes no parameters, append empty parentheses after it. For external functions, the Macintosh resource name correlates to the function name—the word preceding parentheses in the function call.

You can pass a maximum of 16 parameters to an XCMD or XFCN.

Message-Passing Hierarchy

External commands and functions use the message-passing hierarchy in the same way as message and function handlers and built-in commands and functions. External commands and functions can be attached to any stack or to the HyperCard application.

If a stack receives a message or function call for which it has no handler, then before passing the message or function call to another stack (if added to the message-passing hierarchy with the start using command) or HyperCard, it checks to see if it has an external command or function of the same name. When HyperCard receives a message or function call, it checks to see if it has an external command or function before it looks for a built-in command or function.

That is, HyperCard searches for message and function handlers, XCMDs and XFCNs, and built-in commands and functions through the message-passing hierarchy shown in Figure A-1.

Chapter 4 discusses the message-passing hierarchy, including the dynamic path, in detail.
Figure A-1 Message-passing hierarchy, including XCMDs and XFCNs

- Message and function calls
  - Field handlers
  - Button handlers
  - Card handlers
  - Background handlers
  - Stack handlers
  - Stack XCMDs and XFCNs
    - Home stack handlers
    - Home stack XCMDs and XFCNs
    - HyperCard XCMDs and XFCNs
    - System file XCMDs and XFCNs
    - HyperCard commands and functions
  - Dynamic path
    - Current card handlers
    - Current background handlers
    - Current stack handlers
    - Current stack XCMDs and XFCNs
Guidelines for Writing XCMDs and XFCNs

XCMDs and XFCNs can call most of the Macintosh Toolbox traps and routines, but they have certain limitations and restrictions. They can’t do everything that an application can do because they are guests in HyperCard’s memory space. In that regard they are more like desk accessories than applications. Here are some guidelines for writing XCMDs and XFCNs:

- Do not initialize the various Macintosh managers by calling their initialization routines. That is, don’t call `InitGraf`, `InitFonts`, `InitWindows`, and so on.

- Do not rely upon having lots of RAM available for your XCMD. There is some extra space in HyperCard’s heap, but if HyperCard is running in 750 KB under MultiFinder, for example, an XCMD should not be bigger than about 32 KB.

- Do not use register A5 of a 68000-family processor. The value in A5 belongs to HyperCard, and it points to HyperCard’s global data, jump table, and other things that constitute an “A5 world.” XCMDs do not currently have their own A5 world.

- XCMDs cannot have global data.

- You can use string literals in XCMDs compiled with the `-b` switch in MPW C version 3.0 or later. You can use ‘STR ’ resources or put the strings in a short assembly-language glue file.

- XCMDs cannot have a jump table, so they cannot have code segments. This restriction imposes a 32 KB limit on the size of XCMDs for 68000-based machines (the 68020 supports longer branches).

- XCMDs can, however, allocate small chunks of memory by standard `NewHandle` calls. (You can also allocate memory with `NewPtr` calls, but they should be used sparingly to avoid heap fragmentation.)

- If your XCMD allocates some memory in the heap, it should also deallocate the memory.
APPENDIX A

External Commands and Functions

- If an XCMD allocates a handle to save state information between invocations of the XCMD, then you must pass the handle back to HyperCard to be stored somewhere in the current stack, such as in a hidden field. You must convert the handle from a long integer to a string, because all values are treated as strings by HyperTalk.

- Since HyperCard jumps blindly to the start of an XCMD’s code, it is important that the main routine actually ends up at the start of the XCMD. The link order is vitally important.

- If, as you write, the size of your XCMD begins to approach 32 KB, consider converting it to a driver.

Attaching an XCMD or XFCN

To attach an existing XCMD or XFCN (one that has already been compiled or assembled into a resource) to one of your stacks, use a resource editor such as ResEdit. The following steps describe the procedure using ResEdit:

1. Launch ResEdit.
2. Select and open the stack containing the ‘XCMD’ or ‘XFCN’ resource you want.
3. Select and open the resource type of ‘XCMD’ or ‘XFCN’.
4. Select and open the particular resource you want by name.
5. Press Command-C to copy the resource.
6. Select and open the stack you want to paste the resource into.
7. If your stack has no resource fork, ResEdit displays a dialog box asking if you want to open one. Click OK. ResEdit opens a window.
8. Press Command-V to paste the resource into your stack.
9. Click the window’s close box. When ResEdit asks if you want to save the file, click Yes.
Parameter Block Data Structure

If HyperCard matches a message or function call with an external command or function, it passes a single argument to the XCMD or XFCN: a pointer to a parameter block called XCmdBlock. All communication between HyperCard and the XCMD or XFCN passes through the parameter block. In Pascal, the parameter block data structure is a record; in C it's a struct.

HyperCard uses the first two fields of the parameter block to pass information to the XCMD or XFCN before invoking its execution. The XCMD or XFCN uses the other data fields in XCmdBlock to pass back results and to communicate with HyperCard during execution.

The Pascal parameter block is shown below:

```pascal
TYPE
  XCmdPtr = ^XCmdBlock;
  XCmdBlock = RECORD
    paramCount: INTEGER; { If -1 then this is an event handling call. }
    params: ARRAY[1..16] OF Handle;
    returnValue: Handle;
    passFlag: BOOLEAN;
    entryPoint: ProcPtr; { to call back to HyperCard }
    request: INTEGER;
    result: INTEGER;
    inArgs: ARRAY[1..8] OF LongInt;
    outArgs: ARRAY[1..4] OF LongInt;
  END;
END;
```
APPENDIX A

External Commands and Functions

Passing Parameters to XCMDs and XFCNs

Before calling the XCMD or XFCN, HyperCard places the number of parameters and handles to the parameter strings in two fields of the parameter block: `paramCount` and `params`.

ParamCount

HyperCard puts an integer representing the parameter count in field `paramCount`. You can pass a maximum of 16 parameter strings.

Params

HyperCard evaluates the parameters and puts their values into memory as zero-terminated ASCII strings. Before it invokes the XCMD or XFCN, HyperCard puts the handles to the parameter strings into the `params` array. For example, the command `Beep 5` creates a single handle in `params[1]` containing the ASCII equivalent of 5 and a zero terminator. HyperCard disposes of the handle.

Passing Back Results to HyperCard

When an XCMD or XFCN finishes executing, HyperCard examines two fields of the parameter block: `returnValue` and `passFlag`.

ReturnValue

An XCMD or XFCN can store one zero-terminated string to communicate the result of its execution. HyperCard looks for a handle to the result string in the `returnValue` field of `XCmdBlock`. Storing a result string is optional for an XCMD; it is expected of an XFCN, but it’s not required. If you store a result string handle into `returnValue` in an XCMD, the user can get it by using the HyperTalk function `the result` (useful for explaining why there was an error). For an XFCN, HyperCard uses the `returnValue` string to replace the function call itself in the HyperTalk statement containing the call. If you don’t store anything, the result is the empty string.
APPENDIX A

External Commands and Functions

PassFlag
When an XCMD or XFCN terminates, HyperCard examines the Boolean value of the passFlag field. If passFlag is FALSE (the normal case), control passes back to the previously executing handler (or to HyperCard’s idle state if no handler was executing). If passFlag is TRUE, HyperCard passes the message or function call to the next object in the hierarchy. This has the same effect as the pass control statement in a script.

Callbacks
The remaining five fields of the XCmdBlock record have to do with calling HyperCard back in the middle of execution of an XCMD or XFCN. You use the callback mechanism to obtain data or request HyperCard to perform an action. HyperCard has 77 callback requests (see “Callback Procedures and Functions,” later in this appendix). The five XCmdBlock fields that compose the callback interface are entryPoint, request, result, inArgs, and outArgs. If you link your code with the HyperXLib library (the HyperXLib.o file that comes with MPW), then you will use only the result field.

EntryPoint
When HyperCard sets up the parameter block data structure before passing control to an XCMD or XFCN, it places an address in entryPoint. The XCMD or XFCN uses this address to execute a jump instruction to pass control to HyperCard for the callback.

Request
Before executing the jump instruction, the XCMD or XFCN puts an integer representing the callback request it’s making into the request field.

Result
After it completes the callback request, HyperCard places an integer result code in the result field. The result code can be 0, 1, or 2. If the callback executed successfully, the result is 0; if it failed, the result is 1; if the callback request is not implemented in HyperCard, the result is 2.
APPENDIX A

External Commands and Functions

InArgs
The XCMD or XFCN sends up to eight arguments to HyperCard as long integers in the inArgs array. Depending on the callback request, HyperCard expects arguments in certain elements of the inArgs array. In many callbacks, the arguments are pointers to zero-terminated strings. The callback arguments are shown in Pascal in “Callback Procedures and Functions,” later in this appendix.

OutArgs
After it executes the callback request, HyperCard returns up to four long integers (or other types, such as handles) to the XCMD or XFCN as elements of the outArgs array. The arguments HyperCard returns from callbacks are shown in Pascal in “Callback Procedures and Functions,” later in this appendix.

Callback Procedures and Functions

If you want to manage a callback to HyperCard yourself, you can define the XCmdBlock data structure in your XCMD or XFCN. Then you can put values you want to send to HyperCard in inArgs, put a request code in request, and execute a jump instruction to the address HyperCard places in entryPoint. HyperCard returns values in outArgs and a result code in result.

However, if you use MPW Pascal or C, you can take advantage of interface definition and library files. The definition and library files provide simple procedure and function calls that you can use inside your XCMD or XFCN to handle callback requests more easily. Include them when you compile and link your XCMD or XFCN.

The Pascal code for an XCMD or XFCN should include the definition file HyperXCmd.p at the beginning of the USES clause. There must be an argument of type XCmdPtr passed by HyperCard to the XCMD or XFCN. In the callback procedures and functions, all strings are Pascal strings unless noted as zero-terminated strings (which have no length byte; the end of the string is indicated by a null byte). In general, if a handle is returned, the XCMD or XFCN is responsible for disposing of it.
External Commands and Functions

HyperTalk Utilities

**FUNCTION EvalExpr(paramPtr: XCmdPtr; expr: Str255): Handle;**

EvalExpr evaluates the HyperTalk expression passed in `expr` and returns a handle to a zero-terminated string containing the result of the evaluation. For example, `EvalExpr('the long date')` returns a handle to a string containing the current date in the long format (Saturday, June 25, 1988). The caller must dispose of the handle.

**PROCEDURE RunHandler(paramPtr: XCmdPtr; handler: Handle);**

The zero-terminated string in `handler` is interpreted first as a message. If it is a message (command or function), it is sent to the current card. The text can be one or more lines of HyperTalk, including conditional statements and repeat loops. The lines are executed as though sent from the Message box. If it is multiple lines beginning with `on` messageName and ending with `end` messageName, messageName is sent to this handler in the context of the card. If the handler exits, execution terminates. If the handler contains the line `pass` messageName, messageName is passed down HyperTalk's normal message-passing path, beginning with the current card script.

You cannot override a script executed with RunHandler. For example, if the current card has an `on` messageName handler, and an XCMD issues a runHandler callback with an `on` messageName, the XCMD's handler executes. If the handler passes the message, the card's script executes.

Execution is somewhat slower using RunHandler than it would be running the same script as a card handler. You cannot use the debugging tools to debug scripts executed with RunHandler.

**PROCEDURE SendCardMessage(paramPtr: XCmdPtr; msg: Str255);**

The string in `msg` is sent as a message to the current card.

**PROCEDURE SendHCMessage(paramPtr: XCmdPtr; msg: Str255);**

The string in `msg` is sent as a message directly to HyperCard, bypassing the entire message-passing hierarchy.
APPENDIX A

External Commands and Functions

Memory Utilities

FUNCTION GetGlobal(paramPtr: XCmdPtr; globName: Str255): Handle;

GetGlobal returns a handle to a zero-terminated string that contains a copy of the contents of the HyperTalk global variable globName. If globName doesn’t exist, GetGlobal returns a handle to an empty string. The caller must dispose of the handle.

PROCEDURE SetGlobal(paramPtr: XCmdPtr; globName: Str255; globValue: Handle);

SetGlobal copies the zero-terminated string to which globValue is a handle into the HyperTalk global variable named globName. If globName doesn’t exist, SetGlobal creates it. HyperCard does not dispose of globValue.

PROCEDURE ZeroBytes(paramPtr: XCmdPtr; dstPtr: Ptr; longCount: LongInt);

ZeroBytes sets longCount bytes beginning at dstPtr to 0. It performs no boundary checking. For example, it can write past the end of a zero-terminated string.

String Utilities

PROCEDURE ScanToReturn(paramPtr: XCmdPtr; VAR scanPtr: Ptr);

ScanToReturn scans the zero-terminated string pointed to by scanPtr, stopping at the first return character or at the end of the string. ScanPtr is incremented to point to the new location.

PROCEDURE ScanToZero(paramPtr: XCmdPtr; VAR scanPtr: Ptr);

ScanToZero scans the zero-terminated string pointed to by scanPtr, stopping at the end of the string. ScanPtr is incremented to point to the new location.
External Commands and Functions

FUNCTION StringEqual(paramPtr: XCmdPtr; str1,str2: Str255): BOOLEAN;

StringEqual compares the two Pascal strings str1 and str2 (case-insensitive and diacritical-sensitive) and returns TRUE if the two strings are identical; otherwise, it returns FALSE.

FUNCTION StringLength(paramPtr: XCmdPtr; strPtr: Ptr): LongInt;

StringLength returns the number of characters in the zero-terminated string pointed to by strPtr. Note that strPtr is a pointer, not a handle.

FUNCTION StringMatch(paramPtr: XCmdPtr; pattern: Str255; target: Ptr): Ptr;

StringMatch performs a case-insensitive search for pattern (a Pascal string) in the zero-terminated string pointed to by target. If the search is successful, the location of the first matching character is returned as the function result. If the search is unsuccessful, StringMatch returns nil. This is equivalent to HyperTalk's offset function.

PROCEDURE ZeroTermHandle(paramPtr: XCmdPtr; hndl: Handle);

ZeroTermHandle increases the block referenced by hndl by 1 byte and then sets the extra byte to 0, making hndl legal for operations such as SaveXWScript, FormatScript, and ZeroToPas.

String Conversions

PROCEDURE BoolToStr(paramPtr: XCmdPtr; bool: BOOLEAN; VAR str: Str255);

BoolToStr converts bool to a Pascal string (TRUE or FALSE).

PROCEDURE ExtToStr(paramPtr: XCmdPtr; num: Extended; VAR str: Str255);

ExtToStr converts num (a SANE extended type) to a Pascal string.
APPENDIX A

External Commands and Functions

PROCEDURE LongToStr(paramPtr: XCmdPtr; posNum: LongInt; VAR str: Str255);

LongToStr converts posNum (a 32-bit unsigned integer) to a Pascal string.

PROCEDURE NumToHex(paramPtr: XCmdPtr; num: LongInt; nDigits: INTEGER; VAR str: Str255);

NumToHex returns in str a hexadecimal (base 16) representation of the value of num, expanding the string to nDigits in length.

PROCEDURE NumToStr(paramPtr: XCmdPtr; num: LongInt; VAR str: Str255);

NumToStr converts num (a 32-bit signed integer) to a Pascal string.

FUNCTION PasToZero(paramPtr: XCmdPtr; str: Str255): Handle;

PasToZero converts str to a zero-terminated string and returns a handle to the new string. The caller must dispose of the handle.

PROCEDURE PointToStr(paramPtr: XCmdPtr; pt: Point; VAR pasStr: Str255);

PointToStr converts the point passed in pt to a Pascal string and returns the string in pasStr.

PROCEDURE RectToStr(paramPtr: XCmdPtr; rct: Rect; VAR pasStr: Str255);

RectToStr converts the rectangle passed in rct to a Pascal string and returns the point in pasStr.

PROCEDURE ReturnToPas(paramPtr: XCmdPtr; zeroStr: Ptr; VAR pasStr: Str255);

ReturnToPas copies characters from the zero-terminated string pointed to by zeroStr into the Pascal string pasStr, stopping at the first return character (ASCII $0D), the end of the zero-terminated string, or the 255th
character, whichever comes first. The variable pasStr will not include the return character.

FUNCTION StrToBool(paramPtr: XCmdPtr; str: Str255): BOOLEAN;

StrToBool converts str to a Boolean (TRUE or FALSE) and returns the Boolean value as its result.

FUNCTION StrToExt(paramPtr: XCmdPtr; str: Str255): Extended;

StrToExt converts str to an extended type. Extended numbers contain a sign bit, 15 bits for the exponent, and 63 bits for the significand. This is the standard data type for SANE, the Standard Apple Numerics Environment.

FUNCTION StrToLong(paramPtr: XCmdPtr; str: Str255): LongInt;

Converts str to a long (32-bit) unsigned integer. Unsigned long integers range from 0 to 4,294,967,295.

FUNCTION StrToNum(paramPtr: XCmdPtr; str: Str255): LongInt;

Converts str to a long (32-bit) signed integer. Signed long integers range from –2,147,483,648 to 2,147,483,647.

PROCEDURE StrToPoint(paramPtr: XCmdPtr; str: Str255; VAR pt: Point);

Converts the Pascal string passed in str to a point and returns the point in pt.

PROCEDURE StrToRect(paramPtr: XCmdPtr; str: Str255; VAR rct: Rect);

Converts the Pascal string passed in str to a rectangle and returns the rectangle in rct.
APPENDIX A

External Commands and Functions

PROCEDURE ZeroToPas(paramPtr: XCmdPtr; zeroStr: Ptr; VAR pasStr: Str255);

ZeroToPas converts the zero-terminated string pointed to by zeroStr to a Pascal string and returns the string in pasStr.

Field Utilities

FUNCTION GetFieldByID(paramPtr: XCmdPtr; cardFld: BOOLEAN; fldID: INTEGER): Handle;

GetFieldByID returns a handle to a zero-terminated string that contains a copy of the contents of field ID fldID. If cardFld is TRUE, fldID is a card field; otherwise, it is a background field.

FUNCTION GetFieldByName(paramPtr: XCmdPtr; cardFld: BOOLEAN; fldName: Str255): Handle;

GetFieldByName returns a handle to a zero-terminated string that contains a copy of the contents of field fldName. If cardFld is TRUE, fldName is a card field; otherwise, it is a background field.

FUNCTION GetFieldByNum(paramPtr: XCmdPtr; cardFld: BOOLEAN; fldNum: INTEGER): Handle;

GetFieldByNum returns a handle to a zero-terminated string that contains a copy of the contents of field number fldNum. If cardFld is TRUE, fldNum is a card field; otherwise, it is a background field.

FUNCTION GetFieldTE(paramPtr: XCmdPtr; cardFieldFlag: BOOLEAN; fieldID, fieldNum: INTEGER; fieldNamePtr: StringPtr): TEHandle;

GetFieldTE returns a copy of the styled TEHandle from the specified field, including style runs (see Inside Macintosh: Text). The caller must dispose of this TEHandle.

If fieldID is nonzero, then HyperTalk uses it; else if fieldNum is nonzero, then HyperTalk uses it; else if fieldNamePtr is not NIL, HyperTalk uses the
APPENDIX A

External Commands and Functions

field name pointed to by it. If GetFieldTE returns NIL, the field was not found or there wasn’t enough memory to copy the text and styles.

PROCEDURE SetFieldByID(paramPtr: XCmdPtr; cardFld: BOOLEAN; fldID: INTEGER; fldVal: Handle);
SetFieldByID copies the zero-terminated string to which fldVal is a handle into the field ID fldID. If cardFld is TRUE, fldID is a card field; otherwise, it is a background field. The caller must dispose of the handle.

PROCEDURE SetFieldByName(paramPtr: XCmdPtr; cardFld: BOOLEAN; fldName: Str255; fldVal: Handle);
SetFieldByName copies the zero-terminated string to which fldVal is a handle into field fldName. If cardFld is TRUE, fldName is a card field; otherwise, it is a background field. The caller must dispose of the handle.

PROCEDURE SetFieldByNum(paramPtr: XCmdPtr; cardFld: BOOLEAN; fldNum: INTEGER; fldVal: Handle);
SetFieldByNum copies the zero-terminated string to which fldVal is a handle into the field number fldNum. If cardFld is TRUE, fldNum is a card field; otherwise, it is a background field. The caller must dispose of the handle.

PROCEDURE SetFieldTE(paramPtr: XCmdPtr; cardFieldFlag: BOOLEAN; fieldID, fieldNum: INTEGER; fieldNamePtr: StringPtr; fieldTE: TEHandle);
SetFieldTE sets the text and styles of the field to the text and styles contained in fieldTE.

Miscellaneous Utilities

PROCEDURE BeginXSound(paramPtr: XCmdPtr; window: WindowPtr);
BeginXSound informs HyperCard that an XCMD is about to use the Sound Manager. An XCMD should call BeginXSound before it attempts to allocate a sound channel or perform any other Sound Manager operation. After an
APPENDIX A

External Commands and Functions

XCMD calls BeginXSound, HyperCard’s built-in play command will not operate until the XCMD calls EndXSound (see the next procedure). If an external window is making the callback, it should pass a pointer to its window in the window parameter. An XCMD should pass NIL for the window parameter. An XCMD or an external window can optionally pass a pointer to a valid external window in the window parameter if it wants to “aim” the call at another external window. If HyperCard gets a valid WindowPtr in window, it will post an xGiveUpSoundEvt event to that window at an appropriate time. If the XCMD passes NIL for window, HyperCard will be unable to signal the XCMD when HyperCard needs the sound channel back.

An XCMD that uses the Sound Manager can be structured as follows:

BeginXSound(paramPtr, NIL);
(* allocate a sound channel *)
(* do your sound thing *)
(* deallocate sound channel *)
EndXSound(paramPtr);

PROCEDURE EndXSound(paramPtr: XCmdPtr);

EndXSound informs HyperCard that an XCMD has finished using the Sound Manager. If the XCMD has not previously called BeginXSound, EndXSound does nothing.

FUNCTION FrontDocWindow(paramPtr: XCmdPtr): WindowPtr;

FrontDocWindow returns the WindowPtr of the document that is frontmost in HyperCard’s document layer. It does not return the WindowPtr of a window in the miniwindow layer. To return the frontmost window of any type, use the Window Manager’s FrontWindow function.

FUNCTION GetFilePath(paramPtr: XCmdPtr; fileName: Str255; numTypes: INTEGER; typeList: SFTTypeList; askUser: BOOLEAN; VAR fileType: OSType; VAR fullName: Str255): BOOLEAN;

GetFilePath determines the full pathname of the file fileName using the search paths stored in the Home stack. If fileType is ‘STAK’, GetFilePath uses the stack search paths. If fileType is ‘APPL’, GetFilePath uses the
application search paths. If `fileType` is neither 'STAK' nor 'APPL',
GetFilePath uses the document search paths. The parameters `numTypes` and `typeList` are used as described in the chapter “Standard File Package” in *Inside Macintosh: Files*. If `askUser` is TRUE and HyperCard fails to find the file on its own, GetFilePath prompts the user to find the file with a standard file dialog box. If the file is located either by HyperCard or by the user, the full pathname of the file is returned in `fullName` and the file type is returned in `fileType`. If the user clicks Cancel in the standard file dialog box or HyperCard fails to find the file for any other reason, GetFilePath returns FALSE.

```pascal
PROCEDURE GetXResInfo(paramPtr: XCmdPtr; VAR resFile: INTEGER; VAR resID: INTEGER; VAR rType: ResType; VAR name: Str255);

GetXResInfo returns the file reference number of the resource file from which the calling XCMD was read in `resFile` and the resource ID of the XCMD in `resID`, the resource type (XCMD or XFCN) in `rType`, and the resource name in `name`.

PROCEDURE Notify(paramPtr: XCmdPtr);

If HyperCard is active or MultiFinder is not loaded, Notify returns immediately. Otherwise, Notify blinks the small HyperCard icon over the Apple in the Apple menu (System 6) or over the Applications menu (System 7) until the user switches to HyperCard’s layer. Only then does Notify return. No other HyperCard processing takes place while Notify is waiting.

PROCEDURE SendHCEvent(paramPtr: XCmdPtr; event: EventRecord);

SendHCEvent is useful only to XCMDs that call the Toolbox routines GetNextEvent and WaitNextEvent. Such XCMDs should use SendHCEvent to pass events required by HyperCard. For example, an XCMD that creates a draggable window and calls GetNextEvent may receive update events for HyperCard windows. HyperCard performs the updates if it receives the update events via this callback. More importantly, an XCMD that calls GetNextEvent and receives an app4Evt generated by MultiFinder must pass the event along to HyperCard. **If HyperCard does not receive its suspend and resume events, unexpected results may occur.**

Callback Procedures and Functions 521
APPENDIX A

External Commands and Functions

XCMDs that create windows by means of the NewXWindow callback don’t need to call GetNextEvent or WaitNextEvent and therefore don’t need to use SendHCEvent.

PROCEDURE SendWindowMessage(paramPtr: XCmdPtr; windPtr: WindowPtr; windowName: Str255; msg: Str255);

SendWindowMessage is functionally equivalent to send message to window windowName from HyperTalk. Use this for direct communication between XCMDs that manage external windows. If windPtr is not NIL, the window pointer in WindowPtr is used to determine which window receives the message; otherwise, the name in windowName is used.

FUNCTION StackNameToNum(paramPtr: XCmdPtr; stackName: Str255): LongInt;

Internally, HyperCard no longer remembers stacks only by their name. It uses a stack number to represent the stack. This number is similar to a volume reference number: it is valid as an indicator as long as the application is open, but won’t be valid across multiple launches. This number is valid when used in an XTalkObject to get and set the scripts of objects. StackNameToNum translates the name of a stack into this number.

Creating and Disposing of External Windows

FUNCTION GetNewXWindow(paramPtr: XCmdPtr; templateType: ResType; templateID: INTEGER; colorWind: BOOLEAN; floating: BOOLEAN): WindowPtr;

GetNewXWindow creates a new window or dialog box from a resource. If the Window Manager fails to create the window, GetNewXWindow returns NIL. If the window is created successfully, GetNewXWindow sets up the mechanism by which events pertaining to the window are sent to the XCMD that created it. TemplateType must be either 'WIND' or 'DLOG'. TemplateID is the resource ID of the window or dialog box template to be used. The template resource can exist in any resource file that’s currently open.
If `colorWind` is TRUE, HyperTalk attempts to create a color window using the `GetNewCWindow` or `NewCDialog` toolbox trap. If `colorWind` is TRUE and Color QuickDraw is not present, the window is not created and `GetNewXWindow` returns NIL.

If `floating` is TRUE, HyperTalk places the new window in the miniwindow layer. Otherwise, the window is placed into the document layer. See “Window Layer Management,” later in this appendix, for an explanation of these two layers and their relationship to each other.

**Note**

`GetNewXWindow` is compatible with nonstandard window definition functions (see `NewXWindow` for more information).

```plaintext
FUNCTION NewXWindow(paramPtr: XCmdPtr; boundsRect: Rect; title: Str255; visible: BOOLEAN; procID: INTEGER; colorWind: BOOLEAN; floating: BOOLEAN): WindowPtr;
```

`NewXWindow` creates a new window and returns a pointer to it as the function’s result. If the Window Manager fails to create the window, `NewXWindow` returns NIL. If the window is created successfully, `NewXWindow` sets up the mechanism by which events pertaining to the window are sent to the XCMD that created it.

If `colorWind` is TRUE, HyperTalk attempts to create a color window using the `NewCWindow` Toolbox trap. If `colorWind` is TRUE and Color QuickDraw is not present, the window is not created and `NewXWindow` returns NIL.

The value in `procID` is the same as the `procID` argument to the Window Manager routine `NewWindow`. For example, passing `documentProc` as the `procID` produces a standard document window with no zoom box. `BoundsRect` is the bounding rectangle for the window in global coordinates. `Title` becomes the title of the window.

The `visible` argument determines whether the window is created visible.

To create windows similar to HyperCard miniwindows, such as the Tools palette and the Message box, use HyperCard’s built-in window definition function for windows.
APPENDIX A

External Commands and Functions

Here are the possible built-in window values to use for procID:

\[
\begin{align*}
paletteProc &= 2048; \quad \text{\{ window with grow box \}} \\
palNoGrowProc &= 2052; \quad \text{\{ standard window \}} \\
palZoomProc &= 2056; \quad \text{\{ window with zoom and grow \}} \\
palZoomNoGrow &= 2060; \quad \text{\{ window with zoom and no grow \}} \\
hasZoom &= 8; \\
hastallTBar &= 2; \\
toggleHilite &= 1; \\
\end{align*}
\]

For example, the Navigator palette uses palNoGrowProc for procID.

**Note**

NewXWindow is also compatible with other nonstandard window definition functions.

```
PROCEDURE CloseXWindow(paramPtr: XCmdPtr; window: WindowPtr);
```

When an XCMD that manages an external window requests that the window be closed, it should call CloseXWindow. When all pending calls to the XCMD have returned, HyperCard sends an xCloseEvt to the XCMD. Only in response to this event should it dispose of its data structures and exit.

When the external window receives the xCloseEvt, it signals its willingness to close by setting paramPtr^.passFlag to TRUE. If this is not done, HyperTalk will not proceed with the disposal of the window.

HyperTalk tries to close all open external windows when the user quits HyperCard. If any windows refuse to close at that time, HyperCard will not quit.

Do not use any of the following Toolbox routines to close external windows: CloseWindow, DisposeWindow, CloseDialog, and DisposDialog.

Any XCMD can close any external window created with the NewXWindow or GetNewXWindow calls by means of a call to CloseXWindow. CloseXWindow has no effect on windows that were not created by means of a call to NewXWindow or GetNewXWindow.

You can also close an external window from a script with the close window command.
Window Utilities

PROCEDURE HideHCPalettes(paramPtr: XCmdPtr);

HideHCPalettes hides all of HyperCard's built-in miniwindows (the Tools palette, the Patterns palette, the FatBits window, the Scroll window, and the Message box).

PROCEDURE RegisterXWMenu(paramPtr: XCmdPtr; menu: MenuHandle; registering: BOOLEAN);

RegisterXWMenu is useful only to XCMDs that manage external windows. RegisterXWMenu informs HyperCard that the given menu is meant for use with the external window. When an item in the menu is chosen by the user, the XCMD that registered the menu chosen receives a menu event (xMenuEvt). Note that an XCMD can register one of HyperCard's menus, for example, the Font menu, in order to borrow the menu temporarily.

RegisterXWMenu does not change the menu bar. The XCMD must call the Menu Manager in order to insert or delete the menu and to redraw the menu bar. Once the menu has been registered, it remains the property of the XCMD until RegisterXWMenu is called again with registering set to FALSE.

Note

HyperCard expects XCMD menus to have unique IDs but does not ensure that they do. You may employ the following function to find an unused menu ID. ♦

FUNCTION UnusedMenuID: INTEGER;
VAR thisID: INTEGER;
    menuHndl: MenuHandle;
BEGIN
    thisID := 1023;
    REPEAT
        thisID := thisID + 1;
        menuHndl := GetMHandle(thisID);
        UNTIL menuHndl = NIL;
    UnusedMenuID := thisID;
END;
APPENDIX A

External Commands and Functions

PROCEDURE SetXWIdleTime(paramPtr: XCmdPtr; ticks: LongInt);

XCMDs that manage external windows can request idle time from HyperCard. Use SetXWIdleTime if your XCMD needs to perform a periodic action.

Until SetXWIdleTime is called by an XCMD, the XCMD does not receive periodic calls. A value greater than 0 for ticks represents the requested interval between periodic calls by HyperCard. HyperCard sends the XCMD a nullEvent message when making its periodic call. Call SetXWIdleTime only after your external window has received an xOpenEvt.

Whether HyperCard's periodic calls occur as often as requested depends on whether HyperCard is currently performing a timing-critical or data-intensive operation.

To give up idle time, call SetXWIdleTime with a value of 0 for ticks.

Note
NullEvent messages are sent in both the foreground and background under MultiFinder.

PROCEDURE ShowHCPalettes(paramPtr: XCmdPtr);

ShowHCPalettes reverses the effect of HideHCPalettes, showing all of HyperCard’s miniwindows that were visible when HideHCPalettes was called to hide them.

PROCEDURE XWHasInterruptCode(paramPtr: XCmdPtr; haveCode: BOOLEAN);

When HyperCard creates an external window (by means of either the NewXWindow or GetNewXWindow callback), it is assumed that the code of the XCMD that manages it can be moved in memory when it is not executing.

If haveCode is TRUE, HyperCard never unlocks the relocatable block containing the XCMD code in memory. This allows ProcPtrs and other addresses within an XCMD’s code to be preserved and valid at all times. XWHasInterruptCode should be used with extreme prudence and should be undone as soon as possible—HyperCard’s memory management can become seriously taxed if there are any locked blocks inconveniently located in its heap. In particular, it's impossible to open card windows at the full size of large cards when a nonrelocatable block is located too close to the bottom of the application heap.
APPENDIX A

External Commands and Functions

The preferred method for an XCMD to manage procedure pointers passed to the Toolbox is to refresh them as needed rather than to call XWHasInterruptCode to ensure their validity. This method permits HyperCard to unlock the XCMD’s code between invocations. An example of such an XCMD is a text editor with a custom clikLoop routine. For best results, such an XCMD should recalculate and refresh its procPtr every time the user clicks the viewRect. For example:

```cua
CASE evt.what OF
    mouseDown:
        BEGIN
            GlobalToLocal(evt.where);
            hTE^^.clikLoop := @MyClikLoop;
                { an assembly-language clik loop }
            TEClick(evt.where, FALSE, hTE);
        END;

An XCMD that uses this method will have no need to call XWHasInterruptCode.

An XCMD’s code is always locked while it is actually executing. XWHasInterruptCode determines only whether the block of code can be unlocked when control is returned to HyperCard.

PROCEDURE XWAlwaysMoveHigh(paramPtr: XCmdPtr; moveHigh: BOOLEAN);

XWAlwaysMoveHigh tells HyperTalk to always move the external window’s code high on the heap before locking it down and calling it. External windows that may allocate large amounts of memory or send card messages to do such operations as go to another card should use this call.

In normal XCMD and external window operations, HyperTalk makes a determination at the time the code is jumped to as to whether the code should be moved high in the application heap before being locked down. Currently, this is done whenever an XCMD is called and when an external window is given the following events: xOpenEvt, xMenuEvt, mouseDown, and keyDown. All other events are assumed to rarely cause memory allocation or messages to be sent; therefore, HyperTalk can speed up the process of calling and returning from an external window. However, some external windows may allocate
APPENDIX A

External Commands and Functions

memory on such things as an xSetPropEvt event. For example, a picture XCMD could be asked to change information about the currently displayed PICT, and that may cause memory to be allocated. A good rule of thumb is if your external window is not called repeatedly in a script and you might allocate memory on events other than the aforementioned events, call this with moveHigh = TRUE.

PROCEDURE XWAllowReEntrancy(paramPtr: XCmdPtr; allowSysEvts: BOOLEAN; allowHCEvts: BOOLEAN);

XWAllowReEntrancy allows an XCMD to tell HyperCard whether it is equipped to receive events (either system events or HyperCard-generated events) in a reentrant fashion. Either of the Booleans, allowSysEvts or allowHCEvts, can be set to TRUE or FALSE to enable or disable (respectively) this behavior.

In certain situations, it is possible for an external window to generate events for itself. In other words, calling the Window Manager’s InvalRect routine on a portion of the window could create an updateEvt event for the window.

While running scripts, HyperTalk periodically checks the Event Manager to see if update, activate, or MultiFinder events are pending. This is how HyperCard processes events in the background. In the course of these checks, the Window Manager may report an update event for the external window if it notices that any or all of the window is invalid. The problem arises from external windows that use SendCardMessage, SendHCMmessage, EvalExpr, or any other HyperTalk callback that can run a script. XWAllowReEntrancy was implemented to allow external windows to perform asynchronous handling of certain events. For example, if a SendCardMessage callback causes a lengthy script to be executed, the external window may want to be notified that the user switched out of HyperCard under MultiFinder. The default is for HyperTalk to not allow these reentrant calls to take place. Unlike system events, HyperTalk events are not queued, so unreceived events are ignored.

Note

An external window should only call XWAllowReEntrancy at the end of its response to an xOpenEvt event. The first event that is sent to an external window is its xOpenEvt event. ◆
Because xGetPropEvt and xSetPropEvt are events, if an external window has not called XWAllowReEntrancy(paramPtr, xxxx, TRUE), the scenario described below results in the external window never receiving its xSetPropEvt for its visible property:

SendCardMessage(paramPtr,'go next card');
  on closeCard
    hide window "fred" -- this call will fail
  end closeCard

The xSetPropEvt causes the external window to be reentered, so HyperTalk skips the event and continues. If your external window needs to communicate with running scripts, make sure that you allow it to be reentrant.

Writing completely reentrant code is difficult, and some development systems may not support reentrancy correctly. One tip is to make sure that every time an external window receives an event, it should completely save and restore its state. A good idea is to use the external window’s refCon field to store a handle to all the window’s state information. Beware of the use of global variables in stand-alone code resources. See Macintosh Technical Note 256 for more information.

Text Editing Utilities

PROCEDURE BeginXWEdit(paramPtr: XCmdPtr; window: WindowPtr);

BeginXWEdit registers an external window as the current editing environment. It can be called only by the XCMD that manages the window. Once it is called, HyperCard redirects all keystrokes to the XCMD, with the exception of Command-key combinations recognized by the Menu Manager as the equivalent of menu items. In addition, HyperCard passes events to the XCMD that correspond to the first five commands in the Edit menu whenever these commands are chosen. The XCMD still receives all of the other events pertaining to its window, including nullEvents if it has requested them.

Once the XCMD has registered itself as the current editing environment, it receives an xGiveUpEdit event from HyperCard when the user performs an action that activates a different editing environment, such as a click in the
APPENDIX A

External Commands and Functions

Message box or in an unlocked field. When this happens, the XCMD should deactivate its editable area as appropriate, just as it does when its window is deactivated.

PROCEDURE EndXWEdit(paramPtr: XCmdPtr; window: WindowPtr);
EndXWEdit informs HyperCard that an XCMD no longer wants to receive keystrokes and edit events for its external window.
Call EndXWEdit before an external window that is an editing environment is closed.

FUNCTION HCWordBreakProc(paramPtr: XCmdPtr): ProcPtr;
HCWordBreakProc returns a procedure pointer to HyperCard’s built-in word-break routine used for text in fields, scripts, and the Message box. Script editors and other text editing XCMDs can use this address in the wordBreak field of a TextEdit record.

PROCEDURE PrintTEHandle(paramPtr: XCmdPtr; hTE: TEHandle; header: StringPtr);
Given a handle to a TextEdit record in hTE, PrintTEHandle displays a print job dialog box and prints the record using the font, size, and style information contained within it. PrintTEHandle works for both old- and new-style edit records.

Script Editor Utilities

PROCEDURE FormatScript(paramPtr: XCmdPtr; scriptHndl: Handle; VAR insertionPoint: LongInt; quickFormat: BOOLEAN);
FormatScript reformats the script contained in the zero-terminated string to which scriptHndl is a handle. Call ZeroTermHandle before calling FormatScript if the text is not zero-terminated.
The offset into the text passed in insertionPoint is adjusted as necessary to reflect the same position within the text both before and after formatting. An XCMD that uses TextEdit should pass TEHandle^^.selStart for this parameter, then call TESetSelection after FormatScript returns.
APPENDIX A

External Commands and Functions

InsertionPoint is also used when quickFormat is TRUE to determine which handler within the script should be formatted.

**Note**
Using EvalExpr to get the script of an object formats it automatically. ◆

FUNCTION GetCheckPoints(paramPtr: XCmdPtr): CheckPtHandle;

GetCheckPoints returns a handle to the cached checkpoints for the window’s script. The caller must dispose of the handle. If there are no cached checkpoints, GetCheckPoints returns NIL.

The structure of a CheckPtHandle is as follows:

```
    CheckPtHandle = ^CheckPtPtr;
    CheckPtPtr = ^CheckPts;
    CheckPts = RECORD
        checks: ARRAY[1..16] OF INTEGER;
    END;
```

PROCEDURE SaveXWScript(paramPtr: XCmdPtr; scriptText: Handle);

Given a zero-terminated handle to a script in scriptText, SaveXWScript saves the script to disk. Call ZeroTermHandle before calling SaveXWScript if the text is not zero-terminated.

This callback is reserved for the use by XCMDs that implement script editors. It fails if the XCMD has not been called by HyperCard as the current script editor. SaveXWScript also fails if the text of the script exceeds 32 KB.

**Note**
An XCMD can set the script of any object using the following example code:

```
    SetGlobal(paramPtr,'tempGlobal',scriptHndl);
    SendCardMessage(paramPtr, CONCAT('set the script of ',objectName, ' to tempGlobal'));
    SendCardMessage(paramPtr,'put empty into tempGlobal');
```
External Commands and Functions

PROCEDURE SetCheckPoints(paramPtr: XCmdPtr; checkLines: CheckPtHandle);

SetCheckPoints sets the checkpoints of a script to the array of lines in checkLines.

PROCEDURE GetObjectName(paramPtr: XCmdPtr; object: XTalkObject; VAR objName: Str255);

Given an XTalkObject as input, returns the name of the object in objName.

PROCEDURE GetObjectScript(paramPtr: XCmdPtr; object: XTalkObject; VAR scriptHndl: Handle);

Given an XTalkObject as input, returns the script of the object in scriptHndl.

PROCEDURE SetObjectScript(paramPtr: XCmdPtr; object: XTalkObject; scriptHndl: Handle);

Given an XTalkObject and a handle to a script as input, sets the script of the object described by XTalkObject to the script in scriptHndl.

Variable Watcher Support

PROCEDURE CountHandlers(paramPtr: XCmdPtr; VAR handlerCount: INTEGER);

CountHandlers returns the number of running handlers in handlerCount. If handlerCount is 0, then no handlers are currently running and the only variables that exist are global variables.

PROCEDURE GetHandlerInfo(paramPtr: XCmdPtr; handlerNum: INTEGER; VAR handlerName: Str255; VAR objectName: Str255; VAR varCount: INTEGER);

GetHandlerInfo returns information about handlers. If handlerNum is 1, GetHandlerInfo returns information about the current handler, if there is one. If handlerNum is greater than 1, GetHandlerInfo returns
information about a handler that has a pending call to another handler. For example, if `handlerNum` is 2, the information returned is for the handler that called the current handler. If there is no running handler, passing 0 for `handlerNum` yields information about the global variables.

```pascal
PROCEDURE GetVarInfo(paramPtr: XCmdPtr; handlerNum: INTEGER; varNum: INTEGER; VAR varName: Str255; VAR isGlobal: BOOLEAN; VAR varValue: Str255; varHndl: Handle);
```

Given the variable number `varNum` in active handler number `handlerNum`, `GetVarInfo` returns information about the variable. If `varHndl` is NIL, a truncated version of the contents of the variable, limited to 255 characters, is returned in `varValue`. If `varHndl` is a valid handle, the entire contents of the variable are copied into that handle, which is resized as necessary. If there is no running handler, passing 0 for `handlerNum` returns information about the global variables.

```pascal
PROCEDURE SetVarValue(paramPtr: XCmdPtr; handlerNum: INTEGER; varNum: INTEGER; varHndl: Handle);
```

`SetVarValue` sets the value of variable number `varNum` of handler `handlerNum` to the zero-terminated string passed in `varHndl`.

**Note**
Changing the value of a variable other than the global variables (`handlerNum 0`) or the variables local to the current handler (`handlerNum 1`) does not affect the handler until it is once again current. ♦

**Debugger Support**

```pascal
PROCEDURE AbortScript(paramPtr: XCmdPtr);
```

`AbortScript` cancels the currently executing handlers. The effect is the same as pressing Command-period or choosing Abort from the HyperCard built-in Debugger menu.

**Note**
An XCMD that calls `AbortScript` is not itself aborted. It exits normally. ♦
FUNCTION GetStackCrawl(paramPtr: XCmdPtr): Handle;

Returns a zero-terminated handle to the chain of callers (indented) as in the Message Watcher.

PROCEDURE GoScript(paramPtr: XCmdPtr);

GoScript exits the debugger, closes all temporary debugger windows, and continues normal execution of scripts.

PROCEDURE StepScript(paramPtr: XCmdPtr; stepInto: BOOLEAN);

StepScript creates a temporary checkpoint after the currently executing line. If stepInto is TRUE, the checkpoint is created at the next line of HyperTalk to be executed, even if it belongs to another handler. If stepInto is FALSE, the checkpoint is created at the next line of the current handler. StepScript causes the debugger to step out of a handler if the current line is its last.

PROCEDURE TraceScript(paramPtr: XCmdPtr; traceInto: BOOLEAN);

TraceScript is similar to StepScript above, except that execution continues after the line is highlighted by the debugger. TraceScript respects the new HyperTalk property traceDelay.

---

External Windows

The HyperCard XCMD interface includes support for external windows. HyperCard’s built-in external windows are the script editor and the debugger tools Variable Watcher and Message Watcher, which are all described earlier in this book. All of these windows and their properties can be controlled through scripts. You can write your own XCMDs to replace any one of the built-in HyperCard external windows. You can also write XCMDs that provide any kind of useful application within a window, such as custom text editors, color picture editors, video overlay windows, and so on.

External windows are an extension of external commands. Two new callbacks, NewXWindow and GetNewXWindow, direct HyperCard to create a new external window. When an XCMD executes either of these callbacks, HyperCard creates a new window and saves with it a reference to the XCMD that made the call.
APPENDIX A

External Commands and Functions

Then, whenever HyperCard receives an event from the Toolbox Event Manager, it first determines whether the event pertains to an external window. If so, HyperCard calls the XCMD that created the window with arguments that allow it to handle the event. Otherwise, HyperCard handles the event itself.

Whenever an XCMD is called by HyperCard, it receives a pointer to an XCmdBlock parameter block. See “Parameter Block Data Structure,” earlier in this appendix, for the Pascal parameter block structure.

When HyperCard calls an XCMD to handle an event for an external window, some of the fields of XCmdBlock have new values. The paramCount field is set to -1, indicating that the XCMD has been called to handle an event. The first parameter, params[1], is a pointer to an XWEventInfo block, defined as follows:

XWEventInfoPtr = ^XWEventInfo;
XWEventInfo = RECORD
  ownerWindow: WindowPtr;
  theEvent: EventRecord;
  eventParams: ARRAY[1..9] OF LongInt;
  eventResult: Handle;
END;

Therefore, an XCMD that manages an external window can be structured as follows:

IF paramPtr^.paramCount >=0 THEN CreateMyWindow
ELSE
BEGIN
  WITH XWEventInfoPtr(paramPtr^.params[1])^ DO
  BEGIN
    myEvent := theEvent;
    myWindow := ownerWindow;
    END;
  SetPort(myWindow);
  CASE myEvent.what OF
    mouseDown: DoMouse;
    and more code here...
  END;
END;
END;
When an XCMD has been called to handle an event, it has full access to callback routines, just as it does when invoked from a HyperTalk script.

**Events in External Windows**

HyperCard automatically sends most standard Macintosh events to XCMDs that manage external windows. These include the following events:

- `activateEvt` (miniwindows should hide themselves when suspended and show themselves when resumed)
- `app4Evt`
- `mouseDown`
- `nullEvent` (see `SetXWIdleTime`)
- `updateEvt`

Keyboard events (`keyDown` and `autoKey`) are delivered to the XCMD only if it has registered itself as the active editing environment with the `BeginXWEdit` callback. All keystrokes, with the exception of Command-key combinations recognized by the Menu Manager as equivalents of menu items, are sent to an XCMD when it is the active editing environment.

In addition, XCMDs that manage windows receive messages specific to HyperCard. These are delivered in the same `EventRecord` data structure used for standard Macintosh events, with the `what` field set to one of the following constants:

- `xOpenEvt` = 1000; { the first event after window is created}
- `xCloseEvt` = 1001; { the window will be closed}
- `xGiveUpEditEvt` = 1002; { you are losing Edit}
- `xGiveUpSoundEvt` = 1003; { the sound channel is requested}
- `xEditUndo` = 1100; { Edit--Undo}
- `xEditCut` = 1102; { Edit--Cut}
- `xEditCopy` = 1103; { Edit--Copy}
- `xEditPaste` = 1104; { Edit--Paste}
- `xEditClear` = 1105; { Edit--Clear}
- `xSendEvt` = 1200; { script has sent a message (text)}
- `xSetPropEvt` = 1201; { set a window property}
APPENDIX A

External Commands and Functions

\[ xGetPropEvt = 1202; \text{ (get a window property)} \]
\[ xCursorWithin = 1300; \text{ (cursor is within the window)} \]
\[ xMenuEvt = 1400; \text{ (an item in your menu is selected)} \]
\[ xMBarClickedEvt = 1401; \text{ (menu is about to be shown or updated)} \]
\[ xShowWatchInfoEvt = 1501; \text{ (variable and message watcher event)} \]
\[ xScriptErrorEvt = 1502; \text{ (place the insertion point)} \]
\[ xDebugErrorEvt = 1503; \text{ (user clicked Debug at complaint)} \]
\[ xDebugStepEvt = 1504; \text{ (highlight the line stepping)} \]
\[ xDebugTraceEvt = 1505; \text{ (highlight the line tracing)} \]
\[ xDebugFinishedEvt = 1506; \text{ (script ended)} \]

Handling Events

Many of the events above require special attention. Many are specific to the debugger or other debugging tools. Here is a summary of all the events and their appropriate behavior:

\[ xOpenEvt \]
Use this event to set up the external window-specific parts of your code, such as any of the callbacks that have an XW in their name. These are RegisterXWMenu, SetXWIdleTime, XWHasInterruptCode, XWAlwaysMoveHigh, and XWAllowReEntrancy.

\[ xCloseEvt \]
HyperTalk sends your external window an xCloseEvt event when your window has called CloseXWindow, when another external window or XCMD has called CloseXWindow using your window’s WindowPtr, or when the user has used the new close window "windowName" command. The external window should not dispose of any of its data until receiving this event. If the external window sets passFlag to TRUE, the window is disposed. If not, the window is left open.

\[ xGiveUpEditEvt \]
When an external window calls BeginXWEdit, it will receive keystrokes and Edit menu commands until the user clicks back in the card window or activates some other editing window (for example, the Message box). HyperTalk signals the current editor with an xGiveUpEditEvt event just before it activates the other editor.

---

continued
External Commands and Functions

xGiveUpSoundEvt  In the event that one external window requests the sound channel (with BeginXSound) while another external window has the channel, the current owner is given an xGiveUpSoundEvt. If the external window doesn’t set passFlag to TRUE, the other external window’s callback returns with the result code set to xresFail. If an XCMD owns the sound channel, it can’t be notified of the second request, so that request will fail.

xEditUndo  While an external window is the editing environment, it receives this event, which corresponds to the Undo command in HyperCard’s built-in Edit menu.

xEditCut  While an external window is the editing environment, it receives this event, which corresponds to the Cut command in HyperCard’s built-in Edit menu.

xEditCopy  While an external window is the editing environment, it receives this event, which corresponds to the Copy command in HyperCard’s built-in Edit menu.

xEditPaste  While an external window is the editing environment, it receives this event, which corresponds to the Paste command in HyperCard’s built-in Edit menu.

xEditClear  While an external window is the editing environment, it receives this event, which corresponds to the Clear command in HyperCard’s built-in Edit menu.

xSendEvt  When a user issues the send message to window command, or when an XCMD or external window issues the SendWindowMessage callback, the external command receives an xSendEvt event. When this event is received, eventParams[1] contains a pointer to a Pascal string (Str255) containing the name of the message. This is done for speed purposes and to expedite using the StringEqual callback to index through the commands your window supports.

xSetPropEvt  HyperTalk contains new extensible syntax for setting properties of external windows. The syntax is

set property of window to propertyValue

xGetPropEvt  HyperTalk contains new extensible syntax for getting properties of external windows. The syntax is

get property of window
HyperTalk has two built-in properties of external windows: loc and visible. If an external window doesn’t do anything special in response to being moved, shown, or hidden, it can set passFlag to TRUE in response to both xSetPropEvt and xGetPropEvt and HyperTalk handles the request. If a property is requested other than the two built-in ones and the external window passes the event, HyperTalk displays an error dialog box to the user.

HyperCard 2.2 adds these built-in properties: rectangle, width, height, and the corresponding rectangle properties (see Chapter 12). These properties are read-only. If you want to change these properties, your XCMD must handle this itself.

If the external window wishes to respond specially to these requests, or if it has additional properties it wants to support, it can directly handle all of HyperTalk’s requests. In both xSetPropEvt and xGetPropEvt, eventParams[1] contains a pointer to a Pascal string (Str255) containing the name of the property. In the case of an xSetPropEvt, eventParams[2] contains a handle to the property value. In the case of an xGetPropEvt event, the external window must return a handle in the eventResult field with the value of the property requested.

HyperTalk gets and sets the visible property to translate the hide and show commands. In the event the user says hide window "Variable Watcher", HyperTalk informs the external window by giving it an xSetPropEvt event with the property visible and the value false.

For an example of the ability of external windows to use properties, try these properties of the Message Watcher window: loc, visible, hideIdle, hideUnused, text, and nextLine (set only). The Variable Watcher supports the following properties: loc, visible, hBarLoc, vBarLoc, and rect.

There are four new callbacks to aid in getting and setting properties: PointToStr, RectToStr, StrToPoint, and StrToRect.

xCursorWithin
This event is sent when the cursor is over any part of the external window. If the external window sets passFlag to TRUE, HyperCard sets the cursor to an arrow.

xMenuEvt
When an external window has used RegisterXWMenu for one or more menus in HyperCard’s menu bar, HyperTalk notifies it of an item being chosen by sending it an xMenuEvt event. This holds true for Command-key equivalents as well. When the event is received, eventParams[1] is the menu ID and eventParams[2] is the menu item.

continued
External Commands and Functions

xMBarClickedEvt  
If an external window has registered one or more menus in the menu bar, HyperTalk sends it an xMBarClickedEvt event just before it calls the Menu Manager's MenuSelect or MenuKey routine. This is to allow the external window to adjust its menus just before the user sees them.

xShowWatchInfoEvt  
Sent to the Message Watcher and Variable Watcher whenever it is appropriate. For the Message Watcher, eventParams[1] contains a handle to the current message. For the Variable Watcher, there are no arguments sent.

xScriptErrorEvt  
Sent to the current script editor when HyperTalk displays the “Debug, Script, or Cancel” alert and the user clicks Script. EventParams[1] contains the line number on which the error occurred.

xDebugErrorEvt, xDebugStepEvt, xDebugTraceEvt  
Sent to the current debugger window.

xDebugFinishedEvt  
Sent to all external windows when the user chooses the Go or Abort command in the Debugger menu.

Closing an External Window

External windows should only dispose of their private data structures in response to an xCloseEvt event. Whenever an XCMD calls CloseXWindow, it should not dispose of any of its internal information until it receives its xCloseEvt event.

When an xCloseEvt event is sent, the XCMD must set passFlag to TRUE if it actually wants to close. If it does not set passFlag, it is telling HyperTalk that it wants to cancel the close operation. This is useful for editors that put up a “Save this document?” dialog box that includes a Cancel button.

Special XCmdBlock Values

Because special XCMDs like the Message Watcher and script editor look like normal XCMDs, there has to be some defense against a user typing ScriptEditor in the Message box and launching the script editor. Therefore, the special-case XCMDs are given special parameter blocks when they are opened by HyperTalk.
APPENDIX A

External Commands and Functions

Message Watcher

When HyperTalk calls the Message Watcher to initialize itself, it sets `paramPtr^.paramCount` to `xMessageWatcherID` (-2). There are no parameters sent. The Message Watcher is called at startup time and is always present. If you switch to another message watcher (using `set the messageWatcher to "MyWatcher"`), that watcher is loaded immediately. The normal behavior of a message watcher is that it should be inactive when it is invisible.

Variable Watcher

When HyperTalk calls the Variable Watcher to initialize itself, it sets `paramPtr^.paramCount` to `xVariableWatcherID` (-3). There are no parameters sent. The Variable Watcher is called at startup time and is always present. If you switch to another variable watcher (using `set the variableWatcher to "MyWatcher"`), that watcher is loaded immediately. The normal behavior of a variable watcher is that it should be inactive when it is invisible.

Script Editor

When HyperTalk calls a script editor to initialize itself, it sets `paramPtr^.paramCount` to `xScriptEditorID` (-4). There are three parameters sent. `Params[1]` is a zero-terminated handle to the script of the object (unindented). `Params[2]` is a pointer to a Pascal string (`Str255`) containing the name of the window as proposed by HyperTalk (for example, `script of card button id 4 = "Fred"`). `Params[3]` is a pointer to the `XTalkObject` structure describing the object to be edited (see the next section).

Note

There can be multiple script editors open at once, all sharing the same copy of the XCMD resource in memory. For this reason, XCMDs must not write to their own code, or serious problems can result.

---

External Windows 541
APPENDIX A

External Commands and Functions

Debugger

A debugger window is initialized the same as a script editor, with the exception that the paramPtr^.paramCount is set to xDebuggerID (-5).

XTalkObject

In order to increase a script editor’s flexibility, it can communicate with HyperTalk using a special data structure called XTalkObject. Each time a script editor is opened, it is passed a pointer to this structure describing the object whose script is being edited. However, any XCMD can use it as long as the relevant fields are filled in correctly.

```
XTalkObjectPtr = ^XTalkObject;
XTalkObject = RECORD
  objectKind: INTEGER; { stack, bkgnd, card, field, or button }
  stackNum: LongInt; { reference number of the source stack }
  bkgndID: LongInt; 
  cardID: LongInt;
  buttonID: LongInt;
  fieldID: LongInt;
END;
```

objectKind
The type of object: stackObj, bkgndObj, cardObj, fieldObj, buttonObj.

stackNum
The reference number of the stack containing the object. Use StackNameToNum to get the stacknum of a stack when you know the name. Use GetObjectName to get the name of the stack if all you know is the number. If XTalkObject.objectKind = stackObj, this is the last relevant field.

bkgndID
The ID of the background. If XTalkObject.objectKind = bkgndObj, this is the last relevant field. If the object is a card, this is the background to which the card belongs. If the object is a background field or button, this is the ID of the background to which the field or button belongs.

continued
APPENDIX A

External Commands and Functions

cardID
The ID of the card. If XTalkObject.objectKind = cardObj, this is
the last relevant field. If the object is a card field or button, this is the ID of
the card to which the field or button belongs.

buttonID
The buttonID is the ID of the button. If the button is a card button,
cardID contains the ID of the button’s card. If the button is a background
button, bkgndID contains the ID of the button’s background.

fieldID
The fieldID is the ID of the field. If the field is a card field, cardID
contains the ID of the field’s card. If the field is a background field,
bkgndID contains the ID of the field’s background.

Window Layer Management

Within HyperCard, each window resides in one of three layers: the dialog
layer, the miniwindow layer, or the document layer, as shown in Figure A-2.
The front layer, the dialog layer, is reserved for modal dialog boxes. Any window that forces the user to complete a task before continuing, such as the standard file dialog box and HyperCard’s `ask` dialog box, belongs in this layer.

The center layer, the miniwindow layer, is reserved for windows that have a single state, such as HyperCard’s Tools palette, which is always active when it is visible. All of HyperCard’s floating windows, including the Message box and the Tools palette, reside in this layer. Windows in this layer never receive activate events.

The rear layer is the document layer. Windows that have multiple states, active and inactive, such as HyperCard card windows and script editing windows, reside in this layer.

If your window resides in the document layer and you wish to determine whether it’s frontmost in its layer, use the `FrontDocWindow` callback. The Window Manager routines, such as `FindWindow` and `FrontWindow`, operate
APPENDIX A

External Commands and Functions

in the HyperCard window environment. For example, *FrontWindow* returns a pointer to the frontmost window.

HyperCard automatically inserts new external windows in the proper layer according to type (see the description of the *NewXWindow* callback).

An XCMD can determine the layer in which its external windows reside by using the *floating* argument to *GetNewXWindow* and *NewXWindow*. If *floating* is TRUE, the window is placed at the front of the miniwindow layer; otherwise, it is placed at the front of the document layer.

```pascal
FUNCTION NewXWindow(paramPtr: XCmdPtr; boundsRect: Rect; title: Str255; visible: BOOLEAN; procID: INTEGER; color: BOOLEAN; floating: BOOLEAN): WindowPtr;

FUNCTION GetNewXWindow(paramPtr: XCmdPtr; templateType: ResType; templateID: INTEGER; color: BOOLEAN; floating: BOOLEAN): WindowPtr;
```

The dialog layer isn’t really a layer at all; rather, it is reserved for use by modal windows and dialog boxes. Use the Macintosh Window Manager’s routines to create and remove windows in the dialog layer, but do not leave them up after the XCMD has returned.

**Flash: An Example XCMD**

A simple example external command included with HyperCard is *flash*, which inverts the screen display (changes the black pixels to white and vice versa) a specified number of times. A version of *flash* written and compiled in MPW Pascal has already been attached to the HyperCard application file (that is, to HyperCard itself).

*Flash* is invoked from HyperCard just like a HyperTalk command. That is, you send the message *flash* to HyperCard from the Message box or from an executing script. The *flash* message takes one parameter: an integer. The *flash* XCMD inverts the screen display twice that many times. For example, the following handler, in response to a *mouseUp* message, sends the *flash*
message and its parameter. When the message reaches HyperCard, it invokes the `flash` external command, which inverts the screen display 20 times:

```pascal
on mouseUp
  flash 10
end mouseUp
```

The screen display flashes (is inverted and inverted back again) 10 times.

**Flash Listing in MPW Pascal**

Here’s the Pascal listing for `flash`:

```pascal
(*
 * Flash.p-A sample HyperCard XCMD to highlight the screen
 * Copyright Apple Computer, Inc. 1987-1993.
 * All Rights Reserved.
 *
 * Build instructions for MPW 3.3 (puts XCMD in a ResEdit file):
 *
 Pascal Flash.p -o Flash.p.o
 Link -t rsrc -c RSED -rt XCMD=0 -m ENTRYPOINT -sg Flash \$
   Flash.p.o \$
   "(Libraries)"HyperXLib.o \$
 -o "Flash XCMD"
 *
 *)

{$R-}$

{$S Flash} { Segment name must be same as command name }
(*
 * DummyUnit is what HyperTalk jumps to when running the XCMD.
 * Also note that XCMDs do not support their own A5 World,
A P P E N D I X  A

External Commands and Functions

* thus NO GLOBAL VARIABLES are allowed.
*)

UNIT DummyUnit;

INTERFACE

USES Types, QuickDraw, SysEqu, HyperXCmd;

PROCEDURE EntryPoint(paramPtr: XCmdPtr);

IMPLEMENTATION

PROCEDURE Flash(paramPtr: XCmdPtr); FORWARD;

PROCEDURE EntryPoint(paramPtr: XCmdPtr);
BEGIN
  Flash(paramPtr);
END;

PROCEDURE Flash(paramPtr: XCmdPtr);

VAR flashCount: INTEGER;
  again: INTEGER;
  port: GrafPtr;
  str: Str255;
  when: LONGINT;
  ticksPtr: ^LONGINT;

BEGIN
  flashCount := 0;
  IF (paramPtr^.paramCount = 1) THEN BEGIN
    (first param is flash count)
    ZeroToPas(paramPtr, paramPtr^.params[1]^, str);
Flash: An Example XCMD

Flash: An Example XCMD
APPENDIX A

External Commands and Functions

flashCount := StrToNum(paramPtr, str);
END;

IF (paramPtr^.paramCount <> 1) OR (flashCount < 1) THEN
    flashCount := 3;

GetPort(port);
ticksPtr := POINTER(Ticks);{ 'Ticks' defined in SysEqu.p }

FOR again := 1 TO 2 * flashCount DO BEGIN
    when := ticksPtr^ + 4;
    InvertRect(port^.portRect);
    REPEAT UNTIL ticksPtr^ >= when;
END;
END;
END.

Flash Listing in MPW C

Here's a version of flash written in MPW C:

/*
   Flash.c -A sample HyperCard XCMD to highlight the screen
   Copyright Apple Computer, Inc. 1987-1993.
   All Rights Reserved.

Example:CFlash 5

Build instructions for MPW 3.3:
   C Flash.c -o Flash.c.o
   Link -t rsr -c RSED -rt XCMD=0 -m MAIN -sg CFlash d
   Flash.c.o d
   "(Libraries)"HyperXLib.o d
   -o "CFlash XCMD"
*/
APPENDIX A

External Commands and Functions

Build instructions for THINK C 6.0:
Build as a Code Resource of type XCMD;
add MacTraps and HyperXLib libraries to project;
will not need compiler's standard Prefix.

*/
#include<Types.h>
#include<Quickdraw.h>
#include<SysEqu.h>
#include<HyperXCmd.h>

/*
Your routine MUST be the first code that is generated in the file,
as HyperTalk simply JSRs to the start of the XCMD segment in
memory. Note that XCMDs do not support their own A5 World,
thus NO GLOBAL VARIABLES are allowed.
*/
pascal void main( XCmdPtr paramPtr )
{
    short flashCount = 0, again;
    GrafPtr port;
    Str255 str;
    long when;
    unsigned long *ticksPtr;

    if ( paramPtr->paramCount == 1)
    {
        /* get flash count*/
        ZeroToPas(paramPtr, *(paramPtr->params[0]), (StringPtr)str);
        /* convert string to number*/
        flashCount = StrToNum(paramPtr, (StringPtr)str);
    }
APPENDIX A

External Commands and Functions

if ((paramPtr->paramCount != 1) || (flashCount < 1))
    flashCount = 3;

GetPort(&port);
ticksPtr = (unsigned long *)Ticks;
flashCount *= 2;

for (again = 1; again <= flashCount; again++)
{
    when = *ticksPtr + 4;
    InvertRect( &port->portRect);
    while ( *ticksPtr < when );
}

Flash Listing in 68000 Assembly Language

Here's the 68000 assembly-language listing for flash:

* * Flash.a
* * A sample HyperCard XCMD in 68000 Assembly
* * Copyright Apple Computer, Inc. 1988-1993
* * All Rights Reserved.
* *
* This version of the Flash XCMD, 'AFlash', only looks at the first
* character of parameter 1. It does not have the timing code of the
* Pascal and C versions.
* *
* Build Instructions:
* *
* Asm Flash.a -o Flash.a.o
* Link -t rsrc -c RSED -rt XCMD=7 -sg AFlash Flash.a.o \o
* -o "AFlash XCMD"
APPENDIX A

External Commands and Functions

* * *

INCLUDE 'QuickEqu.a'
INCLUDE 'Traps.a'

SEG 'AFlash' ; Segname must be same as command name

PROC ; uses a0, a1, d1
AFlash
    link a6, #-4 ; save
    move.l d4, -(sp) ; save
    move.w #3, d4 ; StrToNum default result
    move.l 8(a6), a0 ; get paramPtr in a temp reg
    move.l 2(a0), a1 ; get handle to flashCount (as C string)
    cmpa.l #0, a1
    beq.s @2 ; if handle NIL, use default
    move.l (a1), a1 ; deref
    @1 move.b (a1)+, d1 ; get a char
    cmp.b #'1', d1 ; test for a non-0 digit
    blt.s @2 ; less than valid
    cmp.b #'9', d1
    bgt.s @2 ; greater than valid
    and.w #$000F, d1 ; mask to value of legal char
    move.w d1, d4 ; stick value into result

@2 pea -4(a6) ; var result of GetPort
    _GetPort
    bra.s @4 ; get into DBRA loop

@3 move.l -4(a6), a0 ; get port
    pea portRect(a0) ; address or portRect
    _InverRect
    move.l -4(a6), a0 ; get port
    pea portRect(a0) ; address or portRect
_InverRect
@4 dbra d4, @3

move.l (sp)+,d4  ; restore
unlk a6

move.l (sp)+,a0  ; rts Pascal style
add.l #4,a7
jmp (a0)

END
This appendix describes HyperTalk’s built-in constants. A constant is a named value that never changes. It’s different from a variable because you can’t change it, and it’s different from a literal because it does not require quotation marks.

The values of some constants are the string of characters making up the name, while others are different. In some cases, it’s more convenient to use a constant (such as \texttt{pi}) in place of a long string (such as 3.14159265358979323846). In other cases, it’s more convenient to use a constant (such as \texttt{formFeed}) because the only other way to enter that character is with the \texttt{numToChar} function, requiring that you know the ASCII number of the character (as in the \texttt{numToChar} of 12).

You can’t give a variable a name that is the same as that of any built-in constant; if you try, HyperCard displays an error dialog box.

Table B-1 is a list of all the built-in constants in HyperTalk.

<table>
<thead>
<tr>
<th>Constant name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>colon</td>
<td>The &quot;:&quot; character is equivalent to ASCII 58.</td>
</tr>
<tr>
<td>comma</td>
<td>The &quot;,&quot; character, ASCII 44, is used as the default item delimiter by HyperCard.</td>
</tr>
<tr>
<td>down</td>
<td>The value returned by the \texttt{commandKey}, \texttt{mouse}, \texttt{optionKey}, or \texttt{shiftKey} function when the named key (or button, in the case of \texttt{mouse}), is currently pressed. Its value is the same as the literal &quot;down&quot;.</td>
</tr>
<tr>
<td>empty</td>
<td>The null string; the same as the literal &quot;&quot;.</td>
</tr>
<tr>
<td>eof</td>
<td>The end-of-file condition; used with the \texttt{read} and \texttt{write} commands.</td>
</tr>
</tbody>
</table>

\textit{continued}
### Constants

The opposite of `true`; one of the states tested by the `if` control structure and one of the possible results of evaluation of a logical expression. Its value is the same as the literal "false".

The form feed character (ASCII 12), which starts a new page in some file formats.

The line feed character (ASCII 10), which starts a new line in some file formats.

The mathematical value pi to 20 decimal places, denoting the ratio of the circumference of a circle to its diameter, represented by the number 3.14159265358979323846.

The double quotation mark character. It is needed to build a string containing quotation marks because they are stripped out of the string when literals are evaluated:

```plaintext
put "george" into It  -- quotation marks are not in It
put quote & "george" & quote into It  -- quotation marks are in It
```

The return character (ASCII 13), which delimits the lines of a string or container.

The space character (ASCII 32), the same as the literal " ".

The horizontal tab character (ASCII 9).

The opposite of `false`; one of the states tested by the `if` control structure and one of the possible results of evaluation of a logical expression. Its value is the same as the literal "true".

The value returned by the `commandKey`, `mouse`, `optionKey`, or `shiftKey` function when the named key (or button, in the case of `mouse`) is not currently pressed. Its value is the same as the literal "up".

The numbers 0 through 10.

<table>
<thead>
<tr>
<th>Constant name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>false</strong></td>
<td>The opposite of <code>true</code>; one of the states tested by the <code>if</code> control structure and one of the possible results of evaluation of a logical expression. Its value is the same as the literal &quot;false&quot;.</td>
</tr>
<tr>
<td><strong>formFeed</strong></td>
<td>The form feed character (ASCII 12), which starts a new page in some file formats.</td>
</tr>
<tr>
<td><strong>lineFeed</strong></td>
<td>The line feed character (ASCII 10), which starts a new line in some file formats.</td>
</tr>
<tr>
<td><strong>pi</strong></td>
<td>The mathematical value pi to 20 decimal places, denoting the ratio of the circumference of a circle to its diameter, represented by the number 3.14159265358979323846.</td>
</tr>
<tr>
<td><strong>quote</strong></td>
<td>The double quotation mark character. It is needed to build a string containing quotation marks because they are stripped out of the string when literals are evaluated:</td>
</tr>
</tbody>
</table>
|               | ```plaintext
|               | put "george" into It  -- quotation marks are not in It
|               | put quote & "george" & quote into It  -- quotation marks are in It
| **return**    | The return character (ASCII 13), which delimits the lines of a string or container. |
| **space**     | The space character (ASCII 32), the same as the literal " ". |
| **tab**       | The horizontal tab character (ASCII 9). |
| **true**      | The opposite of `false`; one of the states tested by the `if` control structure and one of the possible results of evaluation of a logical expression. Its value is the same as the literal "true". |
| **up**        | The value returned by the `commandKey`, `mouse`, `optionKey`, or `shiftKey` function when the named key (or button, in the case of `mouse`) is not currently pressed. Its value is the same as the literal "up". |
| **zero..ten** | The numbers 0 through 10. |
This appendix provides scripting hints and other techniques for getting the most performance out of HyperCard applications.

One of the key methods for increasing the speed of HyperCard is avoiding disk accesses whenever possible. If you remember a few good scripting techniques, you can keep disk access at a minimum. To avoid excessive disk accesses, do the following:

- Change stacks as seldom as possible.
- Use variables instead of fields for all operations.
- Refer to a remote card rather than going there.

Other good scripting techniques that generally improve the performance of HyperCard are as follows:

- Migrate to XCMDs and XFCNs for highly repetitive tasks, such as sorting.
- Set lockScreen to true to avoid needless redrawing.
- Set lockMessages to true to save time during card-to-card data collection.
- Combine multiple messages.
- Take unnecessary code out of loops.
- Always use quoted literals.
- Use in-line statements rather than handler calls.
- Do complex calculations once.
- Watch overuse of variable references.
- Do visible work first.

Each of the techniques is described in the sections that follow.
Enhancing the Execution Speed of HyperCard

**Change Stacks as Seldom as Possible**

Changing stacks means going to a disk, hard disk, or CD-ROM to retrieve information and, in the case of a floppy disk and hard disk, to store information. The disk accessing process takes more time than any other HyperCard operation.

When you need to read or write data from another stack, go the stack only once. Whenever possible, get or put everything you need at the same time.

Keep related information in the same stack. This includes data that you regularly sort, search, or move between. If you plan to use cards with substantially different appearances within the same application (a HyperCard application could consist of several stacks) and you need to cross boundaries frequently, use multiple backgrounds in the same stack rather than separate stacks.

However, there are exceptions to this rule. Because you might want to put the finished stack on a floppy disk for ease of backup and transportation, you need to keep an eye on the size of stacks as you build them. If you have too much related data to put on a single floppy, you have to use multiple stacks rather than one stack. Alternatively, you could put the final stack on a CD-ROM.

Another exception to the single stack rule is command operations. The HyperCard sort and find commands can operate on the entire stack, and you may not want the entire stack sorted or searched.

**Use Variables, Not Fields, for Operations**

Whenever possible, do operations such as sorting, data collection, and calculations in variables rather than fields. HyperCard operations are much faster in variables.

Fields are for the display of data and for long-term storage; HyperCard keeps field information stored on disk and draws that information onto the screen. Variables, in contrast, are in RAM and are for storage and manipulation of transient data—data that doesn’t appear on the screen and is lost when you quit HyperCard. For example, collecting data on different cards into variables is approximately 50 percent faster than collecting the same data into a field.

Do calculations in variables to get faster results: for example, adding the contents of a series of fields from different cards. Everything that goes into fields, including numbers, is converted to a string, regardless of its original format. The numeric content of variables, in contrast, is stored in binary format, making calculation more precise and less time consuming.
Enhancing the Execution Speed of HyperCard

Refer to a Remote Card Rather Than Going There

Referring to a remote card is generally faster than going there but is dependent on the number of fields referred to per card. For example, when collecting data from fewer than 10 fields, referring to a remote card in the current stack is faster than going to the card. However, when collecting data from 10 fields or more, it is faster to go to that card and then collect the data. For example, this script refers to a remote card containing data needed from the fields on that card. The script works in the current card of the current stack.

```plaintext
on collectGoodies
    repeat with FieldNum = 1 to 8
        put (line 2 in card field FieldNum of card 4) & return &
        & (line 3 in card field FieldNum of card 4) & return after
        field 3 of this card
    end repeat
end collectGoodies

If the data is needed from more than 10 fields on a remote card, the script might look like this:

```plaintext
on collectGoodies
    set lockMessages to true
    push card
    lock screen
    go to card 4
    repeat with FieldNum = 1 to the number of bkgnd fields
        put (line 2 in card field FieldNum of card 4) & return &
        (line 3 in card field FieldNum of card 4) & return &
        after var
    end repeat
    pop card
    put var into field 3
    unlock screen
    set lockMessages to false
end collectGoodies
```
APPENDIX C
Enhancing the Execution Speed of HyperCard

Migrate to XCMDs and XFCNs for Repetitive Tasks
No matter how efficient your HyperTalk code is, sorting 500 items is still going to take a long time. Repetitive time-intensive tasks are best handled with XCMDs and XFCNs. XCMDs and XFCNs have the same calling interface as any other HyperCard command, so are not any harder to use in a script.

If you do not have the programming experience in another high-level language to create an appropriate XCMD for your application, public domain and shareware XCMDs may exist that provide all of the required functions that your application needs. These XCMDs can be found through HyperCard user groups, user bulletin boards, and other information services.

Set LockScreen to True to Avoid Needless Redrawing
Redrawing the screen takes time, and it makes no sense to change the screen if all you are doing is going to another card to collect data. Set lockScreen to true (or use the lock command) while your scripts collect data by going from card to card and stack to stack.

See the second script example in the section “Refer to a Remote Card Rather Than Going There.”

Set LockMessages to True During Card-to-Card Data Collection
The lock messages command prevents HyperCard from sending the six open and close system messages associated with cards, backgrounds, and stacks. See Chapter 8, “System Messages,” for more information about HyperCard system messages.

Combine Multiple Messages
Message sending is relatively time consuming because each message traverses the entire message-passing path. The techniques described here avoid excessive message sending and can save lots of time.

Operators are interpreted directly and don’t incur the cost of message sending. Using the operators is in, is not in, and contains is faster than using offset() because they are interpreted directly and are not sent as messages.
APPENDIX C

Enhancing the Execution Speed of HyperCard

Calling a built-in function with the or of is faster than calling with “()” because “()” functions traverse the message path. Therefore, using date() is slower than using the date.

Use put to move values directly, rather than using get to store a value in it and then using put to move the value out of it. As an example,

get x
put It into y

is slower than

put x into y

Take Unnecessary Code Out of Loops

The reason for removing unnecessary code from loops is fairly obvious but frequently overlooked. Shorter handlers with fewer lines of code take less time to run than longer handlers. This fact is magnified in loops. For example, a loop that comprises 6 lines and runs six iterations is equivalent to 25 lines of code. Every extra line in that six-iteration loop counts as an additional 6 lines. Unwrap your code where possible and avoid unnecessary lines. For example, don’t start a loop with the line set the cursor to watch, because you get that for free.

Use In-Line Statements Rather Than Handler Calls

It always takes time to get from one place to another and back again in any HyperCard application and script. To save time, put all of your code in-line. Create utility handlers where possible to avoid calling back to handlers in the main module. Putting a lot of your code in-line can, however, make your code hard to read. You should consider whether the increase in speed is worth having code that is hard to read. A good solution is to save most of the in-line code for repetitive time-consuming tasks that you haven’t written XCMDs to handle.
APPENDIX C

Enhancing the Execution Speed of HyperCard

Do Complex Calculations Once

When you’ve figured something out, put the results in a variable, then refer to the variable. This also includes parameter values. Retrieve parameter values once rather than many times (assuming you know the values haven’t changed). This technique results in the greatest time savings within loops.

Watch Overuse of Variable References

The speed increases seen by avoiding overusing variable references are minimal, but it is good scripting technique. The line `add x to y` is slightly faster than `put x + y into y`, because the former has fewer variable references (two) than the latter (three).
This appendix lists the character assignments for the 256 single-byte character values used by the Macintosh.

There are 256 possible 8-bit binary values, from 00000000 to 11111111. Of these, the first 128 (from 00000000 to 01111111) have been assigned to a standard set of characters and commands used in data processing and communication. These assignments form the ASCII character set. (ASCII stands for American Standard Code for Information Interchange.)

The remaining 128 binary values, those for which the most significant bit (first digit) is 1 instead of 0, are not assigned in the ASCII standard. Because they have higher numerical values than the first 128 characters, they are often referred to as high-ASCII characters.

This appendix lists all character values by their decimal equivalent.

Table D-1 lists the first 32 characters, the control characters, which have no printable-character representation, with the standard abbreviation for each and its meaning.

<table>
<thead>
<tr>
<th>Value</th>
<th>Name</th>
<th>Meaning</th>
<th>Value</th>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NUL</td>
<td>Null</td>
<td>8</td>
<td>BS</td>
<td>Backspace</td>
</tr>
<tr>
<td>1</td>
<td>SOH</td>
<td>Start of heading</td>
<td>9</td>
<td>HT</td>
<td>Horizontal tab</td>
</tr>
<tr>
<td>2</td>
<td>STX</td>
<td>Start of text</td>
<td>10</td>
<td>LF</td>
<td>Line feed</td>
</tr>
<tr>
<td>3</td>
<td>ETX</td>
<td>End of text</td>
<td>11</td>
<td>VT</td>
<td>Vertical tab</td>
</tr>
<tr>
<td>4</td>
<td>EOT</td>
<td>End of transmission</td>
<td>12</td>
<td>FF</td>
<td>Form feed</td>
</tr>
<tr>
<td>5</td>
<td>ENQ</td>
<td>Enquiry</td>
<td>13</td>
<td>CR</td>
<td>Carriage return</td>
</tr>
<tr>
<td>6</td>
<td>ACK</td>
<td>Acknowledge</td>
<td>14</td>
<td>SO</td>
<td>Shift out</td>
</tr>
<tr>
<td>7</td>
<td>BEL</td>
<td>Bell</td>
<td>15</td>
<td>SI</td>
<td>Shift in</td>
</tr>
</tbody>
</table>

*continued*
Table D-2 lists the remaining 224 character values with the characters to which they are assigned in the Macintosh Courier font.

Table D-2  Character assignments in Macintosh Courier font

<table>
<thead>
<tr>
<th>Value</th>
<th>Character</th>
<th>Value</th>
<th>Character</th>
<th>Value</th>
<th>Character</th>
<th>Value</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>Space</td>
<td>43</td>
<td>+</td>
<td>54</td>
<td>6</td>
<td>65</td>
<td>A</td>
</tr>
<tr>
<td>33</td>
<td>!</td>
<td>44</td>
<td>,</td>
<td>55</td>
<td>7</td>
<td>66</td>
<td>B</td>
</tr>
<tr>
<td>34</td>
<td>&quot;</td>
<td>45</td>
<td>-</td>
<td>56</td>
<td>8</td>
<td>67</td>
<td>C</td>
</tr>
<tr>
<td>35</td>
<td>#</td>
<td>46</td>
<td>.</td>
<td>57</td>
<td>9</td>
<td>68</td>
<td>D</td>
</tr>
<tr>
<td>36</td>
<td>$</td>
<td>47</td>
<td>/</td>
<td>58</td>
<td>:</td>
<td>69</td>
<td>E</td>
</tr>
<tr>
<td>37</td>
<td>%</td>
<td>48</td>
<td>0</td>
<td>59</td>
<td>;</td>
<td>70</td>
<td>F</td>
</tr>
<tr>
<td>38</td>
<td>&amp;</td>
<td>49</td>
<td>1</td>
<td>60</td>
<td>&lt;</td>
<td>71</td>
<td>G</td>
</tr>
<tr>
<td>39</td>
<td>'</td>
<td>50</td>
<td>2</td>
<td>61</td>
<td>=</td>
<td>72</td>
<td>H</td>
</tr>
<tr>
<td>40</td>
<td>(</td>
<td>51</td>
<td>3</td>
<td>62</td>
<td>&gt;</td>
<td>73</td>
<td>I</td>
</tr>
<tr>
<td>41</td>
<td>)</td>
<td>52</td>
<td>4</td>
<td>63</td>
<td>?</td>
<td>74</td>
<td>J</td>
</tr>
<tr>
<td>42</td>
<td>*</td>
<td>53</td>
<td>5</td>
<td>64</td>
<td>@</td>
<td>75</td>
<td>K</td>
</tr>
</tbody>
</table>

continued
<table>
<thead>
<tr>
<th>Value</th>
<th>Character</th>
<th>Value</th>
<th>Character</th>
<th>Value</th>
<th>Character</th>
<th>Value</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td>L</td>
<td>101</td>
<td>e</td>
<td>126</td>
<td>~</td>
<td>151</td>
<td>ó</td>
</tr>
<tr>
<td>77</td>
<td>M</td>
<td>102</td>
<td>f</td>
<td>127</td>
<td>Del</td>
<td>152</td>
<td>ò</td>
</tr>
<tr>
<td>78</td>
<td>N</td>
<td>103</td>
<td>g</td>
<td>128</td>
<td>Ä</td>
<td>153</td>
<td>ö</td>
</tr>
<tr>
<td>79</td>
<td>O</td>
<td>104</td>
<td>h</td>
<td>129</td>
<td>Å</td>
<td>154</td>
<td>ö</td>
</tr>
<tr>
<td>80</td>
<td>P</td>
<td>105</td>
<td>i</td>
<td>130</td>
<td>Ç</td>
<td>155</td>
<td>ō</td>
</tr>
<tr>
<td>81</td>
<td>Q</td>
<td>106</td>
<td>j</td>
<td>131</td>
<td>É</td>
<td>156</td>
<td>ù</td>
</tr>
<tr>
<td>82</td>
<td>R</td>
<td>107</td>
<td>k</td>
<td>132</td>
<td>Ñ</td>
<td>157</td>
<td>ũ</td>
</tr>
<tr>
<td>83</td>
<td>S</td>
<td>108</td>
<td>l</td>
<td>133</td>
<td>Ö</td>
<td>158</td>
<td>ũ</td>
</tr>
<tr>
<td>84</td>
<td>T</td>
<td>109</td>
<td>m</td>
<td>134</td>
<td>Ü</td>
<td>159</td>
<td>ū</td>
</tr>
<tr>
<td>85</td>
<td>U</td>
<td>110</td>
<td>n</td>
<td>135</td>
<td>á</td>
<td>160</td>
<td>†</td>
</tr>
<tr>
<td>86</td>
<td>V</td>
<td>111</td>
<td>o</td>
<td>136</td>
<td>à</td>
<td>161</td>
<td>°</td>
</tr>
<tr>
<td>87</td>
<td>W</td>
<td>112</td>
<td>p</td>
<td>137</td>
<td>à</td>
<td>162</td>
<td>€</td>
</tr>
<tr>
<td>88</td>
<td>X</td>
<td>113</td>
<td>q</td>
<td>138</td>
<td>ä</td>
<td>163</td>
<td>£</td>
</tr>
<tr>
<td>89</td>
<td>Y</td>
<td>114</td>
<td>r</td>
<td>139</td>
<td>ã</td>
<td>164</td>
<td>§</td>
</tr>
<tr>
<td>90</td>
<td>Z</td>
<td>115</td>
<td>s</td>
<td>140</td>
<td>å</td>
<td>165</td>
<td>•</td>
</tr>
<tr>
<td>91</td>
<td>[</td>
<td>116</td>
<td>t</td>
<td>141</td>
<td>ç</td>
<td>166</td>
<td>¶</td>
</tr>
<tr>
<td>92</td>
<td>\</td>
<td>117</td>
<td>u</td>
<td>142</td>
<td>é</td>
<td>167</td>
<td>ß</td>
</tr>
<tr>
<td>93</td>
<td>]</td>
<td>118</td>
<td>v</td>
<td>143</td>
<td>è</td>
<td>168</td>
<td>®</td>
</tr>
<tr>
<td>94</td>
<td>^</td>
<td>119</td>
<td>w</td>
<td>144</td>
<td>ë</td>
<td>169</td>
<td>©</td>
</tr>
<tr>
<td>95</td>
<td>_</td>
<td>120</td>
<td>x</td>
<td>145</td>
<td>ë</td>
<td>170</td>
<td>™</td>
</tr>
<tr>
<td>96</td>
<td>`</td>
<td>121</td>
<td>y</td>
<td>146</td>
<td>í</td>
<td>171</td>
<td>'</td>
</tr>
<tr>
<td>97</td>
<td>a</td>
<td>122</td>
<td>z</td>
<td>147</td>
<td>î</td>
<td>172</td>
<td>&quot;</td>
</tr>
<tr>
<td>98</td>
<td>b</td>
<td>123</td>
<td>{</td>
<td>148</td>
<td>î</td>
<td>173</td>
<td>≠</td>
</tr>
<tr>
<td>99</td>
<td>c</td>
<td>124</td>
<td></td>
<td></td>
<td>149</td>
<td>i</td>
<td>174</td>
</tr>
<tr>
<td>100</td>
<td>d</td>
<td>125</td>
<td>}</td>
<td>150</td>
<td>n</td>
<td>175</td>
<td>Ø</td>
</tr>
</tbody>
</table>

continued
## Table D-2  Character assignments in Macintosh Courier font  (continued)

<table>
<thead>
<tr>
<th>Value</th>
<th>Character</th>
<th>Value</th>
<th>Character</th>
<th>Value</th>
<th>Character</th>
<th>Value</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>176</td>
<td>¥</td>
<td>200</td>
<td>»</td>
<td>224</td>
<td>†</td>
<td>248</td>
<td>&quot;</td>
</tr>
<tr>
<td>177</td>
<td>±</td>
<td>201</td>
<td>…</td>
<td>225</td>
<td>·</td>
<td>249</td>
<td>&quot;</td>
</tr>
<tr>
<td>178</td>
<td>≤</td>
<td>202</td>
<td>**</td>
<td>226</td>
<td>,</td>
<td>250</td>
<td>'</td>
</tr>
<tr>
<td>179</td>
<td>≥</td>
<td>203</td>
<td>Å</td>
<td>227</td>
<td>&quot;</td>
<td>251</td>
<td>&quot;</td>
</tr>
<tr>
<td>180</td>
<td>¥</td>
<td>204</td>
<td>Á</td>
<td>228</td>
<td>%</td>
<td>252</td>
<td>'</td>
</tr>
<tr>
<td>181</td>
<td>µ</td>
<td>205</td>
<td>Ō</td>
<td>229</td>
<td>Å</td>
<td>253</td>
<td>&quot;</td>
</tr>
<tr>
<td>182</td>
<td>Ω</td>
<td>206</td>
<td>©</td>
<td>230</td>
<td>Œ</td>
<td>254</td>
<td>'</td>
</tr>
<tr>
<td>183</td>
<td>Σ</td>
<td>207</td>
<td>æ</td>
<td>231</td>
<td>Å</td>
<td>255</td>
<td>&quot;</td>
</tr>
<tr>
<td>184</td>
<td>Π</td>
<td>208</td>
<td>–</td>
<td>232</td>
<td>Ê</td>
<td></td>
<td></td>
</tr>
<tr>
<td>185</td>
<td>π</td>
<td>209</td>
<td>–</td>
<td>233</td>
<td>Ê</td>
<td></td>
<td></td>
</tr>
<tr>
<td>186</td>
<td>¡</td>
<td>210</td>
<td>“</td>
<td>234</td>
<td>ñ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>187</td>
<td>&quot;</td>
<td>211</td>
<td>”</td>
<td>235</td>
<td>ñ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>188</td>
<td>o</td>
<td>212</td>
<td>’</td>
<td>236</td>
<td>ñ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>189</td>
<td>Ω</td>
<td>213</td>
<td>’</td>
<td>237</td>
<td>ñ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>190</td>
<td>æ</td>
<td>214</td>
<td>÷</td>
<td>238</td>
<td>õ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>191</td>
<td>ø</td>
<td>215</td>
<td>Ø</td>
<td>239</td>
<td>õ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>192</td>
<td>é</td>
<td>216</td>
<td>ý</td>
<td>240</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>193</td>
<td>i</td>
<td>217</td>
<td>ÿ</td>
<td>241</td>
<td>ò</td>
<td></td>
<td></td>
</tr>
<tr>
<td>194</td>
<td>ŋ</td>
<td>218</td>
<td>¿</td>
<td>242</td>
<td>û</td>
<td></td>
<td></td>
</tr>
<tr>
<td>195</td>
<td>√</td>
<td>219</td>
<td>¿</td>
<td>243</td>
<td>û</td>
<td></td>
<td></td>
</tr>
<tr>
<td>196</td>
<td>f</td>
<td>220</td>
<td>‹</td>
<td>244</td>
<td>û</td>
<td></td>
<td></td>
</tr>
<tr>
<td>197</td>
<td>=</td>
<td>221</td>
<td>›</td>
<td>245</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>198</td>
<td>Δ</td>
<td>222</td>
<td>fi</td>
<td>246</td>
<td>&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>199</td>
<td>«</td>
<td>223</td>
<td>f1</td>
<td>247</td>
<td>&quot;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Stands for a nonbreaking space**
Operator Precedence Table

Table E-1 shows the order of precedence of HyperTalk’s operators. In a complex expression containing more than one operator, HyperTalk performs the operation indicated by operators with lower-numbered precedence before those with higher-numbered precedence. Operators of equal precedence are evaluated left-to-right, except for exponentiation, which goes right-to-left. If you use parentheses, HyperTalk evaluates the innermost parenthetical expression first.

Chapter 7 discusses expression evaluation.

<table>
<thead>
<tr>
<th>Order</th>
<th>Operators</th>
<th>Type of operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( )</td>
<td>Grouping</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>Minus sign for numbers</td>
</tr>
<tr>
<td></td>
<td>not</td>
<td>Logical negation for Boolean values</td>
</tr>
<tr>
<td></td>
<td>there is a</td>
<td>Comparison for HyperCard items</td>
</tr>
<tr>
<td></td>
<td>there is an</td>
<td>Comparison for HyperCard items</td>
</tr>
<tr>
<td></td>
<td>there is not a</td>
<td>Comparison for HyperCard items</td>
</tr>
<tr>
<td></td>
<td>there is not an</td>
<td>Comparison for HyperCard items</td>
</tr>
<tr>
<td>3</td>
<td>^</td>
<td>Exponentiation for numbers</td>
</tr>
<tr>
<td>4</td>
<td>* / div mod</td>
<td>Multiplication and division for numbers</td>
</tr>
<tr>
<td>5</td>
<td>+ -</td>
<td>Addition and subtraction for numbers</td>
</tr>
<tr>
<td>6</td>
<td>&amp; &amp; &amp; &amp;</td>
<td>Concatenation of text</td>
</tr>
<tr>
<td>7</td>
<td>&gt; &lt; &lt;= &gt;= ≤ ≥ is in</td>
<td>Comparison for numbers or text</td>
</tr>
</tbody>
</table>

*continued*
### Table E-1  Operator precedence (continued)

<table>
<thead>
<tr>
<th>Order</th>
<th>Operators</th>
<th>Type of operator</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>contains</td>
<td>Comparison for text</td>
</tr>
<tr>
<td></td>
<td>is within</td>
<td>Boolean test for point within rectangle</td>
</tr>
<tr>
<td></td>
<td>is not within</td>
<td>Boolean test for point within rectangle</td>
</tr>
<tr>
<td></td>
<td>is not in</td>
<td>Comparison for text</td>
</tr>
<tr>
<td></td>
<td>is a</td>
<td>Comparison for types</td>
</tr>
<tr>
<td></td>
<td>is an</td>
<td>Comparison for types</td>
</tr>
<tr>
<td></td>
<td>is not a</td>
<td>Comparison for types</td>
</tr>
<tr>
<td></td>
<td>is not an</td>
<td>Comparison for types</td>
</tr>
<tr>
<td>8</td>
<td>= is</td>
<td>Comparison for numbers or text</td>
</tr>
<tr>
<td></td>
<td>is not &lt;&gt; #</td>
<td>Comparison for numbers or text</td>
</tr>
<tr>
<td>9</td>
<td>and</td>
<td>Logical for Boolean values</td>
</tr>
<tr>
<td>10</td>
<td>or</td>
<td>Logical for Boolean values</td>
</tr>
</tbody>
</table>
HyperCard Synonyms

Table F-1 lists the alternative ways that HyperTalk terms can be used.

<table>
<thead>
<tr>
<th>Synonym</th>
<th>Term</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>abbr</td>
<td>abbrev</td>
<td>abbreviated</td>
</tr>
<tr>
<td>bg</td>
<td>bkgnd</td>
<td>background</td>
</tr>
<tr>
<td>bgs</td>
<td>bkgnds</td>
<td>backgrounds</td>
</tr>
<tr>
<td>botRight</td>
<td>bottomRight</td>
<td></td>
</tr>
<tr>
<td>btn</td>
<td>button</td>
<td></td>
</tr>
<tr>
<td>btns</td>
<td>buttons</td>
<td></td>
</tr>
<tr>
<td>cd</td>
<td>card</td>
<td></td>
</tr>
<tr>
<td>cds</td>
<td>cards</td>
<td></td>
</tr>
<tr>
<td>char</td>
<td>character</td>
<td></td>
</tr>
<tr>
<td>chars</td>
<td>characters</td>
<td></td>
</tr>
<tr>
<td>fld</td>
<td>field</td>
<td></td>
</tr>
<tr>
<td>flds</td>
<td>fields</td>
<td></td>
</tr>
<tr>
<td>grey</td>
<td>gray</td>
<td></td>
</tr>
<tr>
<td>hilite</td>
<td>highlight</td>
<td>highlite</td>
</tr>
<tr>
<td>in</td>
<td>of</td>
<td>hilight</td>
</tr>
<tr>
<td>loc</td>
<td>location</td>
<td></td>
</tr>
<tr>
<td>mid</td>
<td>middle</td>
<td></td>
</tr>
<tr>
<td>msg</td>
<td>message</td>
<td>message box</td>
</tr>
</tbody>
</table>

continued
## Table F-1  HyperTalk synonyms (continued)

<table>
<thead>
<tr>
<th>Synonym</th>
<th>Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>msg watcher</td>
<td>message watcher</td>
</tr>
<tr>
<td>part</td>
<td>button</td>
</tr>
<tr>
<td>poly</td>
<td>polygon</td>
</tr>
<tr>
<td>prev</td>
<td>previous</td>
</tr>
<tr>
<td>rect</td>
<td>rectangle</td>
</tr>
<tr>
<td>reg</td>
<td>regular</td>
</tr>
<tr>
<td>sec</td>
<td>secs</td>
</tr>
<tr>
<td>spray</td>
<td>spray can</td>
</tr>
<tr>
<td>tick</td>
<td>ticks</td>
</tr>
</tbody>
</table>
This appendix lists various minimum and maximum sizes and numbers of elements defined in HyperCard.

The maximum limits shown in Table G-1 are theoretical. Some of them are lower in practice. For example, HyperCard currently brings an entire card into memory at once, so the maximum size of a card is limited by available memory. It’s possible that a card with a lot of text and long scripts, created while running HyperCard on a Macintosh with 2 MB of RAM, could not be opened on a Macintosh with 1 MB. The current useful size of a card (or background) is therefore between 50 and 100 KB.

The term part, in this appendix and internally in HyperCard, refers to buttons or fields. The value represented by LongInt is $2,147,483,647$; the value represented by Integer is $32,767$.

**Table G-1** HyperCard limits

<table>
<thead>
<tr>
<th>Item</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Stack limits</strong></td>
<td></td>
</tr>
<tr>
<td>Stack size</td>
<td>512 MB</td>
</tr>
<tr>
<td>Minimum stack size</td>
<td>4896 bytes</td>
</tr>
<tr>
<td>Maximum total number of bitmaps, cards,</td>
<td>16,777,216</td>
</tr>
<tr>
<td>and backgrounds per stack</td>
<td></td>
</tr>
<tr>
<td>Maximum stack name size</td>
<td>31 characters</td>
</tr>
<tr>
<td>Maximum stack script size</td>
<td>30,000 characters</td>
</tr>
<tr>
<td><strong>Background limits</strong></td>
<td></td>
</tr>
<tr>
<td>Background size (bytes)</td>
<td>LongInt$^*$</td>
</tr>
<tr>
<td>Minimum background size</td>
<td>64 bytes</td>
</tr>
<tr>
<td>Maximum parts per background</td>
<td>Integer</td>
</tr>
</tbody>
</table>

*continued*
## Appendix G

HyperCard Limits

### Table G-1  HyperCard limits (continued)

<table>
<thead>
<tr>
<th>Item</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Background limits (continued)</strong></td>
<td></td>
</tr>
<tr>
<td>Maximum total part size per background (bytes)</td>
<td>LongInt</td>
</tr>
<tr>
<td>Maximum background name size</td>
<td>31 characters</td>
</tr>
<tr>
<td>Maximum background script size</td>
<td>30,000 characters</td>
</tr>
<tr>
<td><strong>Card limits</strong></td>
<td></td>
</tr>
<tr>
<td>Card size (bytes)</td>
<td>LongInt*</td>
</tr>
<tr>
<td>Minimum card size</td>
<td>64 bytes</td>
</tr>
<tr>
<td>Maximum parts per card</td>
<td>Integer</td>
</tr>
<tr>
<td>Maximum total part size per card (bytes)</td>
<td>LongInt</td>
</tr>
<tr>
<td>Maximum total text size per card (bytes)</td>
<td>LongInt</td>
</tr>
<tr>
<td>Maximum card name size</td>
<td>31 characters</td>
</tr>
<tr>
<td>Maximum card script size</td>
<td>30,000 characters</td>
</tr>
<tr>
<td><strong>Part (button or field) limits</strong></td>
<td></td>
</tr>
<tr>
<td>Part size (bytes)</td>
<td>Integer†</td>
</tr>
<tr>
<td>Minimum overhead per part</td>
<td>30 bytes</td>
</tr>
<tr>
<td>Maximum part name size</td>
<td>31 characters</td>
</tr>
<tr>
<td>Maximum part text size</td>
<td>30,000 characters</td>
</tr>
<tr>
<td>Maximum part script size</td>
<td>30,000 characters</td>
</tr>
<tr>
<td><strong>HyperTalk limits</strong></td>
<td></td>
</tr>
<tr>
<td>Maximum nested repeat structures</td>
<td>30</td>
</tr>
<tr>
<td>Maximum active variables (all pending handlers)</td>
<td>512</td>
</tr>
<tr>
<td>Maximum size card name with go command</td>
<td>31 characters</td>
</tr>
<tr>
<td>Maximum variable name size</td>
<td>31 characters</td>
</tr>
<tr>
<td>Maximum number format size</td>
<td>31 characters</td>
</tr>
</tbody>
</table>

*continued*
HyperCard Limits

Table G-1  HyperCard limits (continued)

<table>
<thead>
<tr>
<th>Item</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>HyperTalk limits (continued)</strong></td>
<td></td>
</tr>
<tr>
<td>Maximum size of command with arguments</td>
<td>254 characters</td>
</tr>
<tr>
<td>Maximum handler name size</td>
<td>254 characters</td>
</tr>
<tr>
<td>Maximum script size</td>
<td>30,000 characters</td>
</tr>
<tr>
<td>Maximum variable value size</td>
<td>Limited by available memory</td>
</tr>
</tbody>
</table>

* Limited by HyperCard stack size; less than 100 KB for practical use.
† The sum of the other elements in the button or field must be less than the part size.
APPENDIX H

HyperCard Syntax Summary

This appendix lists HyperTalk's built-in commands (Table H-1) and functions (Table H-2), showing the syntax of their parameters.

HyperTalk's built-in commands and functions are described in more detail in Chapters 10 and 11, respectively. A brief description for each is included in Appendix I.

Syntax Description Notation

The syntax descriptions use the following typographic conventions. Words or phrases in this type are HyperTalk language elements that you type to the computer literally, exactly as shown. Words in italic type are metasymbols (used to describe general elements), not specific names—you must substitute the actual instances. Brackets ([]) enclose optional elements that may be included if you need them. (Don’t type the brackets.) In some cases, optional elements change what the message does; in other cases they are helper words that have no effect except to make the message more readable.

It doesn’t matter whether you use uppercase or lowercase letters; names that are formed from two words are shown in lowercase letters with a capital in the middle (likeThis) merely to make them more readable. The HyperTalk prepositions of and in are interchangeable—the syntax descriptions use the one that sounds more natural.

The terms factor and expression are defined in Chapter 7. Briefly, a factor can be a constant, literal, function, property, number, or container, and an expression can be a factor or a complex expression built with factors and operators. Also, a factor can be an expression within parentheses.
APPENDIX H

HyperCard Syntax Summary

Table H-1 HyperTalk command syntax

<table>
<thead>
<tr>
<th>Command</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>add number to [chunk of] container</td>
<td>answer file [promptText] [of type fileType]</td>
</tr>
<tr>
<td>answer program [promptText] [of type processType]</td>
<td>answer question with reply</td>
</tr>
<tr>
<td>answer question with reply1 or reply2</td>
<td>answer question with reply1 or reply2 or reply3</td>
</tr>
<tr>
<td>arrowKey direction</td>
<td>ask file promptText [with [default] fileName]</td>
</tr>
<tr>
<td>ask password [clear] question [with defaultAnswer]</td>
<td>ask question [with defaultAnswer]</td>
</tr>
<tr>
<td>beep [number]</td>
<td>choose toolName tool</td>
</tr>
<tr>
<td>choose tool toolNumber</td>
<td>click at point [with key]</td>
</tr>
<tr>
<td>click at point with key, key2</td>
<td>click at point with key, key2, key3</td>
</tr>
<tr>
<td>close [docPathname [with</td>
<td>in]] appPathname</td>
</tr>
<tr>
<td>close printing</td>
<td>close window windowName</td>
</tr>
<tr>
<td>commandKeyDown char</td>
<td>controlkey keyNumber</td>
</tr>
<tr>
<td>convert [chunk of] container\literal [from format [and format]] to format [and format]</td>
<td>create menu menuName</td>
</tr>
</tbody>
</table>

continued
### Table H-1  HyperTalk command syntax (continued)

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>create stack stackName [with background] [in a new window]</code></td>
<td>Create a stack with an optional background and window.</td>
</tr>
<tr>
<td><code>debug checkpoint</code></td>
<td>Debug a checkpoint.</td>
</tr>
<tr>
<td><code>delete chunk [of container]</code></td>
<td>Delete a chunk from a container.</td>
</tr>
<tr>
<td><code>delete menu</code></td>
<td>Delete a menu.</td>
</tr>
<tr>
<td><code>delete menuItem of menu</code></td>
<td>Delete a menu item from a menu.</td>
</tr>
<tr>
<td><code>delete partName</code></td>
<td>Delete a part.</td>
</tr>
<tr>
<td><code>dial number</code></td>
<td>Make a phone call.</td>
</tr>
<tr>
<td><code>dial number with [modem [modemCommands]]</code></td>
<td>Dial a number with modem commands.</td>
</tr>
<tr>
<td>`disable [card</td>
<td>background] button`</td>
</tr>
<tr>
<td><code>disable menu</code></td>
<td>Disable a menu.</td>
</tr>
<tr>
<td><code>disable menuItem of menu</code></td>
<td>Disable a menu item.</td>
</tr>
<tr>
<td><code>divide [chunk of] container by number</code></td>
<td>Divide a container by a number.</td>
</tr>
<tr>
<td><code>do expression [as scriptLanguage]</code></td>
<td>Execute an expression in a specified script language.</td>
</tr>
<tr>
<td><code>doMenu itemName [,menuName][without dialog] [with modifierKey [,modifierKey]]</code></td>
<td>Execute a menu item with or without dialog and modifier keys.</td>
</tr>
<tr>
<td><code>drag from point1 to point2</code></td>
<td>Drag from one point to another.</td>
</tr>
<tr>
<td><code>drag from point1 to point2 with key</code></td>
<td>Drag with a specified key.</td>
</tr>
<tr>
<td><code>drag from point1 to point2 with key, key2</code></td>
<td>Drag with two specified keys.</td>
</tr>
<tr>
<td><code>drag from point1 to point2 with key, key2, key3</code></td>
<td>Drag with three specified keys.</td>
</tr>
<tr>
<td><code>edit script of object</code></td>
<td>Edit the script of an object.</td>
</tr>
<tr>
<td><code>enable button</code></td>
<td>Enable a button.</td>
</tr>
<tr>
<td><code>enable menu</code></td>
<td>Enable a menu.</td>
</tr>
<tr>
<td><code>enable menuItem of menu</code></td>
<td>Enable a menu item.</td>
</tr>
<tr>
<td><code>enterInField</code></td>
<td>Enter text into a field.</td>
</tr>
<tr>
<td><code>enterKey</code></td>
<td>Enter a key.</td>
</tr>
<tr>
<td><code>export paint to file fileName</code></td>
<td>Export paint to a file.</td>
</tr>
</tbody>
</table>

*continued*
HyperCard Syntax Summary

Table H-1  HyperTalk command syntax  (continued)

find chars [international] text [in field] [of marked cards]
find [international] text [in field] [of marked cards]
find string [international] text [in field] [of marked cards]
find whole [international] text [in field] [of marked cards]
find word [international] text [in field] [of marked cards]
functionKey keyNumber
get expression
get [the] property [of object]
go back
go forth
go [to] background [of [stack] stackName] [in a new window] [without dialog]

continued
APPENDIX H

HyperCard Syntax Summary

Table H-1  HyperTalk command syntax  (continued)

<table>
<thead>
<tr>
<th>Command</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>hide window stackName</td>
<td></td>
</tr>
<tr>
<td>hide window windowName</td>
<td></td>
</tr>
<tr>
<td>import paint from file fileName</td>
<td></td>
</tr>
<tr>
<td>keyDown char</td>
<td></td>
</tr>
<tr>
<td>lock error dialogs</td>
<td></td>
</tr>
<tr>
<td>lock messages</td>
<td></td>
</tr>
<tr>
<td>lock recent</td>
<td></td>
</tr>
<tr>
<td>lock screen</td>
<td></td>
</tr>
<tr>
<td>mark all cards</td>
<td></td>
</tr>
<tr>
<td>mark card</td>
<td></td>
</tr>
<tr>
<td>mark cards by finding chars text [in field]</td>
<td></td>
</tr>
<tr>
<td>mark cards by finding string text [in field]</td>
<td></td>
</tr>
<tr>
<td>mark cards by finding text [in field]</td>
<td></td>
</tr>
<tr>
<td>mark cards by finding whole text [in field]</td>
<td></td>
</tr>
<tr>
<td>mark cards by finding word text [in field]</td>
<td></td>
</tr>
<tr>
<td>mark cards where expression</td>
<td></td>
</tr>
<tr>
<td>multiply [chunk of] container by number</td>
<td></td>
</tr>
<tr>
<td>open file fileName</td>
<td></td>
</tr>
<tr>
<td>open [fileName with] application</td>
<td></td>
</tr>
<tr>
<td>open printing [with dialog]</td>
<td></td>
</tr>
<tr>
<td>open report printing [with dialog]</td>
<td></td>
</tr>
<tr>
<td>open report printing [with template className]</td>
<td></td>
</tr>
<tr>
<td>palette paletteName[, point]</td>
<td></td>
</tr>
<tr>
<td>picture [fileName, fileType, windowStyle, visible, depth, floatingLayer]</td>
<td></td>
</tr>
<tr>
<td>play sound [tempo tempo] [notes]</td>
<td></td>
</tr>
</tbody>
</table>

continued
APPENDIX H

HyperCard Syntax Summary

Table H-1 HyperTalk command syntax (continued)

<table>
<thead>
<tr>
<th>Command</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>play</td>
<td>stop</td>
</tr>
<tr>
<td>pop card</td>
<td>preposition [chunk of] container</td>
</tr>
<tr>
<td>print</td>
<td>button</td>
</tr>
<tr>
<td>print card</td>
<td>from point1 to point2</td>
</tr>
<tr>
<td>print</td>
<td>expression</td>
</tr>
<tr>
<td>print</td>
<td>field</td>
</tr>
<tr>
<td>print</td>
<td>fileName with application</td>
</tr>
<tr>
<td>print</td>
<td>marked cards</td>
</tr>
<tr>
<td>print</td>
<td>number cards</td>
</tr>
<tr>
<td>push background</td>
<td>of stack stackName</td>
</tr>
<tr>
<td>push</td>
<td>card</td>
</tr>
<tr>
<td>push</td>
<td>card of stack stackName</td>
</tr>
<tr>
<td>push</td>
<td>stack</td>
</tr>
<tr>
<td>put expression</td>
<td>preposition [chunk of] container</td>
</tr>
<tr>
<td>put itemName</td>
<td>preposition [menuItem of] menu [with menuMsg message]</td>
</tr>
<tr>
<td>read from file</td>
<td>fileName at start for numberOfChars</td>
</tr>
<tr>
<td>read from file</td>
<td>fileName for numberOfChars</td>
</tr>
<tr>
<td>read from file</td>
<td>fileName until char</td>
</tr>
<tr>
<td>read from file</td>
<td>fileName until constant</td>
</tr>
<tr>
<td>reply expression</td>
<td>with keyword aeKeyword</td>
</tr>
<tr>
<td>reply error expression</td>
<td></td>
</tr>
<tr>
<td>request appleEvent data</td>
<td>class</td>
</tr>
<tr>
<td>request appleEvent data with keyword aeKeyword</td>
<td></td>
</tr>
<tr>
<td>request expression</td>
<td>from program</td>
</tr>
<tr>
<td>request expression</td>
<td>of</td>
</tr>
</tbody>
</table>

continued
<table>
<thead>
<tr>
<th>Table H-1</th>
<th>HyperTalk command syntax (continued)</th>
</tr>
</thead>
<tbody>
<tr>
<td>request expression of from this program</td>
<td></td>
</tr>
<tr>
<td>reset menubar</td>
<td></td>
</tr>
<tr>
<td>reset paint</td>
<td></td>
</tr>
<tr>
<td>reset printing</td>
<td></td>
</tr>
<tr>
<td>returnInField</td>
<td></td>
</tr>
<tr>
<td>returnKey</td>
<td></td>
</tr>
<tr>
<td>save stack stackName as fileName</td>
<td></td>
</tr>
<tr>
<td>save [this] stack as [stack] fileName</td>
<td></td>
</tr>
<tr>
<td>select empty</td>
<td></td>
</tr>
<tr>
<td>select object</td>
<td></td>
</tr>
<tr>
<td>select [preposition] chunk of field</td>
<td></td>
</tr>
<tr>
<td>select [preposition] text of field</td>
<td></td>
</tr>
<tr>
<td>selectedButton(family)</td>
<td></td>
</tr>
<tr>
<td>the selectedButton of family</td>
<td></td>
</tr>
<tr>
<td>set [the] property [of element] to value</td>
<td></td>
</tr>
<tr>
<td>show all cards</td>
<td></td>
</tr>
<tr>
<td>show background picture</td>
<td></td>
</tr>
<tr>
<td>show card picture</td>
<td></td>
</tr>
<tr>
<td>show cards</td>
<td></td>
</tr>
<tr>
<td>show groups</td>
<td></td>
</tr>
<tr>
<td>show marked cards</td>
<td></td>
</tr>
<tr>
<td>show menuBar</td>
<td></td>
</tr>
<tr>
<td>show [number] cards</td>
<td></td>
</tr>
<tr>
<td>show object [at point]</td>
<td></td>
</tr>
<tr>
<td>show picture of background</td>
<td></td>
</tr>
</tbody>
</table>

continued
show picture of card
show titlebar
show window stackName
show window windowName [at point]
sort [lines|items of] container [sortDirection] ¬ [sortType] [by sortKey]
sort [[[marked] cards of] background][sortDirection] ¬ [sortType] [by sortKey]
sort [[[marked] cards of [this]]stack [sortDirection] ¬ [sortType] [by sortKey]
start using stack stackName
stop using stack stackName
subtract number from [chunk of] container
tabKey
type text
type text with key
type text with key, key2
type text with key, key2, key3
unlock error dialogs
unlock messages
unlock recent
unlock screen [with effectName]
unmark all cards
unmark card
unmark cards by finding chars text in field
unmark cards by finding string text in field

continued
APPENDIX H

HyperCard Syntax Summary

Table H-1  HyperTalk command syntax (continued)

- unmark cards by finding text in field
- unmark cards by finding whole text in field
- unmark cards by finding word text in field
- unmark cards where expression
- visual [effect] effectName [speed] [to image]
- wait [for] time [seconds]
- wait until condition
- wait while condition
- write text to file fileName
- write text to file fileName at end
- write text to file fileName at eof
- write text to file fileName at start

Table H-2  HyperTalk function syntax

- the abs of factor
  abs(expression)
- annuity (rate, periods)
- the atan of factor
  atan(expression)
- average (list)
- average function
- the charToNum of factor
  charToNum (expression)
- the clickChunk

continued
## HyperCard Syntax Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>clickChunk()</td>
<td>the clickH</td>
</tr>
<tr>
<td>clickH()</td>
<td>the clickLine</td>
</tr>
<tr>
<td>clickLine()</td>
<td>the clickLoc</td>
</tr>
<tr>
<td>clickLoc()</td>
<td>the clickText</td>
</tr>
<tr>
<td>clickText()</td>
<td>the clickV</td>
</tr>
<tr>
<td>clickV()</td>
<td>the commandKey</td>
</tr>
<tr>
<td>commandKey()</td>
<td>compound(rate, periods)</td>
</tr>
<tr>
<td>cos(factor)</td>
<td>the cos of factor</td>
</tr>
<tr>
<td>cos(expression)</td>
<td>the [adjective] date</td>
</tr>
<tr>
<td>date()</td>
<td>the destination</td>
</tr>
<tr>
<td>destination()</td>
<td>the diskSpace</td>
</tr>
<tr>
<td>diskSpace()</td>
<td>the exp of factor</td>
</tr>
<tr>
<td>exp(expression)</td>
<td>the expl of factor</td>
</tr>
</tbody>
</table>

*continued*
### HyperCard Syntax Summary

**Table H-2**  
HyperTalk function syntax (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>exp1(expression)</code></td>
<td>the exp2 of factor</td>
</tr>
<tr>
<td><code>exp2(expression)</code></td>
<td>the foundChunk</td>
</tr>
<tr>
<td><code>foundChunk()</code></td>
<td>the foundField</td>
</tr>
<tr>
<td><code>foundField()</code></td>
<td>the foundLine</td>
</tr>
<tr>
<td><code>foundLine()</code></td>
<td>the foundText</td>
</tr>
<tr>
<td><code>foundText()</code></td>
<td>the heapSpace</td>
</tr>
<tr>
<td><code>length(expression)</code></td>
<td>the length of factor</td>
</tr>
<tr>
<td><code>ln(expression)</code></td>
<td>the ln of factor</td>
</tr>
<tr>
<td><code>ln1(expression)</code></td>
<td>the ln1 of factor</td>
</tr>
<tr>
<td><code>log2(expression)</code></td>
<td>the log2 of factor</td>
</tr>
<tr>
<td><code>max(list)</code></td>
<td>the menus</td>
</tr>
<tr>
<td><code>menus()</code></td>
<td>min(list)</td>
</tr>
<tr>
<td></td>
<td>the mouse</td>
</tr>
</tbody>
</table>

*continued*
Mouse-related functions:

- mouse()
- the mouseClick
- mouseClicked()
- the mouseH
- mouseH()
- the mouseLoc
- mouseLoc()
- the mouseV
- mouseV()

Other functions:

- [the] number of objects
- [the] number of chunks in expression
- [the] number of backgrounds [in this stack]
- [the] number of cards in background
- [the] number of cards [in this stack]
- [the] number of marked cards
- [the] number of menus
- [the] number of menuItems of menu
- [the] number of [card|background] parts
- [the] number of windows
- the numToChar of factor
- numToChar(expression)
- offset(string1, string2)
- the optionKey
- optionKey()
- the param of factor

Table H-2: HyperTalk function syntax (continued)
APPENDIX H

HyperCard Syntax Summary

Table H-2   HyperTalk function syntax (continued)

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>param(expression)</td>
</tr>
<tr>
<td>the paramCount</td>
</tr>
<tr>
<td>paramCount()</td>
</tr>
<tr>
<td>the params</td>
</tr>
<tr>
<td>params()</td>
</tr>
<tr>
<td>the programs</td>
</tr>
<tr>
<td>programs()</td>
</tr>
<tr>
<td>the random of factor</td>
</tr>
<tr>
<td>random(expression)</td>
</tr>
<tr>
<td>the result</td>
</tr>
<tr>
<td>result()</td>
</tr>
<tr>
<td>the round of factor</td>
</tr>
<tr>
<td>round(expression)</td>
</tr>
<tr>
<td>the screenRect</td>
</tr>
<tr>
<td>screenRect()</td>
</tr>
<tr>
<td>the seconds</td>
</tr>
<tr>
<td>seconds()</td>
</tr>
<tr>
<td>the selectedButton of family</td>
</tr>
<tr>
<td>selectedButton (family)</td>
</tr>
<tr>
<td>the selectedChunk</td>
</tr>
<tr>
<td>selectedChunk()</td>
</tr>
<tr>
<td>the selectedField</td>
</tr>
<tr>
<td>selectedField()</td>
</tr>
<tr>
<td>the selectedLine [of button</td>
</tr>
<tr>
<td>selectedLine([button</td>
</tr>
</tbody>
</table>

continued
## Appendix H

HyperCard Syntax Summary

### Table H-2  HyperTalk function syntax (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>the selectedLoc</code></td>
<td><code>selectedLoc()</code></td>
</tr>
<tr>
<td>`the selectedText [of button</td>
<td>`selectedText([button</td>
</tr>
<tr>
<td><code>the shiftKey</code></td>
<td><code>shiftKey()</code></td>
</tr>
<tr>
<td><code>the sin of factor</code></td>
<td><code>sin(expression)</code></td>
</tr>
<tr>
<td><code>the sound</code></td>
<td><code>sound()</code></td>
</tr>
<tr>
<td><code>the sqrt of factor</code></td>
<td><code>sqrt(expression)</code></td>
</tr>
<tr>
<td><code>the stacks</code></td>
<td><code>stacks()</code></td>
</tr>
<tr>
<td><code>the stackSpace</code></td>
<td><code>stackSpace()</code></td>
</tr>
<tr>
<td><code>sum(list)</code></td>
<td><code>sum(list)</code></td>
</tr>
<tr>
<td><code>the systemVersion</code></td>
<td><code>systemVersion()</code></td>
</tr>
<tr>
<td><code>the tan of factor</code></td>
<td><code>tan(expression)</code></td>
</tr>
<tr>
<td><code>the target</code></td>
<td><code>the target</code></td>
</tr>
<tr>
<td><code>the ticks</code></td>
<td><code>ticks()</code></td>
</tr>
<tr>
<td><code>the [adjective] time</code></td>
<td></td>
</tr>
</tbody>
</table>

*continued*
APPENDIX H

HyperCard Syntax Summary

Table H-2  HyperTalk function syntax (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>time()</td>
<td>the tool</td>
</tr>
<tr>
<td>tool()</td>
<td>the trunc of factor</td>
</tr>
<tr>
<td>trunc(expression)</td>
<td>the value of factor</td>
</tr>
<tr>
<td>value(expression)</td>
<td>the windows</td>
</tr>
<tr>
<td>windows()</td>
<td></td>
</tr>
</tbody>
</table>
Table I-1 lists and defines, in alphabetical order, HyperTalk’s native vocabulary—the names of its built-in commands and functions, its system messages, keywords, the names of objects and their properties, and various adjectives, constants, ordinals, and other terms.

This list is not exhaustive—there are other terms with specific meanings recognized by HyperCard in particular contexts, and they are described with the primary term to which they relate. For example, the names of the various visual effects are listed with the `visual` command in Chapter 10.

The parameter syntax of HyperTalk’s built-in commands and functions is shown in Appendix H.

<table>
<thead>
<tr>
<th>Term</th>
<th>Category</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>abbr[eviated]</td>
<td>Adjective</td>
<td>Modifies the value returned by the date function or the name or ID property.</td>
</tr>
<tr>
<td>abs</td>
<td>Function</td>
<td>Returns the absolute value of a number.</td>
</tr>
<tr>
<td>add</td>
<td>Command</td>
<td>Adds the value of an expression to a value in a container.</td>
</tr>
<tr>
<td>address</td>
<td>Property</td>
<td>Returns the path of the currently executing HyperCard program.</td>
</tr>
<tr>
<td>after</td>
<td>Preposition</td>
<td>Used with <code>put</code> command, directing HyperCard to append a new value following any preexisting value in a container.</td>
</tr>
<tr>
<td>all</td>
<td>Adjective</td>
<td>Specifies total number of cards in stack to <code>show cards</code> command.</td>
</tr>
<tr>
<td>annuity</td>
<td>Function</td>
<td>Computes present or future value of an ordinary annuity.</td>
</tr>
</tbody>
</table>

continued
# HyperTalk Vocabulary

<table>
<thead>
<tr>
<th>Term</th>
<th>Category</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>answer</td>
<td>Command</td>
<td>Displays a dialog box with question and reply buttons.</td>
</tr>
<tr>
<td>answer file</td>
<td>Command</td>
<td>Presents the standard dialog box for locating a file; used for opening files of a specified type.</td>
</tr>
<tr>
<td>answer program</td>
<td>Command</td>
<td>Presents the standard dialog box for locating a program to link to.</td>
</tr>
<tr>
<td>any</td>
<td>Ordinal</td>
<td>Special ordinal used with object or chunk to specify a random element within its enclosing set.</td>
</tr>
<tr>
<td>appleEvent</td>
<td>System message</td>
<td>Sent to the current card when an Apple event is received.</td>
</tr>
<tr>
<td>arrowKey</td>
<td>Command</td>
<td>Takes you to another card.</td>
</tr>
<tr>
<td>arrowKey</td>
<td>System message</td>
<td>Sent to current card when an arrow key is pressed.</td>
</tr>
<tr>
<td>ask</td>
<td>Command</td>
<td>Displays a dialog box with a question and default answer.</td>
</tr>
<tr>
<td>ask file</td>
<td>Command</td>
<td>Presents the standard dialog box for locating where to save a file; used for saving files.</td>
</tr>
<tr>
<td>ask password</td>
<td>Command</td>
<td>Displays a dialog box with a field for a password.</td>
</tr>
<tr>
<td>atan</td>
<td>Function</td>
<td>Returns trigonometric arc tangent of a number.</td>
</tr>
<tr>
<td>autoHilite</td>
<td>Property</td>
<td>Determines whether or not a button’s hilite property is affected by the message mouseDown. Also determines whether or not a field behaves as a list.</td>
</tr>
<tr>
<td>autoTab</td>
<td>Property</td>
<td>Determines whether the specified nonscrolling field sends the tabKey message to the current card.</td>
</tr>
</tbody>
</table>

Table I-1 HyperTalk vocabulary (continued)
### HyperTalk Vocabulary

**Table I-1** HyperTalk vocabulary (continued)

<table>
<thead>
<tr>
<th>Term</th>
<th>Category</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>average</td>
<td>Function</td>
<td>Returns the average value of numbers in a list.</td>
</tr>
<tr>
<td>background</td>
<td>Object</td>
<td>Generic name of background object; used with specific designation (go to next background). Also used to specify containing object for buttons and, optionally, fields (background button 2).</td>
</tr>
<tr>
<td>backgrounds</td>
<td>Object type</td>
<td>Specifies backgrounds as type of object to the number function.</td>
</tr>
<tr>
<td>beep</td>
<td>Command</td>
<td>Causes Macintosh to make a beep sound.</td>
</tr>
<tr>
<td>before</td>
<td>Preposition</td>
<td>Used with put command, directing HyperCard to place a new value at the beginning of any preexisting value in a container.</td>
</tr>
<tr>
<td>bg</td>
<td>Object</td>
<td>Abbreviation for background.</td>
</tr>
<tr>
<td>bkgnd</td>
<td>Object</td>
<td>Abbreviation for background.</td>
</tr>
<tr>
<td>bkgnds</td>
<td>Object type</td>
<td>Specifies backgrounds as type of object to the number function.</td>
</tr>
<tr>
<td>blindTyping</td>
<td>Property</td>
<td>Allows typing into Message box when hidden.</td>
</tr>
<tr>
<td>botRight</td>
<td>Property</td>
<td>Abbreviation for bottomRight.</td>
</tr>
<tr>
<td>bottom</td>
<td>Property</td>
<td>Determines or changes the value of item 4 of the rectangle property when applied to the specified object or window.</td>
</tr>
<tr>
<td>bottomRight</td>
<td>Property</td>
<td>Determines or changes items 3 and 4 of the value of the rectangle property when applied to the specified object or window.</td>
</tr>
<tr>
<td>browse</td>
<td>Tool</td>
<td>Name of tool from Tools palette; used with choose command or returned by the tool function.</td>
</tr>
</tbody>
</table>

*continued*
HyperTalk Vocabulary

**Table I-1** HyperTalk vocabulary (continued)

<table>
<thead>
<tr>
<th>Term</th>
<th>Category</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>brush</td>
<td>Property</td>
<td>Determines the current brush shape.</td>
</tr>
<tr>
<td>brush</td>
<td>Tool</td>
<td>Name of tool from Tools palette; used with choose command or returned by the tool function.</td>
</tr>
<tr>
<td>btn</td>
<td>Object</td>
<td>Abbreviation for button.</td>
</tr>
<tr>
<td>bucket</td>
<td>Tool</td>
<td>Name of tool from Tools palette; used with choose command or returned by the tool function.</td>
</tr>
<tr>
<td>button</td>
<td>Object</td>
<td>Generic name of button object; used with a specific designation (hide button one).</td>
</tr>
<tr>
<td>button</td>
<td>Tool</td>
<td>Name of tool from Tools palette; used with choose command or returned by the tool function.</td>
</tr>
<tr>
<td>buttonCount</td>
<td>Property</td>
<td>Determines the number of buttons in an open palette XCMD displayed by the palette command.</td>
</tr>
<tr>
<td>buttons</td>
<td>Object type</td>
<td>Specifies buttons as type of object to the number function.</td>
</tr>
<tr>
<td>cantAbort</td>
<td>Property</td>
<td>Determines if a script can be stopped by pressing Command-period.</td>
</tr>
<tr>
<td>cantDelete</td>
<td>Property</td>
<td>Determines if a background, card, or stack can be deleted.</td>
</tr>
<tr>
<td>cantModify</td>
<td>Property</td>
<td>Determines if a stack can be modified. Can be used with a password to prevent anyone without the password from modifying a stack.</td>
</tr>
<tr>
<td>cantPeek</td>
<td>Property</td>
<td>Determines if the outline is shown around buttons and fields when the Command-Option or Command-Option-Shift keys are pressed.</td>
</tr>
</tbody>
</table>

*continued*
### HyperTalk Vocabulary (continued)

<table>
<thead>
<tr>
<th>Term</th>
<th>Category</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>card</td>
<td>Object</td>
<td>Generic name of a card object; used with a specific designation (go to card &quot;fred&quot;). Also used to specify containing object for fields and, optionally, buttons (card field &quot;date&quot;).</td>
</tr>
<tr>
<td>cards</td>
<td>Object type</td>
<td>Specifies cards as type of object to the number function or show command.</td>
</tr>
<tr>
<td>cd</td>
<td>Object</td>
<td>Abbreviation for card.</td>
</tr>
<tr>
<td>center</td>
<td>Adjective</td>
<td>Specifies center alignment of text in a field.</td>
</tr>
<tr>
<td>centered</td>
<td>Property</td>
<td>Determines whether shapes are drawn from the center or from the corner.</td>
</tr>
<tr>
<td>char[acter]</td>
<td>Chunk</td>
<td>A character of text in any container or expression.</td>
</tr>
<tr>
<td>char[acter]s</td>
<td>Chunk type</td>
<td>Specifies characters as type of chunk to the number function.</td>
</tr>
<tr>
<td>charToNum</td>
<td>Function</td>
<td>Returns ASCII value of a character.</td>
</tr>
<tr>
<td>checkMark</td>
<td>Property</td>
<td>Determines check character for a menu item.</td>
</tr>
<tr>
<td>choose</td>
<td>Command</td>
<td>Changes the current tool.</td>
</tr>
<tr>
<td>click</td>
<td>Command</td>
<td>Causes same actions as clicking at a specified location.</td>
</tr>
<tr>
<td>clickChunk</td>
<td>Function</td>
<td>Returns chunk information about text that is clicked.</td>
</tr>
<tr>
<td>clickH</td>
<td>Function</td>
<td>Returns horizontal position of last mouse click.</td>
</tr>
<tr>
<td>clickLine</td>
<td>Function</td>
<td>Returns line information about text that is clicked.</td>
</tr>
<tr>
<td>clickLoc</td>
<td>Function</td>
<td>Returns location of most recent click.</td>
</tr>
<tr>
<td>clickText</td>
<td>Function</td>
<td>Returns text information about word or group phrase that is clicked.</td>
</tr>
</tbody>
</table>

*continued*
## Appendix I

HyperTalk Vocabulary

<table>
<thead>
<tr>
<th>Term</th>
<th>Category</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>close</td>
<td>Command</td>
<td>Closes an application, document opened by another application, or desk accessory.</td>
</tr>
<tr>
<td>clickV</td>
<td>Function</td>
<td>Returns vertical position of last mouse click.</td>
</tr>
<tr>
<td>close</td>
<td>System message</td>
<td>Sent to the current card when you close a stack window with the <code>close window</code> command or by clicking the close box.</td>
</tr>
<tr>
<td>closeBackground</td>
<td>System message</td>
<td>Sent to current card just before you leave the current background.</td>
</tr>
<tr>
<td>closeCard</td>
<td>System message</td>
<td>Sent to current card just before you leave it.</td>
</tr>
<tr>
<td>closeField</td>
<td>System message</td>
<td>Sent to unlocked field when it closes.</td>
</tr>
<tr>
<td>close file</td>
<td>Command</td>
<td>Closes a previously opened disk file.</td>
</tr>
<tr>
<td>closePalette</td>
<td>System message</td>
<td>Sent to the current card when a palette that was opened with the palette command is closed.</td>
</tr>
<tr>
<td>closePicture</td>
<td>System message</td>
<td>Sent to the current card when a window that was created with the picture command is closed.</td>
</tr>
<tr>
<td>close printing</td>
<td>Command</td>
<td>Ends a print job.</td>
</tr>
<tr>
<td>closeStack</td>
<td>System message</td>
<td>Sent to current card just before you leave the current stack.</td>
</tr>
<tr>
<td>close window</td>
<td>Command</td>
<td>Closes a stack or picture window.</td>
</tr>
<tr>
<td>commandChar</td>
<td>Property</td>
<td>Determines the character to use with the Command key to invoke a menu item.</td>
</tr>
<tr>
<td>commandKey</td>
<td>Function</td>
<td>Returns the state of the Command key: up or down.</td>
</tr>
<tr>
<td>commandKeyDown</td>
<td>Command</td>
<td>Causes a built-in HyperCard response, depending on key pressed with Command key.</td>
</tr>
<tr>
<td>commandKeyDown</td>
<td>System message</td>
<td>Sent to current card when a combination of the Command key and another key is pressed.</td>
</tr>
</tbody>
</table>

**Table I-1**  HyperTalk vocabulary (continued)
### HyperTalk Vocabulary (continued)

<table>
<thead>
<tr>
<th>Term</th>
<th>Category</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>commands</td>
<td>Property</td>
<td>Returns a list of the commands associated with the buttons in an open palette displayed by the <code>palette</code> command.</td>
</tr>
<tr>
<td>compound</td>
<td>Function</td>
<td>Computes present or future value of a compound interest-bearing account.</td>
</tr>
<tr>
<td>controlKey</td>
<td>Command</td>
<td>Sends the <code>controlKey</code> system message.</td>
</tr>
<tr>
<td>controlKey</td>
<td>System message</td>
<td>Sent to current card when a combination of the Control key and another key is pressed.</td>
</tr>
<tr>
<td>convert</td>
<td>Command</td>
<td>Converts a date or time to specified format.</td>
</tr>
<tr>
<td>create menu</td>
<td>Command</td>
<td>Creates a new menu with the specified name.</td>
</tr>
<tr>
<td>create stack</td>
<td>Command</td>
<td>Creates a new stack with the specified name and background.</td>
</tr>
<tr>
<td>cos</td>
<td>Function</td>
<td>Returns the cosine of the angle that is passed to it.</td>
</tr>
<tr>
<td>cursor</td>
<td>Property</td>
<td>Sets image appearing at pointer location on screen. You can only set <code>cursor</code>; you can’t get it.</td>
</tr>
<tr>
<td>curve</td>
<td>Tool</td>
<td>Name of tool from Tools palette; used with <code>choose</code> command or returned by the <code>tool</code> function.</td>
</tr>
<tr>
<td>date</td>
<td>Function</td>
<td>Returns a string representing the current date.</td>
</tr>
<tr>
<td>debug checkpoint</td>
<td>Command</td>
<td>Sets a checkpoint in a script to invoke the built-in debugger.</td>
</tr>
<tr>
<td>debugger</td>
<td>Property</td>
<td>Determines the debugger to use.</td>
</tr>
<tr>
<td>delete (menu)</td>
<td>Command</td>
<td>Deletes a menu.</td>
</tr>
<tr>
<td>delete (menu items)</td>
<td>Command</td>
<td>Deletes a menu item.</td>
</tr>
<tr>
<td>delete (object)</td>
<td>Command</td>
<td>Deletes a button or field.</td>
</tr>
</tbody>
</table>
## HyperTalk Vocabulary

<table>
<thead>
<tr>
<th>Term</th>
<th>Category</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete (part)</td>
<td>Command</td>
<td>Deletes a button or field.</td>
</tr>
<tr>
<td>delete (text)</td>
<td>Command</td>
<td>Removes a chunk of text from a container.</td>
</tr>
<tr>
<td>deleteBackground</td>
<td>System message</td>
<td>Sent to current card just before the background is deleted.</td>
</tr>
<tr>
<td>deleteButton</td>
<td>System message</td>
<td>Sent to a button just before it is deleted.</td>
</tr>
<tr>
<td>deleteCard</td>
<td>System message</td>
<td>Sent to current card just before it is deleted.</td>
</tr>
<tr>
<td>deleteField</td>
<td>System message</td>
<td>Sent to a field just before it is deleted.</td>
</tr>
<tr>
<td>deleteStack</td>
<td>System message</td>
<td>Sent to the current card just before a stack is deleted.</td>
</tr>
<tr>
<td>destination</td>
<td>Function</td>
<td>Returns the name of a stack to which HyperCard is going.</td>
</tr>
<tr>
<td>dial</td>
<td>Command</td>
<td>Generates touch-tone sounds through audio output or modem attached to serial port.</td>
</tr>
<tr>
<td>dialingTime</td>
<td>Property</td>
<td>Determines how long HyperCard waits before closing the serial connection to a modem after dialing.</td>
</tr>
<tr>
<td>dialingVolume</td>
<td>Property</td>
<td>Determines the volume of the touch tones generated through the speaker by the <code>dial</code> command.</td>
</tr>
<tr>
<td>disable</td>
<td>Command</td>
<td>Disables the specified menu, menu item, or button.</td>
</tr>
<tr>
<td>diskSpace</td>
<td>Function</td>
<td>Displays the amount of free space available on the disk containing the current stack.</td>
</tr>
<tr>
<td>dithering</td>
<td>Property</td>
<td>Determines whether or not the picture opened by the picture command is dithered.</td>
</tr>
<tr>
<td>divide</td>
<td>Command</td>
<td>Divides the value in a container by the value of an expression.</td>
</tr>
<tr>
<td>do</td>
<td>Keyword</td>
<td>Sends the value of an expression as a message to the current card.</td>
</tr>
</tbody>
</table>

*continued*
APPENDIX I

HyperTalk Vocabulary

Table I-1  HyperTalk vocabulary (continued)

<table>
<thead>
<tr>
<th>Term</th>
<th>Category</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>doMenu</td>
<td>Command</td>
<td>Performs a specified menu command.</td>
</tr>
<tr>
<td>doMenu</td>
<td>System message</td>
<td>Sent to current card when any menu item is chosen.</td>
</tr>
<tr>
<td>dontSearch</td>
<td>Property</td>
<td>Determines whether a card, background, or field can be searched by the find command.</td>
</tr>
<tr>
<td>dontWrap</td>
<td>Property</td>
<td>Determines whether the text in a field wraps onto the next line.</td>
</tr>
<tr>
<td>down</td>
<td>Constant</td>
<td>Value returned by various functions to describe the state of a key or the mouse button.</td>
</tr>
<tr>
<td>drag</td>
<td>Command</td>
<td>Performs same action as a manual drag.</td>
</tr>
<tr>
<td>dragSpeed</td>
<td>Property</td>
<td>Sets pixels-per-second speed at which pointer moves with drag command.</td>
</tr>
<tr>
<td>editBkgrnd</td>
<td>Property</td>
<td>Determines whether manipulation of buttons, fields, or paintings occurs on current card or background.</td>
</tr>
<tr>
<td>edit script</td>
<td>Command</td>
<td>Opens the script of a specified object.</td>
</tr>
<tr>
<td>eight</td>
<td>Constant</td>
<td>String representation of the numerical value 8.</td>
</tr>
<tr>
<td>eighth</td>
<td>Ordinal</td>
<td>Designates object or chunk number eight within its enclosing set.</td>
</tr>
<tr>
<td>else</td>
<td>Keyword</td>
<td>Optionally follows then clause in an if structure to introduce an alternative action clause.</td>
</tr>
<tr>
<td>empty</td>
<td>Constant</td>
<td>The null string; same as the literal &quot;&quot;.</td>
</tr>
<tr>
<td>enable</td>
<td>Command</td>
<td>Enables the specified button, menu, or menu item.</td>
</tr>
<tr>
<td>enabled</td>
<td>Property</td>
<td>Determines whether the specified button, menu, or menu item is enabled.</td>
</tr>
</tbody>
</table>

continued
### Table I-1  HyperTalk vocabulary (continued)

<table>
<thead>
<tr>
<th>Term</th>
<th>Category</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>end</td>
<td>Keyword</td>
<td>Marks the end of a message handler, function handler, repeat loop, or multiple-statement then or else clause of an if structure.</td>
</tr>
<tr>
<td>enterInField</td>
<td>Command</td>
<td>Closes a field that is open for text editing.</td>
</tr>
<tr>
<td>enterInField</td>
<td>System message</td>
<td>Sent to the field when the Enter key is pressed while there is an insertion point or selection in the field.</td>
</tr>
<tr>
<td>enterKey</td>
<td>Command</td>
<td>Sends contents of Message box to the current card.</td>
</tr>
<tr>
<td>enterKey</td>
<td>System message</td>
<td>Sent to the current card when the Enter key is pressed unless the text insertion point is in a field.</td>
</tr>
<tr>
<td>environment</td>
<td>Property</td>
<td>Determines whether HyperCard Player or a fully enabled development version of HyperCard is running.</td>
</tr>
<tr>
<td>eraser</td>
<td>Tool</td>
<td>Name of tool from Tools palette; used with choose command or returned by the tool function.</td>
</tr>
<tr>
<td>exit</td>
<td>Keyword</td>
<td>Immediately ends execution of a message handler, function handler, or repeat loop.</td>
</tr>
<tr>
<td>exitField</td>
<td>System message</td>
<td>Sent to a field when the pointer leaves the field's rectangle.</td>
</tr>
<tr>
<td>exp</td>
<td>Function</td>
<td>Returns the mathematical exponential of its argument.</td>
</tr>
<tr>
<td>exp1</td>
<td>Function</td>
<td>Returns one less than the mathematical exponential of its argument.</td>
</tr>
<tr>
<td>export paint</td>
<td>Command</td>
<td>Creates a Macintosh paint file with the image of the current card.</td>
</tr>
<tr>
<td>exp2</td>
<td>Function</td>
<td>Returns the value of 2 raised to the power specified by the argument.</td>
</tr>
</tbody>
</table>
## HyperTalk Vocabulary (continued)

<table>
<thead>
<tr>
<th>Term</th>
<th>Category</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>Constant</td>
<td>Boolean value resulting from evaluation of a comparative expression and returned from some functions.</td>
</tr>
<tr>
<td>family</td>
<td>Property</td>
<td>Groups a set of buttons to function in a coordinated manner.</td>
</tr>
<tr>
<td>field</td>
<td>Container</td>
<td>Generic name of field container; used with specific designation (put the time into card field &quot;time&quot;).</td>
</tr>
<tr>
<td>field</td>
<td>Object</td>
<td>Generic name of field object; used with specific designation (get name of first field).</td>
</tr>
<tr>
<td>field</td>
<td>Tool</td>
<td>Name of tool from Tools palette; used with choose command or returned by the tool function.</td>
</tr>
<tr>
<td>fields</td>
<td>Object type</td>
<td>Specifies fields as type of object to the number function.</td>
</tr>
<tr>
<td>fifth</td>
<td>Ordinal</td>
<td>Designates object or chunk number five within its enclosing set.</td>
</tr>
<tr>
<td>filled</td>
<td>Property</td>
<td>Determines the Draw Filled setting.</td>
</tr>
<tr>
<td>find</td>
<td>Command</td>
<td>Searches card and background fields for text strings derived from an expression.</td>
</tr>
<tr>
<td>first</td>
<td>Ordinal</td>
<td>Designates object or chunk number one within its enclosing set.</td>
</tr>
<tr>
<td>five</td>
<td>Constant</td>
<td>String representation of the numerical value 5.</td>
</tr>
<tr>
<td>fixedLineHeight</td>
<td>Property</td>
<td>Determine whether or not a field has fixed line spacing.</td>
</tr>
<tr>
<td>formFeed</td>
<td>Constant</td>
<td>The form feed character (ASCII 12), which starts a new page in some file formats.</td>
</tr>
<tr>
<td>foundChunk</td>
<td>Function</td>
<td>Returns a chunk expression describing the text found with the find command.</td>
</tr>
</tbody>
</table>

*continued*
### APPENDIX I

HyperTalk Vocabulary

<table>
<thead>
<tr>
<th>Term</th>
<th>Category</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>foundField</td>
<td>Function</td>
<td>Returns an expression describing the field the text was found in with the find command.</td>
</tr>
<tr>
<td>foundLine</td>
<td>Function</td>
<td>Returns an expression describing the line the text was found in with the find command.</td>
</tr>
<tr>
<td>foundText</td>
<td>Function</td>
<td>Returns the text found with the find command.</td>
</tr>
<tr>
<td>four</td>
<td>Constant</td>
<td>String representation of the numerical value 4.</td>
</tr>
<tr>
<td>fourth</td>
<td>Ordinal</td>
<td>Designates object or chunk number four within its enclosing set.</td>
</tr>
<tr>
<td>freeSize</td>
<td>Property</td>
<td>Determines the amount of free space available in a specified stack.</td>
</tr>
<tr>
<td>function</td>
<td>Keyword</td>
<td>Marks the beginning of a function handler. Connects the handler with a particular function call.</td>
</tr>
<tr>
<td>functionKey</td>
<td>Command</td>
<td>Performs Undo, Cut, Copy, or Paste operations with parameter values of 1, 2, 3, or 4, respectively.</td>
</tr>
<tr>
<td>functionKey</td>
<td>System message</td>
<td>Sent to current card when any function key on the Apple Extended Keyboard is pressed.</td>
</tr>
<tr>
<td>get</td>
<td>Command</td>
<td>Puts the value of an expression into the local variable It.</td>
</tr>
<tr>
<td>global</td>
<td>Keyword</td>
<td>Declares specified variables to be valid beyond current execution of current handler.</td>
</tr>
<tr>
<td>globalLoc</td>
<td>Property</td>
<td>Determines the location of a window created with the picture command in global coordinates.</td>
</tr>
</tbody>
</table>

*continued*
**APPENDIX I**

HyperTalk Vocabulary

<table>
<thead>
<tr>
<th>Term</th>
<th>Category</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>globalRect</code></td>
<td>Property</td>
<td>Determines the rectangle of a window created with the <code>picture</code> command in global coordinates.</td>
</tr>
<tr>
<td><code>go</code></td>
<td>Command</td>
<td>Takes you to a specified card or stack.</td>
</tr>
<tr>
<td><code>grid</code></td>
<td>Property</td>
<td>Determines the Grid setting.</td>
</tr>
<tr>
<td><code>hBarLoc</code></td>
<td>Property</td>
<td>Determines the location of the horizontal bar in the Variable Watcher window.</td>
</tr>
<tr>
<td><code>heapSpace</code></td>
<td>Function</td>
<td>Returns an integer representing the amount of heap space available to HyperCard.</td>
</tr>
<tr>
<td><code>help</code></td>
<td>Command</td>
<td>Takes you to the first card in the stack named HyperCard Help.</td>
</tr>
<tr>
<td><code>help</code></td>
<td>System message</td>
<td>Sent to the current card, just before leaving that card, when Help is chosen from the Go menu (or Command-? is pressed).</td>
</tr>
<tr>
<td><code>height</code></td>
<td>Property</td>
<td>Determines or changes the vertical distance in pixels occupied by the rectangle of the specified button or field.</td>
</tr>
<tr>
<td><code>hide</code></td>
<td>Command</td>
<td>Hides the specified window from view.</td>
</tr>
<tr>
<td><code>hide groups</code></td>
<td>Command</td>
<td>Hides the gray underline displayed beneath text by the <code>show groups</code> command.</td>
</tr>
<tr>
<td><code>hideIdle</code></td>
<td>Property</td>
<td>Determines whether or not the “Hide idle” checkbox is checked in the Message Watcher window.</td>
</tr>
<tr>
<td><code>hide menubar</code></td>
<td>System message</td>
<td>Hides the HyperCard menu bar.</td>
</tr>
<tr>
<td><code>hideUnused</code></td>
<td>Property</td>
<td>Determines whether or not the “Hide unused messages” checkbox is checked in the Message Watcher window.</td>
</tr>
<tr>
<td><code>hilite</code></td>
<td>Property</td>
<td>Determines whether a specified button is highlighted.</td>
</tr>
</tbody>
</table>

*continued*
# Appendix I

HyperTalk Vocabulary

<table>
<thead>
<tr>
<th>Term</th>
<th>Category</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>hilitedButton</td>
<td>Property</td>
<td>Determines whether a button is highlighted in a palette XCMD displayed by the palette command.</td>
</tr>
<tr>
<td>icon</td>
<td>Property</td>
<td>Determines the icon that is displayed with a specified button.</td>
</tr>
<tr>
<td>ID</td>
<td>Property</td>
<td>Determines the permanent ID number of a specified background, card, field, or button.</td>
</tr>
<tr>
<td>idle</td>
<td>System message</td>
<td>Sent to the current card repeatedly whenever nothing else is happening.</td>
</tr>
<tr>
<td>if</td>
<td>Keyword</td>
<td>Introduces a conditional structure containing statements to be executed only if a specified condition is true.</td>
</tr>
<tr>
<td>import paint</td>
<td>Command</td>
<td>Reads in a Macintosh paint file and makes it the current selection.</td>
</tr>
<tr>
<td>in</td>
<td>Operator</td>
<td>Used with the comparison operators is in and is not in.</td>
</tr>
<tr>
<td>in</td>
<td>Preposition</td>
<td>Used as a connective preposition in chunk expressions—for example, card 12 in this stack.</td>
</tr>
<tr>
<td>into</td>
<td>Preposition</td>
<td>Used with put command, directing HyperCard to replace any preexisting value in a container with a new value.</td>
</tr>
<tr>
<td>It</td>
<td>Container</td>
<td>Local variable that is the default destination for get, ask, answer, read, request, and convert commands.</td>
</tr>
<tr>
<td>item</td>
<td>Chunk</td>
<td>A piece of text delimited by commas in any container or expression.</td>
</tr>
<tr>
<td>itemDelimiter</td>
<td>Property</td>
<td>Determines the character that delimits items in a container.</td>
</tr>
<tr>
<td>items</td>
<td>Chunk type</td>
<td>Specifies items as type of chunk to the number function.</td>
</tr>
</tbody>
</table>

Table I-1 (continued)
<table>
<thead>
<tr>
<th>Term</th>
<th>Category</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>keyDown</td>
<td>Command</td>
<td>Causes HyperCard to enter the character passed with the command at the insertion point.</td>
</tr>
<tr>
<td>keyDown</td>
<td>System message</td>
<td>Sent to the current card when a key is pressed.</td>
</tr>
<tr>
<td>language</td>
<td>Property</td>
<td>Used to choose language in which scripts are displayed.</td>
</tr>
<tr>
<td>lasso</td>
<td>Tool</td>
<td>Name of tool from Tools palette; used with choose command or returned by the tool function.</td>
</tr>
<tr>
<td>last</td>
<td>Ordinal</td>
<td>Special ordinal used with object or chunk to specify the element whose number is equal to the total number of elements in its enclosing set.</td>
</tr>
<tr>
<td>length</td>
<td>Function</td>
<td>Returns the number of characters in the text string derived from an expression.</td>
</tr>
<tr>
<td>left</td>
<td>Adjective</td>
<td>Specifies left-justified alignment of text in a field.</td>
</tr>
<tr>
<td>left</td>
<td>Property</td>
<td>Determines or changes the value of item 1 of the rectangle property when applied to the specified object or window.</td>
</tr>
<tr>
<td>line</td>
<td>Chunk</td>
<td>A piece of text delimited by return characters in any container.</td>
</tr>
<tr>
<td>line</td>
<td>Tool</td>
<td>Name of tool from Tools palette; used with choose command or returned by the tool function.</td>
</tr>
<tr>
<td>lineFeed</td>
<td>Constant</td>
<td>The line feed character (ASCII 10), which starts a new line in some file formats.</td>
</tr>
<tr>
<td>lines</td>
<td>Chunk type</td>
<td>Specifies lines as type of chunk to the number function.</td>
</tr>
<tr>
<td>lineSize</td>
<td>Property</td>
<td>Determines the thickness of lines drawn with line and shape tools.</td>
</tr>
</tbody>
</table>

continued
## HyperTalk Vocabulary

<table>
<thead>
<tr>
<th>Term</th>
<th>Category</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ln</td>
<td>Function</td>
<td>Returns the base-(e) (natural) logarithm of the number passed to it.</td>
</tr>
<tr>
<td>ln1</td>
<td>Function</td>
<td>Returns the base-(e) (natural) logarithm of the sum of the number passed to it plus 1.</td>
</tr>
<tr>
<td>loc</td>
<td>Property</td>
<td>Determines the location at which a picture window created with the <code>picture</code> command is displayed.</td>
</tr>
<tr>
<td>loc[ation]</td>
<td>Property</td>
<td>Determines the location at which a window, field, or button is displayed.</td>
</tr>
<tr>
<td>lock</td>
<td>Command</td>
<td>Prevents updating of the screen from card to card.</td>
</tr>
<tr>
<td>lockErrorDialogs</td>
<td>Property</td>
<td>Allows or prevents HyperCard from displaying error messages.</td>
</tr>
<tr>
<td>lockMessages</td>
<td>Property</td>
<td>Allows or prevents HyperCard from sending all automatic messages such as <code>openCard</code>.</td>
</tr>
<tr>
<td>lockRecent</td>
<td>Property</td>
<td>Allows or prevents HyperCard from adding miniature representations to the Recent card.</td>
</tr>
<tr>
<td>lockScreen</td>
<td>Property</td>
<td>Determines whether the screen is updated when moving from card to card.</td>
</tr>
<tr>
<td>lockText</td>
<td>Property</td>
<td>Determines whether text editing is allowed in a specified field.</td>
</tr>
<tr>
<td>log2</td>
<td>Function</td>
<td>Returns the base-2 logarithm of the number passed to it.</td>
</tr>
<tr>
<td>long</td>
<td>Adjective</td>
<td>Modifies value returned by <code>date</code> function and by <code>name</code> and <code>ID</code> properties.</td>
</tr>
<tr>
<td>longWindowTitles</td>
<td>Property</td>
<td>Determines whether the window title bar contains the full pathname of a stack or the short name.</td>
</tr>
<tr>
<td>mark</td>
<td>Command</td>
<td>Marks cards.</td>
</tr>
</tbody>
</table>

*continued*
## APPENDIX I

HyperTalk Vocabulary

### Table I-1  HyperTalk vocabulary (continued)

<table>
<thead>
<tr>
<th>Term</th>
<th>Category</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>markChar</td>
<td>Property</td>
<td>Determines the checkmark character used to indicate a menu item is chosen.</td>
</tr>
<tr>
<td>marked</td>
<td>Property</td>
<td>Determines whether or not a specified card is marked.</td>
</tr>
<tr>
<td>max</td>
<td>Function</td>
<td>Returns the highest-value number from a list of numbers.</td>
</tr>
<tr>
<td>me</td>
<td>Object</td>
<td>The object containing the executing handler.</td>
</tr>
<tr>
<td>menu</td>
<td>Function</td>
<td>Returns a list of the menu items in a specified menu.</td>
</tr>
<tr>
<td>menuMessage</td>
<td>Property</td>
<td>Determines the message to be sent by a specified menu item.</td>
</tr>
<tr>
<td>menus</td>
<td>Function</td>
<td>Returns a list of the menu names in the HyperCard menu bar.</td>
</tr>
<tr>
<td>messageWatcher</td>
<td>Property</td>
<td>Determines the message watcher to use.</td>
</tr>
<tr>
<td>mid[dle]</td>
<td>Ordinal</td>
<td>Special ordinal used with object or chunk to specify the element whose number is equal to one more than half the total number of elements in its enclosing set.</td>
</tr>
<tr>
<td>min</td>
<td>Function</td>
<td>Returns the lowest-value number from a list of numbers.</td>
</tr>
<tr>
<td>mouse</td>
<td>Function</td>
<td>Returns state of the mouse button: up or down.</td>
</tr>
<tr>
<td>mouseClicked</td>
<td>Function</td>
<td>Returns whether the mouse button has been clicked.</td>
</tr>
<tr>
<td>mouseDoubleClick</td>
<td>System message</td>
<td>Sent to a button, locked field, or the current card when the mouse button is double-clicked.</td>
</tr>
<tr>
<td>mouseDown</td>
<td>System message</td>
<td>Sent to a button, locked field, or the current card when the mouse button is pressed down.</td>
</tr>
</tbody>
</table>
## APPENDIX I

HyperTalk Vocabulary

<table>
<thead>
<tr>
<th>Term</th>
<th>Category</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>mouseDownInPicture</td>
<td>System message</td>
<td>Sent to the current card when the mouse button is down while the pointer is in a window created with the picture command.</td>
</tr>
<tr>
<td>mouseEnter</td>
<td>System message</td>
<td>Sent to a button or field when the pointer is first moved inside its rectangle.</td>
</tr>
<tr>
<td>mouseH</td>
<td>Function</td>
<td>Returns the horizontal offset in pixels of the pointer from the left edge of the card window.</td>
</tr>
<tr>
<td>mouseLeave</td>
<td>System message</td>
<td>Sent to a button or field when the pointer is first removed from its rectangle.</td>
</tr>
<tr>
<td>mouseLoc</td>
<td>Function</td>
<td>Returns the point on the screen where the pointer is currently located.</td>
</tr>
<tr>
<td>mouseStillDown</td>
<td>System message</td>
<td>Sent to a button, locked field, or the current card repeatedly when the mouse button is held down.</td>
</tr>
<tr>
<td>mouseUp</td>
<td>System message</td>
<td>Sent to a button, locked field, or the current card when the mouse button is released after having been previously pressed down within the same object’s rectangle.</td>
</tr>
<tr>
<td>mouseUpInPicture</td>
<td>System message</td>
<td>Sent to the current card when the mouse button is released after being down while the pointer is in a window created with the picture command.</td>
</tr>
<tr>
<td>mouseV</td>
<td>Function</td>
<td>Returns the vertical offset in pixels of the pointer from the top of the screen.</td>
</tr>
<tr>
<td>mouseWithin</td>
<td>System message</td>
<td>Sent to a button or field repeatedly while the pointer remains inside its rectangle.</td>
</tr>
<tr>
<td>moveWindow</td>
<td>System message</td>
<td>Sent to a card when you change a card window’s location property with HyperTalk, drag or zoom the card window, or change the location of the card window with the show command.</td>
</tr>
</tbody>
</table>

Table I-1 HyperTalk vocabulary (continued)

continued
### Table I-1  HyperTalk vocabulary (continued)

<table>
<thead>
<tr>
<th>Term</th>
<th>Category</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>msg [box]</td>
<td>Container</td>
<td>The Message box.</td>
</tr>
<tr>
<td>multiple</td>
<td>Property</td>
<td>Determines whether multiple images are drawn with a shape tool.</td>
</tr>
<tr>
<td>multipleLines</td>
<td>Property</td>
<td>Used to determine or change whether multiple-line selections are allowed in a field configured as a list field.</td>
</tr>
<tr>
<td>multiply</td>
<td>Command</td>
<td>Multiplies the value in a container by the value derived from an expression.</td>
</tr>
<tr>
<td>multiSpace</td>
<td>Property</td>
<td>Determines the space between objects drawn when the multiple property is true.</td>
</tr>
<tr>
<td>name</td>
<td>Property</td>
<td>Determines the name of a stack, background, card, field, button, menu, or menu item.</td>
</tr>
<tr>
<td>next</td>
<td>Keyword</td>
<td>Ends execution of current iteration of a repeat loop, beginning next iteration.</td>
</tr>
<tr>
<td>next</td>
<td>Object modifier</td>
<td>Used with card or background to refer to the one following the current one.</td>
</tr>
<tr>
<td>newBackground</td>
<td>System message</td>
<td>Sent to the current card as soon as a background has been created.</td>
</tr>
<tr>
<td>newButton</td>
<td>System message</td>
<td>Sent to a button as soon as it has been created.</td>
</tr>
<tr>
<td>newCard</td>
<td>System message</td>
<td>Sent to a card as soon as it has been created.</td>
</tr>
<tr>
<td>newField</td>
<td>System message</td>
<td>Sent to a field as soon as it has been created.</td>
</tr>
<tr>
<td>newStack</td>
<td>System message</td>
<td>Sent to the current card as soon as a stack has been created.</td>
</tr>
<tr>
<td>nine</td>
<td>Constant</td>
<td>String representation of the numerical value 9.</td>
</tr>
<tr>
<td>ninth</td>
<td>Ordinal</td>
<td>Designates object or chunk number nine within its enclosing set.</td>
</tr>
</tbody>
</table>

continued
## HyperTalk Vocabulary

### Table I-1 HyperTalk vocabulary (continued)

<table>
<thead>
<tr>
<th>Term</th>
<th>Category</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>Function</td>
<td>Returns the number of buttons or fields on the current card or background, the number of marked cards, the number of HyperCard menus, the number of menu items in a menu, the number of windows in HyperCard, or the number of a specified type of chunk within a value.</td>
</tr>
<tr>
<td>number</td>
<td>Property</td>
<td>Determines the number of a background, card, field, or button.</td>
</tr>
<tr>
<td>numberFormat</td>
<td>Property</td>
<td>Determines the precision with which results of mathematical operations are displayed.</td>
</tr>
<tr>
<td>numToChar</td>
<td>Function</td>
<td>Returns the character whose ASCII equivalent value is that of the integer passed to it.</td>
</tr>
<tr>
<td>offset</td>
<td>Function</td>
<td>Returns the number of characters from the beginning of the source string.</td>
</tr>
<tr>
<td>on</td>
<td>Keyword</td>
<td>Marks the beginning of a message handler and connects it with a particular message.</td>
</tr>
<tr>
<td>one</td>
<td>Constant</td>
<td>String representation of the numerical value 1.</td>
</tr>
<tr>
<td>open</td>
<td>Command</td>
<td>Launches the specified application.</td>
</tr>
<tr>
<td>openBackground</td>
<td>System message</td>
<td>Sent to a card when you go to it and its background is different from the one you were formerly on.</td>
</tr>
<tr>
<td>openCard</td>
<td>System message</td>
<td>Sent to a card when you go to it.</td>
</tr>
<tr>
<td>openField</td>
<td>System message</td>
<td>Sent to an unlocked field when you place the insertion point in it for text editing.</td>
</tr>
<tr>
<td>open file</td>
<td>Command</td>
<td>Opens the specified file for a read or write command operation.</td>
</tr>
<tr>
<td>openPalette</td>
<td>System message</td>
<td>Sent to the current card when a palette is opened with the palette command.</td>
</tr>
</tbody>
</table>

*continued*
### Table I-1  HyperTalk vocabulary (continued)

<table>
<thead>
<tr>
<th>Term</th>
<th>Category</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>openPicture</td>
<td>System message</td>
<td>Sent to the current card when a palette is opened with the picture command.</td>
</tr>
<tr>
<td>open printing</td>
<td>Command</td>
<td>Begins a print job.</td>
</tr>
<tr>
<td>open report printing</td>
<td>Command</td>
<td>Begins a print job for a specified report.</td>
</tr>
<tr>
<td>openStack</td>
<td>System message</td>
<td>Sent to a card when you go to it and it’s in a stack different from the one containing the card you were formerly on.</td>
</tr>
<tr>
<td>optionKey</td>
<td>Function</td>
<td>Returns the state of the Option key: up or down.</td>
</tr>
<tr>
<td>oval</td>
<td>Tool</td>
<td>Name of tool from Tools palette; used with choose command or returned by the tool function.</td>
</tr>
<tr>
<td>owner</td>
<td>Property</td>
<td>For a card, determines its background; for a window, determines its creator.</td>
</tr>
<tr>
<td>palette</td>
<td>Command</td>
<td>Invokes the specified palette XCMD.</td>
</tr>
<tr>
<td>param</td>
<td>Function</td>
<td>Returns a parameter value from the parameter list passed to the currently executing handler.</td>
</tr>
<tr>
<td>paramCount</td>
<td>Function</td>
<td>Returns the number of parameters passed to the currently executing handler.</td>
</tr>
<tr>
<td>params</td>
<td>Function</td>
<td>Returns the entire parameter list passed to the currently executing handler.</td>
</tr>
<tr>
<td>partNumber</td>
<td>Property</td>
<td>Determines the position of a button or field among all the buttons and fields of its enclosing card or background.</td>
</tr>
<tr>
<td>pass</td>
<td>Keyword</td>
<td>Ends execution of a message handler or function handler and sends the invoking message or function call to the next object in the hierarchy.</td>
</tr>
<tr>
<td>pattern</td>
<td>Property</td>
<td>Determines the Paint pattern.</td>
</tr>
</tbody>
</table>
### Table I-1 HyperTalk vocabulary (continued)

<table>
<thead>
<tr>
<th>Term</th>
<th>Category</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>pencil</td>
<td>Tool</td>
<td>Name of tool from Tools palette; used with <code>choose</code> command or returned by the <code>tool</code> function.</td>
</tr>
<tr>
<td>pi</td>
<td>Constant</td>
<td>The mathematical value pi to 20 decimal places, equal to the number 3.14159265358979323846.</td>
</tr>
<tr>
<td>picture</td>
<td>Command</td>
<td>Displays the specified picture file in an external window.</td>
</tr>
<tr>
<td>play</td>
<td>Command</td>
<td>Starts the HyperCard sound-playing feature.</td>
</tr>
<tr>
<td>poly[gon]</td>
<td>Tool</td>
<td>Name of tool from Tools palette; used with <code>choose</code> command or returned by the <code>tool</code> function.</td>
</tr>
<tr>
<td>polySides</td>
<td>Property</td>
<td>Determines the number of sides created by the Regular Polygon tool.</td>
</tr>
<tr>
<td>pop card</td>
<td>Command</td>
<td>Returns you to last card saved with the <code>push card</code> command.</td>
</tr>
<tr>
<td>powerKeys</td>
<td>Property</td>
<td>Provides keyboard shortcuts of commonly used painting actions.</td>
</tr>
<tr>
<td>prev[ious]</td>
<td>Object modifier</td>
<td>Used with <code>card</code> or <code>background</code> to refer to the one preceding the current one.</td>
</tr>
<tr>
<td>print</td>
<td>Command</td>
<td>Prints the specified file.</td>
</tr>
<tr>
<td>print card</td>
<td>Command</td>
<td>Prints the current card or a specified number of cards beginning with the current card.</td>
</tr>
<tr>
<td>printMargins</td>
<td>Property</td>
<td>Determines or sets the current page print margin.</td>
</tr>
<tr>
<td>printTextAlign</td>
<td>Property</td>
<td>Determines or sets the text alignment for fields when printing.</td>
</tr>
<tr>
<td>printTextFont</td>
<td>Property</td>
<td>Determines or sets the text font for fields when printing.</td>
</tr>
</tbody>
</table>
### HyperTalk Vocabulary (continued)

<table>
<thead>
<tr>
<th>Term</th>
<th>Category</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>printTextHeight</td>
<td>Property</td>
<td>Determines or sets the line height for fields when printing.</td>
</tr>
<tr>
<td>printTextSize</td>
<td>Property</td>
<td>Determines or sets the text size for fields when printing.</td>
</tr>
<tr>
<td>printTextStyle</td>
<td>Property</td>
<td>Determines or sets the text style for fields when printing.</td>
</tr>
<tr>
<td>programs</td>
<td>Function</td>
<td>Returns a list of the System 7–friendly programs running on your machine.</td>
</tr>
<tr>
<td>properties</td>
<td>Property</td>
<td>Returns a list of the names of the palette properties supported by a palette XCMD displayed by the palette command.</td>
</tr>
<tr>
<td>push</td>
<td>Command</td>
<td>Saves the identification of a specified card in a LIFO memory stack for later retrieval.</td>
</tr>
<tr>
<td>put</td>
<td>Command</td>
<td>Copies the value of an expression into a container.</td>
</tr>
<tr>
<td>quit</td>
<td>System message</td>
<td>Sent to the current card when you choose Quit HyperCard from the File menu (or press Command-Q), just before HyperCard goes away.</td>
</tr>
<tr>
<td>quote</td>
<td>Constant</td>
<td>The straight double quotation mark character.</td>
</tr>
<tr>
<td>random</td>
<td>Function</td>
<td>Returns a random integer between 1 and the integer derived from a specified expression.</td>
</tr>
<tr>
<td>read</td>
<td>Command</td>
<td>Reads a file previously opened with the open file command into the local variable It. See also write.</td>
</tr>
<tr>
<td>rect</td>
<td>Property</td>
<td>Determines the rectangle property for a variable watcher window and for windows created with the picture command.</td>
</tr>
<tr>
<td>rect[angle]</td>
<td>Property</td>
<td>Determines the rectangle occupied by a specified window, field, or button.</td>
</tr>
</tbody>
</table>
## HyperTalk Vocabulary

### Table I-1
HyperTalk vocabulary (continued)

<table>
<thead>
<tr>
<th>Term</th>
<th>Category</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>rect[angle]</td>
<td>Tool</td>
<td>Name of tool from Tools palette; used with choose command or returned by the tool function.</td>
</tr>
<tr>
<td>reg[ular] poly[gon]</td>
<td>Tool</td>
<td>Name of tool from Tools palette; used with choose command or returned by the tool function.</td>
</tr>
<tr>
<td>repeat</td>
<td>Keyword</td>
<td>Introduces a repeat loop, an iterative structure containing a block of one or more statements executed multiple times.</td>
</tr>
<tr>
<td>reply</td>
<td>Command</td>
<td>Used to answer an incoming Apple event.</td>
</tr>
<tr>
<td>reportTemplates</td>
<td>Property</td>
<td>A read-only stack property that returns the names of the report templates for a stack.</td>
</tr>
<tr>
<td>request</td>
<td>Command</td>
<td>Sends an “evaluate expression” Apple event to another application.</td>
</tr>
<tr>
<td>reset menubar</td>
<td>Command</td>
<td>Reinstates the default values of all the HyperCard menus and removes any user-defined menus.</td>
</tr>
<tr>
<td>reset paint</td>
<td>Command</td>
<td>Reinstates the default values of all the painting properties.</td>
</tr>
<tr>
<td>reset printing</td>
<td>Command</td>
<td>Reinstates the default values of all the printing properties.</td>
</tr>
<tr>
<td>result</td>
<td>Function</td>
<td>Returns the status of commands previously executed in current handler.</td>
</tr>
<tr>
<td>resume</td>
<td>System message</td>
<td>Sent to the current card when HyperCard resumes running after having been suspended.</td>
</tr>
<tr>
<td>resumeStack</td>
<td>System message</td>
<td>Sent to the current card when HyperCard returns to a stack.</td>
</tr>
<tr>
<td>return</td>
<td>Keyword</td>
<td>Returns a value from a function handler or message handler.</td>
</tr>
</tbody>
</table>

*continued*
**APPENDIX I**

HyperTalk Vocabulary

<table>
<thead>
<tr>
<th>Term</th>
<th>Category</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>returnInField</td>
<td>Command</td>
<td>Enters a return character into a field that is open for text editing.</td>
</tr>
<tr>
<td>returnInField</td>
<td>System message</td>
<td>Sent to a field when the Return key is pressed and there is an insertion point or selection in the field.</td>
</tr>
<tr>
<td>returnKey</td>
<td>Command</td>
<td>Sends any statement in the Message box to the current card.</td>
</tr>
<tr>
<td>returnKey</td>
<td>System message</td>
<td>Sent to current card when Return key is pressed.</td>
</tr>
<tr>
<td>right</td>
<td>Adjective</td>
<td>Specifies right-justified alignment of text in a field.</td>
</tr>
<tr>
<td>right</td>
<td>Property</td>
<td>Determines or changes the value of item 3 of the rectangle property when applied to the specified object or window.</td>
</tr>
<tr>
<td>round</td>
<td>Function</td>
<td>Returns the number derived from an expression, rounded off to the nearest integer.</td>
</tr>
<tr>
<td>round rect[angle]</td>
<td>Tool</td>
<td>Name of tool from Tools palette; used with choose command or returned by the tool function.</td>
</tr>
<tr>
<td>save</td>
<td>Command</td>
<td>Saves a copy of the specified stack with a specified name.</td>
</tr>
<tr>
<td>screenRect</td>
<td>Function</td>
<td>Returns the size of the screen HyperCard’s menu bar is in.</td>
</tr>
<tr>
<td>script</td>
<td>Property</td>
<td>Retrieves or replaces the script of the specified stack, background, card, field, or button.</td>
</tr>
<tr>
<td>scriptEditor</td>
<td>Property</td>
<td>Determines the script editor to use.</td>
</tr>
<tr>
<td>scriptingLanguage</td>
<td>Property</td>
<td>Used to set HyperCard objects to accept scripts written in the scripting language you choose.</td>
</tr>
</tbody>
</table>

*continued*
## HyperTalk Vocabulary

Table I-1  HyperTalk vocabulary (continued)

<table>
<thead>
<tr>
<th>Term</th>
<th>Category</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>scriptTextFont</td>
<td>Property</td>
<td>Determines the font displayed in the script editor.</td>
</tr>
<tr>
<td>scriptTextSize</td>
<td>Property</td>
<td>Determines the size of text displayed in the script editor.</td>
</tr>
<tr>
<td>scroll (fields)</td>
<td>Property</td>
<td>Determines the amount of material that is hidden above the top of the specified scrolling field's rectangle.</td>
</tr>
<tr>
<td>scroll (windows)</td>
<td>Property</td>
<td>Determines the position of the window over the card or picture.</td>
</tr>
<tr>
<td>second</td>
<td>Ordinal</td>
<td>Designates object or chunk number two within its enclosing set.</td>
</tr>
<tr>
<td>seconds</td>
<td>Function</td>
<td>Returns the number of seconds between midnight, January 1, 1904, and the current time.</td>
</tr>
<tr>
<td>select</td>
<td>Command</td>
<td>Selects an object, a tool, a chunk of text, or a line in a list field or pop-up button.</td>
</tr>
<tr>
<td>select</td>
<td>Tool</td>
<td>Name of tool from Tools palette; used with choose command or returned by the tool function.</td>
</tr>
<tr>
<td>selectedButton</td>
<td>Function</td>
<td>Returns the descriptor of the currently highlighted button.</td>
</tr>
<tr>
<td>selectedChunk</td>
<td>Function</td>
<td>Returns a chunk expression describing the selected text in a field.</td>
</tr>
<tr>
<td>selectedField</td>
<td>Function</td>
<td>Returns an expression describing the field the selected text is in.</td>
</tr>
<tr>
<td>selectedLine</td>
<td>Function</td>
<td>Returns an expression describing the line in a field where the selected text is.</td>
</tr>
<tr>
<td>selectedLoc</td>
<td>Function</td>
<td>Returns the point at which the selected text begins.</td>
</tr>
<tr>
<td>selectedText</td>
<td>Function</td>
<td>Returns the selected text in a field.</td>
</tr>
</tbody>
</table>

*continued*
### Table I-1  HyperTalk vocabulary (continued)

<table>
<thead>
<tr>
<th>Term</th>
<th>Category</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection</td>
<td>Container</td>
<td>Currently selected area of text in a field.</td>
</tr>
<tr>
<td>send</td>
<td>Keyword</td>
<td>Sends a specified message directly to a specified object; sends a <code>do script</code> Apple event to another application.</td>
</tr>
<tr>
<td>set</td>
<td>Command</td>
<td>Changes the state of a specified global, painting, window, or object property.</td>
</tr>
<tr>
<td>seven</td>
<td>Constant</td>
<td>String representation of the numerical value 7.</td>
</tr>
<tr>
<td>seventh</td>
<td>Ordinal</td>
<td>Designates object or chunk number seven within its enclosing set.</td>
</tr>
<tr>
<td>sharedHilite</td>
<td>Property</td>
<td>Determines or sets whether a background button shares the same highlight state on each card.</td>
</tr>
<tr>
<td>sharedText</td>
<td>Property</td>
<td>Determines or sets whether a background field shares the same text on each card. If set to <code>true</code>, it also sets the <code>dontSearch</code> property of the field to <code>true</code>.</td>
</tr>
<tr>
<td>shiftKey</td>
<td>Function</td>
<td>Returns the state of the Shift key: up or down.</td>
</tr>
<tr>
<td>short</td>
<td>Adjective</td>
<td>Modifies value returned by <code>date</code> function and by name and <code>ID</code> properties.</td>
</tr>
<tr>
<td>show</td>
<td>Command</td>
<td>Displays a specified window or object.</td>
</tr>
<tr>
<td>show cards</td>
<td>Command</td>
<td>Displays a specified number of cards in the current stack.</td>
</tr>
<tr>
<td>showLines</td>
<td>Property</td>
<td>Determines whether or not the text baselines are visible in a field.</td>
</tr>
<tr>
<td>show menubar</td>
<td>System message</td>
<td>Displays the menu bar if it was hidden.</td>
</tr>
<tr>
<td>showName</td>
<td>Property</td>
<td>Determines whether or not the name of a specified button is displayed in its rectangle on the screen.</td>
</tr>
</tbody>
</table>

*continued*
## Table I-1 HyperTalk vocabulary (continued)

<table>
<thead>
<tr>
<th>Term</th>
<th>Category</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>showPict</td>
<td>Property</td>
<td>Determines whether or not a specified card or background picture is displayed.</td>
</tr>
<tr>
<td>show titlebar</td>
<td>Command</td>
<td>Shows the title bar of the current card window if it was hidden.</td>
</tr>
<tr>
<td>sin</td>
<td>Function</td>
<td>Returns the sine of the angle that is passed to it.</td>
</tr>
<tr>
<td>six</td>
<td>Constant</td>
<td>String representation of the numerical value 6.</td>
</tr>
<tr>
<td>sixth</td>
<td>Ordinal</td>
<td>Designates object or chunk number six within its enclosing set.</td>
</tr>
<tr>
<td>size</td>
<td>Property</td>
<td>Returns the size of a specified stack.</td>
</tr>
<tr>
<td>sizeWindow</td>
<td>System message</td>
<td>Sent to the current card when the card window is resized.</td>
</tr>
<tr>
<td>sort</td>
<td>Command</td>
<td>Puts all of the cards in a specified stack in a specified order.</td>
</tr>
<tr>
<td>sound</td>
<td>Function</td>
<td>Returns the name of the sound that is currently playing.</td>
</tr>
<tr>
<td>space</td>
<td>Constant</td>
<td>The space character (ASCII 32); same as the literal &quot; &quot;.</td>
</tr>
<tr>
<td>spray [can]</td>
<td>Tool</td>
<td>Name of tool from Tools palette; used with choose command or returned by the tool function.</td>
</tr>
<tr>
<td>sqrt</td>
<td>Function</td>
<td>Returns the square root of a number.</td>
</tr>
<tr>
<td>stack</td>
<td>Object</td>
<td>Generic name of stack object; used with specific name (go to stack &quot;help&quot;).</td>
</tr>
<tr>
<td>stacks</td>
<td>Function</td>
<td>Returns a list of the currently open stacks.</td>
</tr>
<tr>
<td>stacksInUse</td>
<td>Property</td>
<td>Determines the current list of stacks in the message-passing hierarchy.</td>
</tr>
<tr>
<td>stackSpace</td>
<td>Function</td>
<td>Returns the amount of space remaining on the Macintosh Operating System stack.</td>
</tr>
</tbody>
</table>
# HyperTalk Vocabulary (continued)

<table>
<thead>
<tr>
<th>Term</th>
<th>Category</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>startUp</td>
<td>System message</td>
<td>Sent to the current card (first card of the Home stack) when HyperCard first begins running.</td>
</tr>
<tr>
<td>start using</td>
<td>Command</td>
<td>Specifies a stack to add to the message-passing hierarchy.</td>
</tr>
<tr>
<td>stop using</td>
<td>Command</td>
<td>Specifies a stack to remove from the message-passing hierarchy.</td>
</tr>
<tr>
<td>style</td>
<td>Property</td>
<td>Determines the style of a specified field or button.</td>
</tr>
<tr>
<td>subtract</td>
<td>Command</td>
<td>Subtracts the value of an expression from the value in a container.</td>
</tr>
<tr>
<td>sum</td>
<td>Function</td>
<td>Returns the sum of a list of numbers.</td>
</tr>
<tr>
<td>suspend</td>
<td>System message</td>
<td>Sent to the current card when HyperCard is suspended by launching another application with the open command.</td>
</tr>
<tr>
<td>suspendStack</td>
<td>System message</td>
<td>Sent to the current card when you leave an open stack to go to another.</td>
</tr>
<tr>
<td>systemVersion</td>
<td>Function</td>
<td>Returns a decimal string representing the running version of system software.</td>
</tr>
<tr>
<td>tab</td>
<td>Constant</td>
<td>The horizontal tab character (ASCII 9).</td>
</tr>
<tr>
<td>tabKey</td>
<td>Command</td>
<td>Places the insertion point in the next unlocked field on the current background or card.</td>
</tr>
<tr>
<td>tabKey</td>
<td>System message</td>
<td>Sent to the current card or a field when Tab key is pressed.</td>
</tr>
<tr>
<td>tan</td>
<td>Function</td>
<td>Returns the tangent of an angle.</td>
</tr>
<tr>
<td>target</td>
<td>Function</td>
<td>Indicates the object that initially received the message that initiated execution of the current handler.</td>
</tr>
<tr>
<td>ten</td>
<td>Constant</td>
<td>String representation of the numerical value 10.</td>
</tr>
</tbody>
</table>

*continued*
## APPENDIX I

HyperTalk Vocabulary

### Table I-1 HyperTalk vocabulary (continued)

<table>
<thead>
<tr>
<th>Term</th>
<th>Category</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>tenth</td>
<td>Ordinal</td>
<td>Designates object or chunk number ten within its enclosing set.</td>
</tr>
<tr>
<td>text</td>
<td>Tool</td>
<td>Name of tool from Tools palette; used with choose command or returned by the tool function.</td>
</tr>
<tr>
<td>textAlign</td>
<td>Property</td>
<td>Determines the alignment of characters created with the Paint Text tool, or those in a field, or those in the name of a button.</td>
</tr>
<tr>
<td>textArrows</td>
<td>Property</td>
<td>Determines the functions of the arrow keys.</td>
</tr>
<tr>
<td>textFont</td>
<td>Property</td>
<td>Determines the font of characters created with the Paint Text tool, or those in a field, or those in the name of a button.</td>
</tr>
<tr>
<td>textHeight</td>
<td>Property</td>
<td>Determines the space between the baseline and characters created with the Paint Text tool or those in a field.</td>
</tr>
<tr>
<td>textSize</td>
<td>Property</td>
<td>Determines the size of Paint text, or text in a field, or text in the name of a button.</td>
</tr>
<tr>
<td>textStyle</td>
<td>Property</td>
<td>Determines the style of Paint text, text in a field, text in the name of a button, or text of a menu item.</td>
</tr>
<tr>
<td>the</td>
<td>Special</td>
<td>Precedes a function name to indicate a function call to one of the built-in functions of HyperCard. You can’t call a user-defined function with the. Also allowed, but not required, preceding special container names (the Message box) and properties.</td>
</tr>
<tr>
<td>then</td>
<td>Keyword</td>
<td>Follows the conditional expression in an if structure to introduce the action clause.</td>
</tr>
<tr>
<td>third</td>
<td>Ordinal</td>
<td>Designates object or chunk number three within its enclosing set.</td>
</tr>
<tr>
<td>this</td>
<td>Modifier</td>
<td>Used with card, background, or stack to refer to the current one.</td>
</tr>
</tbody>
</table>

*continued*
### HyperTalk Vocabulary

<table>
<thead>
<tr>
<th>Term</th>
<th>Category</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>three</td>
<td>Constant</td>
<td>String representation of the numerical value 3.</td>
</tr>
<tr>
<td>ticks</td>
<td>Function</td>
<td>Determines the number of ticks since the Macintosh was turned on or restarted.</td>
</tr>
<tr>
<td>time</td>
<td>Function</td>
<td>Returns the current time as a text string.</td>
</tr>
<tr>
<td>titleWidth</td>
<td>Property</td>
<td>Determines or changes the width of the area in a pop-up button which displays its name.</td>
</tr>
<tr>
<td>to</td>
<td>Preposition</td>
<td>Used to specify ranges (3 to 5), connect a message to its destination when used with send, specify a format for the convert command, assign a container for the add command, and connect values to object properties.</td>
</tr>
<tr>
<td>tool</td>
<td>Function</td>
<td>Returns the name of the currently chosen tool.</td>
</tr>
<tr>
<td>top</td>
<td>Property</td>
<td>Determines or changes the value of item 2 of the rectangle property when applied to the specified object or window.</td>
</tr>
<tr>
<td>topLeft</td>
<td>Property</td>
<td>Determines or changes items 1 and 2 of the value of the rectangle property when applied to the specified object or window.</td>
</tr>
<tr>
<td>traceDelay</td>
<td>Property</td>
<td>Determines or changes the delay between the execution of lines of HyperTalk during a debugger trace.</td>
</tr>
<tr>
<td>true</td>
<td>Constant</td>
<td>Boolean value resulting from evaluation of a comparative expression and returned from some functions.</td>
</tr>
<tr>
<td>trunc</td>
<td>Function</td>
<td>Determines the integer part of a number.</td>
</tr>
<tr>
<td>two</td>
<td>Constant</td>
<td>String representation of the numerical value 2.</td>
</tr>
</tbody>
</table>

*continued*
### HyperTalk Vocabulary (continued)

<table>
<thead>
<tr>
<th>Term</th>
<th>Category</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>Command</td>
<td>Inserts the specified text at the insertion point.</td>
</tr>
<tr>
<td>unlock</td>
<td>Command</td>
<td>Allows updating of the screen.</td>
</tr>
<tr>
<td>unmark</td>
<td>Command</td>
<td>Unmarks the specified marked card.</td>
</tr>
<tr>
<td>up</td>
<td>Constant</td>
<td>Value returned by various functions to describe the state of a key or the mouse button.</td>
</tr>
<tr>
<td>userLevel</td>
<td>Property</td>
<td>Determines the user level from 1 to 5.</td>
</tr>
<tr>
<td>userModify</td>
<td>Property</td>
<td>Determines or changes whether or not the user can type into fields or use Paint tools on a stack that has been write-protected.</td>
</tr>
<tr>
<td>value</td>
<td>Function</td>
<td>Evaluates an expression.</td>
</tr>
<tr>
<td>variableWatcher</td>
<td>Property</td>
<td>Determines the variable watcher to use.</td>
</tr>
<tr>
<td>vBarLoc</td>
<td>Property</td>
<td>Determines the location of the vertical bar in the Variable Watcher window.</td>
</tr>
<tr>
<td>version</td>
<td>Property</td>
<td>Returns the version number of the currently running HyperCard application.</td>
</tr>
<tr>
<td>visible</td>
<td>Property</td>
<td>Determines whether or not a window, field, or button appears on the screen.</td>
</tr>
<tr>
<td>visual</td>
<td>Command</td>
<td>Sets up a specified visual transition to the next card opened.</td>
</tr>
<tr>
<td>wait</td>
<td>Command</td>
<td>Causes HyperCard to pause before executing the rest of the current handler.</td>
</tr>
<tr>
<td>wideMargins</td>
<td>Property</td>
<td>Determines whether or not additional space is displayed in the margins of a specified field.</td>
</tr>
<tr>
<td>width</td>
<td>Property</td>
<td>Determines or changes the horizontal distance in pixels occupied by the rectangle of the specified button or field.</td>
</tr>
</tbody>
</table>

*continued*
### APPENDIX I

HyperTalk Vocabulary

<table>
<thead>
<tr>
<th>Term</th>
<th>Category</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>windows</td>
<td>Function</td>
<td>Returns a list of the windows currently available to HyperCard.</td>
</tr>
<tr>
<td>within</td>
<td>Operator</td>
<td>Tests whether or not a point lies inside a specified rectangle.</td>
</tr>
<tr>
<td>word</td>
<td>Chunk</td>
<td>Piece of text delimited by spaces in any container or expression.</td>
</tr>
<tr>
<td>words</td>
<td>Chunk type</td>
<td>Specifies words as type of chunk to the number function.</td>
</tr>
<tr>
<td>write</td>
<td>Command</td>
<td>Copies specified text into a specified disk file starting at a specified point.</td>
</tr>
<tr>
<td>zero</td>
<td>Constant</td>
<td>String representation of the numerical value 0.</td>
</tr>
<tr>
<td>zoom</td>
<td>Property</td>
<td>Zooms a window created with the picture command in or out.</td>
</tr>
</tbody>
</table>
Glossary

actual parameters  See parameters.

'aete' resource  Apple event terminology extension resource. This resource includes the “grammar” for a scriptable application, including the events that the application can respond to and the classes of objects that it supports, with their relevant properties and their default data types. The 'aete' resource allows scripting components to map scripts written in human terms to the corresponding Apple events understood by the application.

algorithm  A step-by-step procedure for solving a problem or accomplishing a task. Writing HyperTalk handlers or programs in other languages often begins with figuring out a suitable algorithm for a task.

Apple event  A high-level event that adheres to the Apple Event Interprocess Messaging Protocol. An Apple event consists of attributes (including the event class and event ID, which identify the event and its task) and, usually, parameters (which contain data used by the target application of the event). See also Apple event attribute, Apple event parameter.

Apple event attribute  A keyword-specified descriptor record that identifies the event class, event ID, target application, or some other characteristic of an Apple event. Taken together, the attributes of an Apple event identify the event and denote the task to be performed on the data specified in the Apple event’s parameters. Compared to parameters (which contain data used only by the target application of the Apple event), attributes contain information that can be used by both the Apple Event Manager and the target application. See also Apple event parameter.

Apple Event Interprocess Messaging Protocol (AEIMP)  A standard defined by Apple Computer, Inc., for communication and data sharing among applications. High-level events that adhere to this protocol are called Apple events.

Apple event parameter  A keyword-specified descriptor record that contains data that the target application of an Apple event must use. Compared to attributes (which contain information that can be used by both the Apple Event Manager and the target application), parameters contain data used only by the target application of the Apple event. See also Apple event attribute, direct parameter, optional parameter, required parameter.

AppleScript Component  A scripting system integrated at the system level that utilizes the Component Manager. This technology, through the use of its scripting language, AppleScript, enables users to create scripts to control applications and the system.
**background**  A type of HyperCard object; a template shared by a number of cards. Each card with the same background has the same background picture, background fields, and background buttons in its background layer. Like other HyperCard objects, every background has a script. You can place handlers in a background script that you want to be accessible to all the cards with that background.

**background button**  A button that is common to all cards sharing a background. Compare with **card button**.

**background field**  A field that is common to all cards sharing a background; its size, position, and default text format remain constant on all cards associated with that background, but its text can change from card to card. Compare with **card field**.

**background picture**  The graphics in the background layer; the entire picture that is common to all cards sharing a background. You see the background picture by choosing Background from the Edit menu. Compare with **card picture**.

**Browse tool**  The tool you use to click buttons and to set the insertion point in fields.

**button**  A type of HyperCard object; a rectangular “hot spot” on a card or background that responds when you click it according to the instructions in its script. For example, clicking a right arrow button with the Browse tool can take you to the next card. See also **background button**, **card button**.

**Button tool**  The tool you use to create, change, and select buttons.

**card**  A type of HyperCard object; a rectangular area that can hold buttons, fields, and graphics. All cards in a stack are the same size. Each layer can contain its own buttons, fields, and graphics.

**card button**  A button that belongs to a card; it appears on, and its actions apply to, a single card. Compare with **background button**.

**card field**  A field that belongs to a card; its size, position, text attributes, and contents are limited to the card on which the field is created. Compare with **background field**.

**card picture**  A picture that belongs to and applies only to a specific card. Compare with **background picture**.

**chunk**  A piece of a character string represented as a chunk expression. Chunks can be specified as any combination of characters, words, items, or lines in a container or other source of value.

**chunk expression**  A HyperTalk description of a unique chunk of the contents of any container or other source of value.

**coercion handler**  A routine that coerces data from one Apple event descriptor type to another.

**command**  A response to a particular message; a command is a built-in message handler residing in HyperCard. See also **external command**.
Command key  The key at the lower-left side of the keyboard that has a propeller-shaped symbol. This key also has an Apple symbol and is sometimes called the Apple key.

comments  Descriptive lines of text in a script or program that are intended not as instructions for the computer but as explanations for people to read. Comments are set off from instructions by symbols called delimiters, which vary from language to language. In HyperTalk, two hyphens (--) indicate the beginning of a comment.

Component Manager  The Component Manager provides a database service that classifies software objects by function. In much the same way that the Resource Manager allows applications that are running to access data objects dynamically, the Component Manager provides services that allow run-time location of and access to functional objects.

constant  A named value that never changes. For example, the constant empty stands for the null string, a value that can also be represented by the literal expression "". Compare with variable.

container  A place where you can store a value (text or a number). Examples are fields, the Message box, the selection, and variables.

control structure  A block of HyperTalk statements defined with keywords that enable a script to control the order or conditions under which specific statements execute.

Core suite  The suite of core Apple event constructs that are common to all or nearly all applications. These definitions form the basic vocabulary for interapplication communication. Using only the constructs defined in the Core suite, applications can perform a wide range of useful tasks. This suite includes such events as Get Data, Set Data, and Count Elements. The Core suite of Apple events is described in the Apple Event Registry. Apple Computer, Inc., recommends that all applications support the core Apple events.

current  (adj.)  Applies to the card, background, or stack you’re using now. For example, the current card is the one you can see on your screen.

debug  To locate and correct an error or the cause of a problem or malfunction in a computer program, such as a HyperTalk script.

delimiter  A character or characters used to mark the beginning or end of a sequence of characters; that is, to define limits. For example, in HyperTalk double quotation marks act as delimiters for literals, and comments are set off with two hyphens at the beginning of the comment and a return character at the end.

descriptor  The combination of an object's generic name, immediately followed by its particular name, number, or ID number.

direct parameter  The parameter in an Apple event that contains the data to be used by the server application. For example, a list of documents to be opened is
specified in the direct parameter of the Open Documents event. See also Apple event parameter.

dynamic path  A series of extra objects inserted into the path through which a message passes when its static path does not include the current card. The dynamic path comprises the current card, current background, and current stack. Compare static path.

event handler  Any part of an application that deals with any event. Sometimes the term event handler is used to refer to any object that is eligible to handle menu commands.

expression  A description of how to get a value; a source of value or complex expression built from sources of value and operators.

external command (XCMD)  A command written in a computer language other than HyperTalk but made available to HyperCard to extend its built-in command set. External commands can be attached to a specific stack or to HyperCard itself. See also external function.

external function (XFCN)  A function written in a computer language other than HyperTalk but made available to HyperCard to extend its built-in function set. External functions can be attached to a specific stack or to HyperCard itself. See also external command.

factor  A single element of value in an expression. See also value. Factoring is the separation of the interface links to the application and the core functionality. By factoring an application, all features are accessed through event handlers via Apple events.

field  A type of HyperCard object; a container in which you type field text (as opposed to Paint text). HyperCard has two kinds of fields—card fields and background fields.

Field tool  The tool you use to create, change, and select fields.

formal parameters  See parameter variables.

function  A named value that HyperCard calculates each time it is used. The way in which the value is calculated is defined internally for HyperTalk’s built-in functions, and you can define your own functions with function handlers.

function call  The use of a function name in a HyperTalk statement or in the Message box, invoking either a function handler or a built-in function.

function handler  A handler that executes in response to a function call matching its name.

generic scripting component  A special scripting component that establishes connections dynamically with the appropriate scripting component for each script that a client application attempts to manipulate or execute.

global properties  The properties that determine aspects of the overall HyperCard environment. For example, userLevel is a global property that determines the current user level setting.
**global variable**  A variable that is valid for all handlers in which it is declared. You declare a global variable by preceding its name with the keyword `global`. Compare with **local variable**.

**handler**  A block of HyperTalk statements in the script of an object that executes in response to a message or a function call. The first line in a handler must begin with the word `on`, and the last line must end with the word `end`. Both `on` and `end` must be followed by the name of the message or function. HyperTalk has message handlers and function handlers.

**hierarchy**  See **message-passing hierarchy**.

**HyperTalk**  HyperCard’s built-in script language for HyperCard users.

**identifier**  A character string of any length, beginning with an alphabetic character; it can contain any alphanumeric character and the underscore character. Identifiers are used for variable and handler names.

**keyboard equivalent key**  A key you press together with the Command key to issue a menu command.

**keyword**  Any one of the 14 words that have a predefined meaning in HyperTalk. Examples of keywords are `on`, `if`, `do`, and `repeat`.

**layer**  The order of a button or field relative to other buttons or fields on the same card or background. The object created most recently is ordinarily the topmost object (that is, on the front layer).

**literal**  A string of characters intended to be taken literally. In HyperTalk, you use quotation marks (" ") as delimiters to set off a string of characters as a literal, such as the name of an object or a group of words you want to be treated as a text string.

**local variable**  A variable that is valid only within the handler in which it is used (local variables need not be declared). Contrast with **global variable**.

**loop**  A section of a handler that is repeated until a limit or condition is met, such as in a repeat structure.

**message**  A string of characters sent to an object from a script or the Message box, or that HyperCard sends in response to an event. Messages that come from the system—from events such as mouse clicks, keyboard actions, or menu commands—are called system messages. Examples of HyperTalk messages are `mouseUp`, `go`, and `push card`. See also **handler**.

**Message box**  A container that you use to send messages to objects or to evaluate expressions.

**message handler**  A handler that executes in response to a message matching its name.

**message-passing hierarchy**  The ordering of HyperCard objects that determines the path through which messages pass.

**metasymbol**  A word used in a syntax statement as a placeholder for an element that is different for each specific use of the statement. For example, the metasymbol `filename` is used to show where you put the name of a file you want a command to act on. In this book, metasymbols are shown as italics.
number  A character string consisting of any combination of the numerals 0 through 9, optionally including one period (.) representing a decimal value. A number can be preceded by a hyphen or a minus sign to represent a negative value.

object  An element of the HyperCard environment that has a script associated with it and that can send and receive messages. There are five kinds of HyperCard objects: buttons, fields, cards, backgrounds, and stacks.

object class  An application defines specific objects as distinct classes. In HyperCard, each of its objects (stack, background, card, button, and field) can be an individual class.

object descriptor  Designation used to refer to an object. An object descriptor is formed by combining the name of the type of object with a specific name, number, or ID number. For example, `background button 3` is an object descriptor. Stacks do not have a number or ID number, so only the name can be used for a stack descriptor.

object properties  The properties that determine how HyperCard objects look and act. For example, the location property of a button determines where it appears on the screen.

object specifier  A specific data type that contains references to specific classes (objects) and their relating specifiers (such as names, indexes, or IDs). For HyperCard, the metaphor that chunk expressions describe is an example of an object specifier.

online help  Assistance you can get from an application program while it's running. In HyperCard, online help refers to the HyperCard disk-based Help system.

Open Scripting Architecture (OSA)  A standard proposed by Apple Computer, Inc., to provide a uniform way for applications to provide or utilize scripting functionality.

operator  A character or group of characters that causes a particular calculation or comparison to occur. In HyperTalk, operators operate on values. For example, the plus sign (+) is an arithmetic operator that adds numerical values.

optional parameter  A supplemental parameter in an Apple event used to specify data that the server application should use in addition to the data specified in the direct parameter. Optional parameters are listed in the attribute identified by the keyOptionalKeywordAttr keyword. Applications use this attribute to specify or determine whether data exists in the form of optional parameters. Optional parameters need not be included in an Apple event; default values for optional parameters are part of the event definition. It is the responsibility of the server application that handles the event to supply values if optional parameters are omitted. See also Apple event attribute, Apple event parameter.

painting properties  The properties that control aspects of the HyperCard painting environment, which is invoked when
you choose a Paint tool. For example, the brush property determines the shape of the Brush tool.

**Paint text** Text you type using the Paint Text tool. Paint text can appear anywhere, while regular text must appear in a field created with the Field tool. Paint text is part of a card or background picture.

**Paint tool** Any HyperCard tool you use to make pictures. Paint tools include Lasso, Brush, Spray, Eraser, and many others.

**palette** A small window that displays icons or patterns you can select by clicking. You can see two of HyperCard's palettes, the Tools palette and the Patterns palette, simply by “tearing off” their respective menus. To see the Navigator palette, type palette "navigator" in the Message box. See also tear-off menu.

**parameters** Values passed to a handler by a message or function call. Any expressions after the first word in a message are evaluated to yield the parameters; the parameters to a function call are enclosed in parentheses, or, if there is only one, it can follow of.

**parameter variables** Local variables in a handler that receive the values of parameters passed with the message or function call initiating the handler's execution.

**picture** Any graphic or part of a graphic, created with a Paint tool or imported from an external file, that is part of a card or background.

**pixel** Short for “picture element”; the smallest dot you can draw on the screen. The position of the pointer is often represented by two numbers separated by commas. These numbers are horizontal and vertical distances of the pointer from the left and top edges of the card window, measured in pixels. The upper-left corner of the screen has the coordinates 0, 0.

**point** (1) A location on the screen described by two integers, separated by a comma, representing horizontal and vertical offsets, measured in pixels from the top-left corner of the card window or (in the case of the card window itself) of the screen. (2) In printing, the unit of measurement of the height of a text character; one point is about \( \frac{1}{72} \) of an inch. When you select a font, you can also select a point size, such as 10 point, 12 point, and so on.

**power key** One of a number of keys on the Macintosh keyboard you can press to initiate a menu action when a Paint tool is active. Power keys are enabled when you choose Power Keys from the Options menu or you check Power Keys on the Preferences card in the Home stack.

**properties** The defining characteristics of any HyperCard object and of HyperCard's environment. For example, setting the user level to Scripting changes the userLevel property of HyperCard to the value 5. Properties are often selected as options in dialog boxes or on palettes, or they can be set from handlers.

**Recent** A special dialog box that holds pictorial representations of the last 42 unique cards viewed. Choose Recent from
the Go menu to get the dialog box. Also, an adjective describing the card you were on immediately prior to the current card, as in recent card.

**recursion** The repetition of an operation or group of operations. Recursion occurs when a handler calls itself.

**regular text** Text you type in a field. You use the Browse tool to set an insertion point in a field and then type. Regular text is editable and searchable, while Paint text is not.

**required parameter** A keyword-specified descriptor record in an Apple event that must be specified. For example, a list of documents to open is a required parameter for the Open Documents event. **Direct parameters** are often required, and other additional parameters may be required. **Optional parameters** are never required.

**Required suite** The smallest of the standard Apple event suites, it includes definitions of four Apple events and four descriptor types. The Apple events defined in this suite, known as required Apple events, are sent to all applications that support high-level events and all applications that call the new Standard File routines under system software version 7.0 and later. All applications that support system software version 7.0 and later should support the Required suite. The events in this suite are the Open Application, Open Document, Print Document, and Quit Application events.

**resource fork** The part of a file that contains resources such as fonts, icons, and sounds, and so on.

**script** A collection of handlers written in HyperTalk and associated with a particular object. You use the script editor to add to and revise an object’s script. Every object has a script, even though some scripts are empty: that is, they contain nothing.

**scriptable** The capability of an application to respond to Apple events sent to it by a **scripting component**. To qualify as scriptable, an application is required to respond to appropriate standard Apple events and include an `aete` resource.

**script editor** A window in which you can type and edit a script. The title bar of the script editor describes the object to which the script belongs. You can use the Edit menu, the Script menu, and keyboard commands to edit text in the script editor. See also handler, object, and script.

**scripting component** A program that responds appropriately to calls made to the standard scripting component routines. Most scripting components implement scripting languages—for example, the AppleScript component implements the AppleScript scripting language.

**search path** When you open a file from within HyperCard, HyperCard attempts to locate the stack, document, or application you want by searching the folders listed on the appropriate Search Paths card in the Home stack. Each line on a Search Paths card indicates the location of a folder, including the disk name (and folder and subfolder names, if any). This information is called a search path. Items in a search path are separated by a colon, like this: my disk:HyperCard folder:my stacks:
Search Path cards  Three cards in the Home stack used to store information about the location of stacks, documents, and applications that you open while HyperCard is running. See also search path.

selection  A container that holds the currently selected area of text. Note that text found by the find command is not selected.

shared text  Field text that appears on every card in a background. Shared text can be edited only from the background layer. Text in shared fields cannot be searched.

source of value  HyperTalk’s most basic expressions; the language elements from which values can be derived: constants, containers, functions, literals, and properties.

stack  A type of HyperCard object that consists of a collection of cards; a HyperCard document.

statement  A line of HyperTalk code inside a handler. A handler can contain many statements. Statements within handlers are first sent as messages to the object containing the handler and then to succeeding objects in the message-passing hierarchy.

static path  The message-passing route defined by an object’s own hierarchy. For example, the static path followed by a message sent to (but not handled by) a button would include the card to which the button belongs, the background associated with that card, and the stack containing them. Compare dynamic path.

string  A sequence of characters. You can compare and combine strings in different ways by using operators. In HyperTalk, for example, 23 + 23 will result in 46; but 23 & 23 will result in 2323.

suite  A group of Apple event constructs that define an area of functionality. Suites provide the common vocabulary for applications. A suite can define constructs such as Apple events, Apple event object classes, descriptor types, key forms, comparison operators, or constants.

syntax  A description of the way in which language elements fit together to form meaningful phrases. A syntax statement for a command shows the command in its most generalized form, including placeholders (sometimes called metasymbols) for elements you must fill in as well as optional elements.

System file  Software a Macintosh computer uses to perform its basic operations.

system message  A message sent by HyperCard to an object in response to an event such as a mouse click, keyboard action, or menu command. Examples of HyperCard system messages are mouseUp, doMenu, and newCard.

target  The object that first receives a message.

tear-off menu  A menu that you can convert to a palette by dragging the pointer beyond the menu’s edge. HyperCard has two tear-off menus—Tools and Patterns. When torn off, these menus are referred to as palettes.
text field  See field.

text property  A quality or attribute of a character’s appearance. Text properties include style, font, and size.

tick  Approximately one-sixtieth (1/60) of a second. The wait command assumes a value in ticks unless you specify seconds by adding secs or seconds.

tool  An implement you use to do work. HyperCard has tools for browsing through cards and stacks, creating text fields, editing text, making buttons, and creating and editing pictures.

user level  A property of HyperCard, ranging from 1 to 5, that determines which of HyperCard’s capabilities are available. You can select the user level on the Preferences card in the Home stack. Each user level makes all the options from the lower levels available, and also gives you additional capabilities. The five user levels are Browsing, Typing, Painting, Authoring, and Scripting.

value  A piece of information on which HyperCard operates. All HyperCard values can be treated as strings of characters—they are not formally separated into types. For example, a numeral could be interpreted as a number or as text, depending on what you do with it in a HyperTalk handler.

variable  A named container that can hold a value consisting of a character string of any length. You can create a variable to hold some value (either numbers or text) simply by using its name with the put command and putting the value into it. HyperCard has local variables and global variables. Compare with constant.

window properties  The properties that determine how the Message box and the Tools and Patterns palettes are displayed. For example, the visible property determines whether or not the specified window is displayed on the screen.
Index

Symbols

<table>
<thead>
<tr>
<th>306–307, 307</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp; &amp; (ampersand, double) operator 113</td>
</tr>
<tr>
<td>&amp; (ampersand) operator 113</td>
</tr>
<tr>
<td>&amp; operator 118</td>
</tr>
<tr>
<td>( ) (parentheses) operator 114</td>
</tr>
<tr>
<td>() parentheses operator 114</td>
</tr>
<tr>
<td>* (asterisk) operator 114</td>
</tr>
<tr>
<td>(chunk) 120</td>
</tr>
<tr>
<td>-- (double hyphen) comment character 26, 112</td>
</tr>
<tr>
<td>- (minus sign) operator 114</td>
</tr>
<tr>
<td>(not equal sign) operator 115</td>
</tr>
<tr>
<td>/ (slash) operator 113</td>
</tr>
<tr>
<td>+ (plus sign) operator 115</td>
</tr>
<tr>
<td>&lt; (less than sign) operator 114</td>
</tr>
<tr>
<td>&lt;= (less than or equal to sign) operator 114</td>
</tr>
<tr>
<td>= (equal sign) operator 113</td>
</tr>
<tr>
<td>&gt; (greater than sign) 113</td>
</tr>
<tr>
<td>&gt;= (greater than or equal to sign) operator 114</td>
</tr>
<tr>
<td>^ (caret) operator 113</td>
</tr>
<tr>
<td>&amp; (not equal sign) operator 114</td>
</tr>
<tr>
<td>&lt;= (less than or equal to sign) operator 114</td>
</tr>
<tr>
<td>&gt;= (greater than or equal to sign) operator 114</td>
</tr>
</tbody>
</table>

\n
A

abbr date format 191
abbrev date format 191
abbreviated (adjective) 84
abbreviated date format 191
abbreviated time format 191
abbr time format 191
abbr time format 191
abs function 291
accessing XCMDs and XFCNs 504

char 120
actual parameters 78
add command 167–168
address property 17, 378
after (preposition) 251
all (preposition) 271
ambiguous stack descriptors 91
ampersand, double operator (&&) 113
ampersand operator (&) 113
and operator 115
annuity function 291–292
annuity. See also compound answer
It as destination 106
answer command 168–172
answer file command 168–172
answer for ask command 176
answer program command 15, 168–172
any (ordinal) 87
appleEvent message 20
appleEvent system message 132
AppleScript 5
and HyperTalk, comparing 6
application, stand-alone, building 14
arithmetic operators 113, 118
arrow cursor 395
arrowKey command 173–174
arrowKey system message 132, 173–174
ASCII codes 561–564
ask
It as destination 106
ask command 174–176
ask file command 174–176
ask password command 174–176
assigning menu names 93
asterisk (*) operator 114
atan function 292–293
autoHilite property 17, 379
autoSelect property 380–381
autoTab property 381–382
average function 293–294

B
background (object) 82
background button properties
  sharedHilite 468–469
background field properties
  sharedText 469–470
  showLines 470–471
background properties 360–361
  cantDelete 388–389
  cantModify 389–390
  dontSearch 398–399
  ID 416–418
  name 439–441
  number 441
  script 460–461
  scriptingLanguage 462–463
  showPict 472
backgrounds
  current 25
  defined 25
  descriptors for 82–87
beep command 177
before (preposition) 251
bgnd (object) 82
blindTyping property 382
bottom property 17, 383–384
bottomRight property 17, 384–385
brush property 386
brush tool name 178
btn (object) 82
bucket tool name 178
built-in functions 100
busy cursor 395
buttonCount palette property 237
button dialog modifications 7–8
Button Info dialog 7–8
button Info dialog box 35
button properties 365–367
  autoHilite 379
  bottom 383–384
  bottomRight 384–385
  enabled 402–403
  family 404–406
  height 411
  hilite 414–415
  icon 415–416
  ID 416–418
  left 421–422
  location 423–425
  name 439–441
  number 441
  partNumber 444
  rectangle 455–458
  right 459–460
  script 460–461
  scriptingLanguage 462–463
  sharedHilite 468–469
  showName 471
  style 475
  style 475
  textAlign 477
  textFont 477
  textHeight 480–481
  textSize 481–482
  textStyle 482–484
  titleWidth 486
  top 486–488
  topLeft 488–489
  version 496–497
  wideMargins 499
buttons
  as containers 105
  defined 24
  descriptors for 82–87
  editing scripts of 33–41
  messages to 58
  new features 8–12
  system messages and 126–128
button text
  text alignment 477
button tool name 178
C

cantAbort property 387–388

cantDelete property 388–389

cantModify property 389–390

cantPeek property 390–391

Can't understand error message 56

card (object) 82

card fields 82

Card Info dialog box 82

card properties 361–362

cantDelete 388–389

cantModify 389–390

dontSearch 398–399

ID 416–418

marked 432

name 439–441

number 441

owner 443

rectangle 455–458

right 460–461

scriptingLanguage 462–463

showPict 472

wideMargins 499

cards

current 25

defined 25

descriptors for 82–87

editing scripts of 34–41

system messages and 131–138

card window

current 96

card window properties

scroll 466–467

card windows 28

defined 28

cd (object) 82

centered property 391

characters as chunk expressions 120

charToNum function 294

checkMark property 392–393

checkpoints 43, 45, 196–197

choose command 178–179

chunk

defined 118

as a destination 123

chunk expression 118–124

ranges in 121

syntax of 119

clickChunk function 295–296

click command 180–181

location 180

clickH function 296, 299–300

clickLine function 296–297

clickLoc function 297–298

clickText function 298–299

closeBackground system message 133

closeCard system message 133

close command 15, 181–??

closeField system message 129

close file command 183–184

closePalette message 20

closePalette system message 133

closePicture message 20

closePicture system message 133

close printing command 185, 233

closeStack system message 133

close system message 132

close window command 186

closing external windows 540

cmdChar property 393–394

Command-hyphen 39

commandKeyDown command 187–188

commandKey function 300

commandKey system message 133

commands 165–287

add 167–168

answer 168–172

answer file 168–172

answer program 15, 168–172

arrowKey 173–174

ask 174–176

ask file 174–176

ask password 174–176

beep 177

choose 178–179

click 180–181

close 15, 181–??

635
INDEX

close file 183–184
close printing 185, 233
close window 186
commandKeyDown 187–188
controlKey 188–190
convert 3, 15, 191–194
create menu 194–195
create stack 195–196
debug checkpoint 196–197
defined 165
delete 15, 197–200, 575
dial 200–201
disable 15, 201–202
divide 202–203
doMenu 15, 203–205
drag 205–207
drag script 207
enable 15, 208
enterInField 209
enterKey 209–210
export paint 210–211
find 15, 211–214
find chars 15
find string 15
find whole 15
find word 15
functionKey 215–216
get 216–217
go 218–219
help 219
hide 220–222
import paint 223
keyDown 224–??
lock 225
lock error dialogs 15
lock recent 225
lock recent 15
lock screen 225
lock 15
mark 226–227, 228–229
open 229–231
open file 231–232
open printing 232–233
open report printing 234–235
open 15
overriding 142
palette 235–237
palette property 237
picture 15, 238–242
play 243–245
play stop 243–245
pop card 245–246
print 246–248
print card 248–250
push card 250–251
put 16, 251–254
read 254–256
read from file 16
reply 16, 256–??
request 258–260
request from 16
reset menubar 260–261
reset paint 261–262
reset printing 262
returnInField 262–263
returnKey 263
save stack 264
select 264–266
send ??–162
set 266–267
show 268–270
show cards 271–272
sort 272–274
sort 16–??
start using 274–275
stop using 275, 276
subtract 277
syntax notation 166–167
syntax summary 574–581	
tabKey 277–278
type 278–279
unlock 279–280
unlock error dialogs 279–280
unlock recent 279–280
unlock screen 279–280
unmark 281–282
visual 16, 282–284
wait 284–285

636
INDEX

write 285–287
write to file 16
comment character 112
comment character (--) 26
comment command 39
commenting scripts 39
comparison operators 113–118
complex expressions 111–112
compound function 301–302
compound function. See also annuity function constant, defined 99
constants 553–554
containers 103–109
  chunk expressions and 122–124
defined 103
fields 104
  Message box 108–109
  the selection 107
  variables 78, 105
contains operator 115
controlKey command 188–190
controlKey system message 133, 189
control structures 141–163
convert command 3, 15, 191–194
cos function 302
create menu command 194–195
create. See also disable, disabled, enable,
  enabled, text, textStyle, markChar, cmdChar, and put
create stack command 195–196
creating menus 194
cross cursor 395
current hierarchy 59–60
current objects 25
cursor property 394–395
cursors 395
curve tool name 178
cutCard system message 139

dateItems format 191
date function 302–304
dateItems format 191
debug checkpoint command 196–197
debugger 43–49
  command summary 49
defined 43
  exiting 45
  stepping through scripts 44
  tracing through scripts 45
debugger checkpoints 45
Debugger menu 43
debugger property 396
debugger tools 45
debugger windows 45–48
debugging environment 43
deleteBackground system message 133, 139
deleteButton system message 127
deleteCard system message 133, 139
delete command 15, 197–200
deleteFields system message 129
deleteStack system message 134, 139
destination function 16, 304
dial command 200–201
dialingTime property 17, 397
dialingVolume property 17, 398
dialog modifications
  button 7–8
  field 12
dialog window layer 544
disable command 15, 201–202
  . See also enable command
disabling background buttons 201–202
  disabling card buttons 201–202
disabling menu items 201–202
disabling menus 201–202
diskSpace function 16, 305
divide command 202–203
div operator 115
document window layer 544
do keyword 158–159
doMenu command 15, 203–205
  intercepting 166
  intercepting handler 204
  doMenu system message 134
dontSearch property 398–399

D
INDEX

dontWrap property 399–400
double hyphen (--) 26, 39, 40, 112
down constant 553
drag command 205–207
dragSpeed property 400–401
Draw Centered setting 373, 391
Draw Filled setting 373
Draw Multiple setting 373, 436
dynamic path 67–71
go command and 67
invoking 67
send keyword and 67, 69–??, 70, ??–71

drag command 205–207
dragSpeed property 400–401
Draw Centered setting 373, 391
Draw Filled setting 373
Draw Multiple setting 373, 436
dynamic path 67–71
go command and 67
invoking 67
send keyword and 67, 69–??, 70, ??–71

drag command 205–207
dragSpeed property 400–401
Draw Centered setting 373, 391
Draw Filled setting 373
Draw Multiple setting 373, 436
dynamic path 67–71
go command and 67
invoking 67
send keyword and 67, 69–??, 70, ??–71

drag command 205–207
dragSpeed property 400–401
Draw Centered setting 373, 391
Draw Filled setting 373
Draw Multiple setting 373, 436
dynamic path 67–71
go command and 67
invoking 67
send keyword and 67, 69–??, 70, ??–71

editBkgnd property 402
edit script command 207
eighth (ordinal) 84
else keyword 155–158
empty (constant) 105, 553
enable command 15, 208
enable property 18, 402–403
. See also disable command, enable command
debug trigger 26, 143, 154
debug trigger statement 154
debug trigger statement 147
enhancements
HyperTalk 15–21
enterInField command 209
enterInField system message 129
enterKey command 209–210
enterKey message 209, 210
enterKey system message 134
environmental properties 368
environmental property 18, 404
equal sign (=) 113
eraser tool name 178
example XCMD 545
exitField system message 129
exit keyword 72, 143, 153
export paint command 210–211
exportPaint command 210–211
expressions 99–109, 111–124
complex 111–118
exp2 function 307
external commands and functions 503–552
external window callbacks 522–529
external window events 536–540
xCloseEvt 537
xCursorWithin 539
xDebugErrorEvt 540
xDebugFinishedEvt 540
xEditClear 538
xEditCopy 538
xEditCut 538
xEditPaste 538
xEditUndo 538
xGetPropEvt 538
xGiveUpEditEvt 537
xGiveUpSoundEvt 538
xMBarClickedEvt 540
xMenuEvt 539
xOpenEvt 537
xScriptErrorEvt 540
xSendEvt 538
xSetPropEvt 538
xShowWatchInfoEvt 540
external windows 522–529, 534–545
closing 540
event handling 528

F

factors 111–112
false constant 554
family property 18, 404–406
field dialog modifications 12
Field Info dialog 12
field properties 362–364
<table>
<thead>
<tr>
<th>Index</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>diskSpace</td>
<td>16, 305</td>
</tr>
<tr>
<td>exp</td>
<td>306</td>
</tr>
<tr>
<td>exp1</td>
<td>306–307</td>
</tr>
<tr>
<td>exp2</td>
<td>307</td>
</tr>
<tr>
<td>foundChunk</td>
<td>307–308</td>
</tr>
<tr>
<td>foundField</td>
<td>308–309</td>
</tr>
<tr>
<td>foundLine</td>
<td>309–310</td>
</tr>
<tr>
<td>foundText</td>
<td>310</td>
</tr>
<tr>
<td>heapSpace</td>
<td>310–311</td>
</tr>
<tr>
<td>length</td>
<td>311–312</td>
</tr>
<tr>
<td>ln</td>
<td>312</td>
</tr>
<tr>
<td>ln1</td>
<td>313</td>
</tr>
<tr>
<td>log2</td>
<td>313</td>
</tr>
<tr>
<td>max</td>
<td>314</td>
</tr>
<tr>
<td>menus</td>
<td>315</td>
</tr>
<tr>
<td>min</td>
<td>315–316</td>
</tr>
<tr>
<td>mouse</td>
<td>316–317</td>
</tr>
<tr>
<td>mouseClick</td>
<td>317–318</td>
</tr>
<tr>
<td>mouseH</td>
<td>318</td>
</tr>
<tr>
<td>mouseLoc</td>
<td>319</td>
</tr>
<tr>
<td>mouseV</td>
<td>320</td>
</tr>
<tr>
<td>numToChar</td>
<td>322–323</td>
</tr>
<tr>
<td>offset</td>
<td>323–324</td>
</tr>
<tr>
<td>optionKey</td>
<td>324–325</td>
</tr>
<tr>
<td>param</td>
<td>325–326, ??–328</td>
</tr>
<tr>
<td>paramCount</td>
<td>326–327</td>
</tr>
<tr>
<td>parameters of</td>
<td>289</td>
</tr>
<tr>
<td>params</td>
<td>??–?</td>
</tr>
<tr>
<td>programs</td>
<td>16, 328</td>
</tr>
<tr>
<td>random</td>
<td>329</td>
</tr>
<tr>
<td>result</td>
<td>330–331</td>
</tr>
<tr>
<td>round</td>
<td>332</td>
</tr>
<tr>
<td>screenRect</td>
<td>333</td>
</tr>
<tr>
<td>seconds</td>
<td>333–334</td>
</tr>
<tr>
<td>selectedButton</td>
<td>17, 334–335</td>
</tr>
<tr>
<td>selectedChunk</td>
<td>335–336</td>
</tr>
<tr>
<td>selectedField</td>
<td>336–337</td>
</tr>
<tr>
<td>selectedLine</td>
<td>17, 337–339</td>
</tr>
<tr>
<td>selectedLoc</td>
<td>339–340</td>
</tr>
<tr>
<td>selectedText</td>
<td>17, 340–341</td>
</tr>
<tr>
<td>shiftKey</td>
<td>341–342</td>
</tr>
<tr>
<td>sin</td>
<td>343</td>
</tr>
<tr>
<td>sound</td>
<td>343–344</td>
</tr>
<tr>
<td>sqrt</td>
<td>345</td>
</tr>
<tr>
<td>stacks</td>
<td>345–??</td>
</tr>
<tr>
<td>stackSpace</td>
<td>346</td>
</tr>
<tr>
<td>sum</td>
<td>17, 346</td>
</tr>
<tr>
<td>syntax notation</td>
<td>290</td>
</tr>
<tr>
<td>systemVersion</td>
<td>17, 347</td>
</tr>
<tr>
<td>tan</td>
<td>347–348</td>
</tr>
<tr>
<td>target</td>
<td>348–349</td>
</tr>
<tr>
<td>ticks</td>
<td>349–350</td>
</tr>
<tr>
<td>time</td>
<td>350–351</td>
</tr>
<tr>
<td>tool</td>
<td>351–352</td>
</tr>
<tr>
<td>trunc</td>
<td>353–354</td>
</tr>
<tr>
<td>value</td>
<td>354–355</td>
</tr>
<tr>
<td>windows</td>
<td>355</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>G</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>get command</td>
<td>216–217</td>
</tr>
<tr>
<td>It as destination</td>
<td>106</td>
</tr>
<tr>
<td>global keyword</td>
<td>106, 159</td>
</tr>
<tr>
<td>global properties</td>
<td>369–372</td>
</tr>
<tr>
<td>address</td>
<td>378</td>
</tr>
<tr>
<td>blindTyping</td>
<td>382</td>
</tr>
<tr>
<td>cursor</td>
<td>394–395</td>
</tr>
<tr>
<td>debugger</td>
<td>396</td>
</tr>
<tr>
<td>dialingTime</td>
<td>397</td>
</tr>
<tr>
<td>dialingVolume</td>
<td>398</td>
</tr>
<tr>
<td>dragSpeed</td>
<td>400–401</td>
</tr>
<tr>
<td>editBkgrnd</td>
<td>402</td>
</tr>
<tr>
<td>environment</td>
<td>404</td>
</tr>
<tr>
<td>itemDelimiter</td>
<td>418–419</td>
</tr>
<tr>
<td>Language</td>
<td>420</td>
</tr>
<tr>
<td>lockErrorDialogs</td>
<td>425–426</td>
</tr>
<tr>
<td>lockMessages</td>
<td>426–427</td>
</tr>
<tr>
<td>lockRecent</td>
<td>427–428</td>
</tr>
<tr>
<td>lockScreen</td>
<td>428–429</td>
</tr>
<tr>
<td>longWindowTitle</td>
<td>430</td>
</tr>
<tr>
<td>messageWatcher</td>
<td>435</td>
</tr>
<tr>
<td>numberFormat</td>
<td>442–443</td>
</tr>
<tr>
<td>powerKeys</td>
<td>447</td>
</tr>
<tr>
<td>printMargin</td>
<td>448</td>
</tr>
<tr>
<td>printTextAlign</td>
<td>449</td>
</tr>
</tbody>
</table>
# INDEX

<table>
<thead>
<tr>
<th>printTextFont 450–??</th>
<th>hide (object) 221</th>
</tr>
</thead>
<tbody>
<tr>
<td>printTextHeight ??–450, 451</td>
<td>hide command 220–222</td>
</tr>
<tr>
<td>printTextStyle 452–453</td>
<td>hideIdle property 412</td>
</tr>
<tr>
<td>scriptEditor 461–462</td>
<td>hide menuBar system message 135</td>
</tr>
<tr>
<td>scriptTextFont 463–464</td>
<td>hideUnused property 413</td>
</tr>
<tr>
<td>scriptTextStyle 464–465</td>
<td>hiding card windows 220</td>
</tr>
<tr>
<td>stacksInUse 474</td>
<td>hiding objects 221–222</td>
</tr>
<tr>
<td>textArrows 478</td>
<td>hiding picture windows 220</td>
</tr>
<tr>
<td>textField 479–480</td>
<td>hiding stack windows 220</td>
</tr>
<tr>
<td>traceDelay 489–490</td>
<td>hierarchy, message-passing 56–70, 73–76</td>
</tr>
<tr>
<td>userLevel 490–491</td>
<td>current 56</td>
</tr>
<tr>
<td>userModify 491–492</td>
<td>defined 56</td>
</tr>
<tr>
<td>variableWatcher 492–493</td>
<td>objects in 58</td>
</tr>
<tr>
<td>global statement 159–160</td>
<td>hilitedButton palette property 237</td>
</tr>
<tr>
<td>global variables 47, 106, 159–160</td>
<td>hilite property 18, 414–415</td>
</tr>
<tr>
<td>go command 218–219</td>
<td>. See also autoHilite property, and family property, sharedHilite property</td>
</tr>
<tr>
<td>greater than (&gt;) operator 113</td>
<td>HyperCard enhancements since HyperCard 2.0 1–21</td>
</tr>
<tr>
<td>grid property 409</td>
<td>and Open Scripting Architecture 4–7</td>
</tr>
<tr>
<td>Grid setting 373</td>
<td>and other scripting systems 3–4</td>
</tr>
<tr>
<td>HyperCard properties</td>
<td>performance hints 555–560</td>
</tr>
<tr>
<td>ID 416–418</td>
<td>system requirements 1–2</td>
</tr>
<tr>
<td>version 494–495</td>
<td>and WorldScript compatibility 2, 3</td>
</tr>
<tr>
<td>HyperTalk, enhancements 15–21</td>
<td>hyphen (−) as minus arithmetic operator 112</td>
</tr>
</tbody>
</table>
| hyphen, double (--) comment character 112 | I, J

| I-beam cursor 395 | I, J
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>icon property 415–416</td>
<td>identifying a stack 90–91</td>
</tr>
<tr>
<td>identifying a stack 90–91</td>
<td>idle system message 52, 135</td>
</tr>
<tr>
<td>identifying a stack 90–91</td>
<td>ID property 18, 88, 416–418</td>
</tr>
<tr>
<td>idle system message 52, 135</td>
<td>if structure 155–158</td>
</tr>
<tr>
<td>if structure 155–158</td>
<td>multiple-statement 156–158</td>
</tr>
<tr>
<td>importing paint files 223</td>
<td>single-statement 155–156</td>
</tr>
<tr>
<td>importing paint files 223</td>
<td></td>
</tr>
</tbody>
</table>

641
import paint command 223
in (preposition) 92, 119, 166
Info menu 34
intercepting messages 76–77, 433
doMenu 433
interrupting executing handlers 72
into (preposition) 251
is an operator 115
is a operator 115
is in operator 115
is not an operator 115
is not a operator 115
is not in operator 115
is not operator 115
is operator 115
It (container) 106
itemDelimiter property 18, 418–419
items as chunk expressions 120
It variable 106

K

keyDown command 224–??
keyDown system message 135
keywords 53, 59, 141–163
   defined 141
do 158–159
else 155–158
end 143, 154
exit 143, 153
function 145, 146
global 159
next 154
on 142–143
pass 143
repeat 150
return 144
send 160

L

Language property 420
lasso tool name 178
last (ordinal) 87
layered buttons and fields 58
left property 421–422
length function 311–312
less than (<) operator 114
less than or equal to (<=) operator 114
lessthan or equal to (≤) operator 114
lineFeed constant 554
lines as chunk expressions 121
lineSize property 422–423
   line tool and 422
   shape tool and 422
line tool name 178
literal strings 100
ln1 function 313
ln function 312
local variables 47, 106
location 180
location property 423–425
   point 424
lock command 15, 225
lock error dialogs command 15
lockErrorDialogs property 19, 425–426
lockMessages property 426–427
lock recent command 15, 225
lockRecent property 427–428
lock screen command 225
lockScreen property 428–429
lockText property 19, 429–430
log2 function 313
long (adjective) 84
long date format 191
long time format 192
longWindowTitle property 430

M

markChar property 431
Creating 94
Defined 30
Disabling 201–202
Enabling 208
Number of 321
Referring to by number 93
Menus function 315
Message box 53, 108
Message handlers 26, 141–144
Example 144
Keywords in 142–145
Syntax of 141
Message name 77
Message-passing hierarchy
Current 59–60
User-defined 62–65
Using 73–76
Messages 51–79
AppleEvent 20
To a button 53, 58
To a card 52
closePalette 20
closePicture 20
Commands 26
From external commands 54
Handling 51
Intercepting 76
Keywords in 53
Matching message names 55–56
mouseDoubleClick 21
openPalette 21
openPicture 21
Receiving 55
Resulting from commands 54
Sending 52
Sent to a field 128–131
Sent to a locked field 53
Sent to buttons 126
Sent to current card 131–138
System 52
System messages in 56
To fields 58
Message sending order 26
Message Watcher 46–47

Index

mark command 226–227, 228–229
Marked property 432
Max function 314
Me (special object descriptor) 89
Menu
  disable command 202
  enable command 208
  Number of 320
Menu bar properties
  Rectangle 455–458
  Visible 496–497
Menu command 204
Menu commands 126
Menu item names 93, 94
Menu item properties 375–376
  CheckMark 392–393
  CheckMark. See also put cmdChar 393–394
  .See also menuMsg command; put command enabled 402–403
MenuChar 431
MenuMsg 433–434
Name 439–441
textStyle 484–485
textStyle. See also put
Menu items 31, 251
Adding messages for 251
Defined 31
Disabling 201–202
Enable 208
Referring to by number 94
MenuMessage property 434
Menu messages 31, 251
Defined 31
Intercepting 433
MenuMsg property 433–434
  .See also doMenu command; put command menu names 93, 94
Menu numbers 93
Menu properties
  enabled 402–403
  Name 439–441
Menus 30, 81
  Controlling through HyperTalk 93–95

643
<table>
<thead>
<tr>
<th>Index Entry</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>messageWatcher property</td>
<td>435</td>
</tr>
<tr>
<td>Message Watcher window properties</td>
<td>376-377</td>
</tr>
<tr>
<td>hideIdle</td>
<td>412</td>
</tr>
<tr>
<td>hideUnused</td>
<td>413</td>
</tr>
<tr>
<td>middle (ordinal)</td>
<td>87</td>
</tr>
<tr>
<td>min function</td>
<td>315-316</td>
</tr>
<tr>
<td>miniwindow layer</td>
<td>544</td>
</tr>
<tr>
<td>minus sign (−) operator</td>
<td>114</td>
</tr>
<tr>
<td>modems, dial command and</td>
<td>200-201</td>
</tr>
<tr>
<td>mod operator</td>
<td>116</td>
</tr>
<tr>
<td>mouseClicked function</td>
<td>317-318</td>
</tr>
<tr>
<td>mouseDoubleClick message</td>
<td>21</td>
</tr>
<tr>
<td>mouseDoubleClick system message</td>
<td>127, 129, 136</td>
</tr>
<tr>
<td>mouseDownInPicture system message</td>
<td>136</td>
</tr>
<tr>
<td>mouseDown message</td>
<td>53</td>
</tr>
<tr>
<td>mouseDown system message</td>
<td>127, 130, 136</td>
</tr>
<tr>
<td>mouseEnter message</td>
<td>52</td>
</tr>
<tr>
<td>mouseEnter system message</td>
<td>127, 130</td>
</tr>
<tr>
<td>mouse function</td>
<td>316-317</td>
</tr>
<tr>
<td>mouseH function</td>
<td>318</td>
</tr>
<tr>
<td>mouseLeave message</td>
<td>52</td>
</tr>
<tr>
<td>mouseLeave system message</td>
<td>127, 130</td>
</tr>
<tr>
<td>mouseLoc function</td>
<td>319</td>
</tr>
<tr>
<td>mouseStillDown system message</td>
<td>127, 130, 136</td>
</tr>
<tr>
<td>mouseUpInPicture system message</td>
<td>136</td>
</tr>
<tr>
<td>mouseUp message</td>
<td>53</td>
</tr>
<tr>
<td>mouseUp system message</td>
<td>128, 130, 136</td>
</tr>
<tr>
<td>mouseV function</td>
<td>320</td>
</tr>
<tr>
<td>mouseWithin message</td>
<td>52</td>
</tr>
<tr>
<td>mouseWithin system message</td>
<td>128, 130</td>
</tr>
<tr>
<td>moveWindow system message</td>
<td>136</td>
</tr>
<tr>
<td>multipleLines property</td>
<td>437-?</td>
</tr>
<tr>
<td>multiple property</td>
<td>436</td>
</tr>
<tr>
<td>multiSpace property</td>
<td>438-439</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index Entry</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>newBackground system message</td>
<td>137, 139</td>
</tr>
<tr>
<td>newButton system message</td>
<td>128</td>
</tr>
<tr>
<td>newCard system message</td>
<td>137, 139</td>
</tr>
<tr>
<td>newField system message</td>
<td>130</td>
</tr>
<tr>
<td>newStack system message</td>
<td>137, 139</td>
</tr>
<tr>
<td>next keyword</td>
<td>154</td>
</tr>
<tr>
<td>next repeat statement</td>
<td>154</td>
</tr>
<tr>
<td>next special object descriptor</td>
<td>89</td>
</tr>
<tr>
<td>nine (constant)</td>
<td>84</td>
</tr>
<tr>
<td>ninth (ordinal)</td>
<td>84</td>
</tr>
<tr>
<td>nonexistent chunks</td>
<td>124</td>
</tr>
<tr>
<td>not equal sign () operator</td>
<td>115</td>
</tr>
<tr>
<td>not equal sign (&lt;&gt;, ≠) operator</td>
<td>114</td>
</tr>
<tr>
<td>not operator</td>
<td>116</td>
</tr>
<tr>
<td>numberFormat property</td>
<td>442-443</td>
</tr>
<tr>
<td>number function</td>
<td>16, 320-322</td>
</tr>
<tr>
<td>number handling</td>
<td>103</td>
</tr>
<tr>
<td>number property</td>
<td>19, 441</td>
</tr>
<tr>
<td>number property. See also number function numbers 101–103</td>
<td></td>
</tr>
<tr>
<td>decimal string precision</td>
<td>102</td>
</tr>
<tr>
<td>numberFormat property and</td>
<td>102</td>
</tr>
<tr>
<td>SANE numeric values</td>
<td>102</td>
</tr>
<tr>
<td>. See also number property numToChar function</td>
<td>322-323</td>
</tr>
<tr>
<td>. See also charToNum function</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index Entry</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>88-89</td>
</tr>
<tr>
<td>object</td>
<td>26</td>
</tr>
<tr>
<td>object descriptors</td>
<td>82</td>
</tr>
<tr>
<td>combining</td>
<td>92</td>
</tr>
<tr>
<td>descriptor phasing</td>
<td>83</td>
</tr>
<tr>
<td>object hierarchy</td>
<td>56</td>
</tr>
<tr>
<td>message-passing</td>
<td>56-?</td>
</tr>
<tr>
<td>message-passing in</td>
<td>?-78</td>
</tr>
<tr>
<td>object ID number</td>
<td>88-89</td>
</tr>
<tr>
<td>object names</td>
<td>83</td>
</tr>
<tr>
<td>object numbers</td>
<td>84</td>
</tr>
<tr>
<td>integer</td>
<td>84</td>
</tr>
</tbody>
</table>
numeric constants 84
ordinal constants 84
reassigning 87
special ordinals 87
tab order 87
object properties 358
ID 88
name 84
objects 24, 81
background buttons 82
backgrounds 25
buttons 24
cards 25
fields 24, 81
generic names 82
referring directly to 92
stacks 25

Objects menu 33
of (preposition) 92, 119, 166
offset function 323–324
one (constant) 84
on keyword 26, 142–143
open
  document with application 229
openBackground system message 137
openCard message 52
openCard system message 137
open command 15, 229–231
openField message 53
openField system message 130
open file command 231–232
opening
  documents 231
  files for reading or writing 232
openPalette message 21
openPalette system message 137
openPicture message 21
openPicture system message 137
open printing command 232–233
open report printing command 234–235
Open Scripting Architecture (OSA) 4–7
  and AppleScript 5
openStack system message 137
operator precedence 565
operators 113–118
  ampersand (\&) 113
  ampersand, double (\&\&) 113
  and 115
  asterisk (*) 114
caret (^) 113
  comparison 113
  contains 115
div 115
equal sign (=) 113
  and expression type 118
greater than (>) 113
greater than or equal to sign (\geq) 114
is 115
  is a 115
  is in 115
  is not 115
  is not a 115
  is not an 115
  is not in 115
  less than or equal to sign (\leq) 114
  less than sign (<) 114
  minus sign (-) 114
  mod 116
  not 116
  not equal sign () 115
  not equal sign (<>, \neq) 114
numeric values 118
  or 116
  parentheses () 114
  plus sign (+) 115
  precedence of 117–118
  slash (/) 113
  there is a 116
  there is an 116
  there is not a 116
  there is not an 116
  within 116
optionKey function 324–325
  See also commandKey function
ordinal constant 119
or operator 116
oval tool name 178
overriding commands 142
INDEX

owner of card property 19
owner of window property 19
owner property 443

P

painting properties 372–373
  brush 386
  centered 391
  filled 406
  grid 409
  lineSize 422–423
  multiple 436
  multiSpace 438–439
  pattern 445–446
  polySides 446
  textAlign 477
  textFont 479–480
  textHeight 480–481
  textSize 481–482
  textStyle 482–484
Paint text 104, 477
Paint Text tool 477
palette command 235–237
palette properties
  properties 237
  palette properties
    buttonCount 237
    commands 237
    hilitedButton 237
paramCount function 326–327
parameter list 78, 142
parameter passing 78
to handlers 78
parameters
  defined 77
    in function handlers 78
  parameter variables 78
param function 325–326, ??–328
param. See also paramCount, params
params function 327–??
partNumber property 19, 444

part properties
  scriptingLanguage 462–463
pass keyword 55, 67, 143
pass statement 147
pasteCard system message 139
pattern property 445–446
Patterns palette 445
pencil tool name 178
pi constant 554
picture command 15, 238–242
  globalRect 239, 240
  loc 240
  rect 239
  scale 241
  scroll 240
picture window properties
  rect 454–455
  scroll 466–467
play command 243–245
playing notes 243
playing sound 244
play stop command 243–245
plus cursor 395
plus sign (+) operator 115
polygon tool name 178
polySides property 446
pop card command 245–246
  . See also push card command
pop-up button properties, titleWidth 486
powerKeys property 447
previous special object descriptor 89
print
  marked cards 248
print card command 248–250
print command 246–248
  document with application 247
  expression 247
  field 247
printing cards 248–250
printing fields 246–248
printing marked cards 248–250
printing reports 235
printMargin property 448
printTextAlign property 449
INDEX

printTextFont property 450–??
printTextHeight property ??–450, 451
printTextSize property 452–453
printTextStyle property 453
programs function 16, 328
properties 357–499
  20
  address 17, 378
  autoHilite 17, 379
  autoSelect 380–381
  autoTab 381–382
  blindTyping 382
  bottom 17, 383–384
  bottomRight 17, 384–385
  brush 386
  buttonCount 237
  cantAbort 387–388
  cantDelete 388–389
  cantModify 389–390
  cantPeek 390–391
  centered 391
  checkMark 392–393
  cmdChar 393–394
  commands 237
  cursor 394–395
  debugger 396
  defined 101
  dialingTime 17, 397
  dialingVolume 17, 398
  dontSearch 398–399
  dontWrap 399–400
  dragSpeed 400–401
  editBkgnd 402
  enabled 18, 402–403
  environment 18, 404
  family 18, 404–406
  filled 406
  fixedLineHeight 407–408
  freeSize 408–409
  grid 409
  hBarLoc 410
  height 18, 411
  hideIdle 412
  hideUnused 413
  hilite 18, 414–415
  hilitedButton 237
  icon 415–416
  ID 18, 416–418
  itemDelimiter 18, 418–419
  Language 420
  left 421–422
  lineSize 422–423
  location 423–425
  lockErrorDialogs 19, 425–426
  lockMessages 426–427
  lockRecent 427–428
  lockScreen 428–429
  lockText 19, 429–430
  longWindowTitle 430
  markChar 431
  marked 432
  menuMsg 433–434
  messageWatcher 435
  multiple 436
  multipleLines 437–??
  multiSpace 438–439
  name 19, 439–441
  number 19, 441
  numberFormat 442–443
  owner 443
  owner of card 19
  owner of window 19
  partNumber 19, 444
  pattern 445–446
  polySides 446
  powerKeys 447
  printMargin 448
  printTextAlign 449
  printTextFont 450–??
  printTextHeight ??–450, 451
  printTextSize 452–453
  printTextStyle 453
  properties 237
  rect 19, 454–455
  rectangle 455–458
  reportTemplates 458–459
  right 19, 459–460
  script 460–461
scriptEditor 461–462
scriptingLanguage 19, 462–463
scriptTextFont 463–464
scriptTextSize 464–465
scroll (fields) 465–466
scroll (windows) 466–467
sharedHillite 468–469
sharedText 469–470
showLines 470–471
showName 471
showPict 472
size 473–474
stacksInUse 474
style 475
style of button 20
textAlign 477
textArrows 478
textFont 479–480
textHeight 480–481
textSize 481–482
textStyle (buttons, fields, painting environment) 482–484
textStyle (menu items) 484–485
titleWidth 20, 486
top 486–488
topleft 20, 488–489
traceDelay 489–490
userLevel 490–491
userModify 491–492
variableWatcher 492–493
vBarLoc 493–494
version 494–495
visible 20, 496–497
wideMargins 498
width 20, 499
properties syntax 377
property name 101
Protect Stack dialog box 389, 390
push card command 250–251
put command 16, 124, 251–254
See also cmdChar property; enabled property; menuMsg property; text property; and textStyle property

Q
quit system message 137
quote constant 554

R
random function 329
ranges of chunks 121
read command 106, 254–256
It as destination 106, 255
limits 255
until character 255
read from file command 16
reading files 254–256
reassigning object numbers 87
recent special object descriptor 89
rectangle 364
rectangle properties 367
  bottom 383–384
  bottomRight 384–385
  constant 488
  height 411
  left 421–422
  right 459–460
top 486–488
topleft 488–489
rectangle property 455–458
rectangle tool name 178
rect property 19, 454–455
recursion 72
redefining
  commands 165–166
redefining commands 165–166
  hints 166
redefining functions 290
referring to card windows 96
referring to external windows 97
referring to fields 104
referring to menu items 94
referring to menus 93
referring to windows 97
INDEX

repeat forever statement 150
repeat for statement 151
repeat keyword 150
repeat statement 150–153
  forms of 150–153
repeat for 151
repeat forever 150
repeat until 151
repeat while 152
repeat with 152–153
repeat with...down to 152–153
repeat with...to 152
repeat statements ??–154
repeat structure 149–154
repeat until statement 151
repeat while statement 152
repeat with...down to statement 152–153
repeat with statement 152–153
repeat with...to statement 152
reply command 16, 256–257
reportTemplates property 458–459
request command 258–260
request from command 16
reserved words (keywords) 141–163
reset menubar command 260–261
reset paint command 261–262
reset printing command 262
result function 330–331
resumeStack system message 137
resume system message 137, 139
retrieving properties 357–376
return, soft 41
return constant 554
returnInField command 262–263
returnInField system message 131
returnKey command 263
returnKey system message 137
return keyword 144
return statement 148
right property 19, 459–460
round function 332
round rectangle tool name 178

S

save stack command 264
screen rectangles 367, 457
screenRect function 333
script attachability 6–7
script comments 26
script debugger 43
script editor 35–42
  automatic formatting 40
  breaking long line statements 41
  command summary 41
  comments 40
  enhancements 7
  formatting scripts 40
  manipulating text 37, 38
  opening multiple scripts 35
  replacing text 39
  saving a script 35
  script size 41
  searching 38
  shortcuts 35
scriptEditor property 461–462
scriptingLanguage property 19, 462–463
scripting systems, other 3–4
scripting userLevel 34
script property 460–461
scripts 26
  attachability 6–7
  background, editing the script of 34
  background buttons 35
  background fields 35
  button, editing the script of 35
  closing 35
  current card, editing the script of 34
  fields, editing the script of 35
  function handlers within 27
  getting to object scripts 33
  opening 33–35
  saving 35
  scripting shortcuts 35
  shortcuts for opening 35
  stack, editing the script of 34
scriptTextFont property 463–464

649
INDEX

scriptTextSize property 464–465
scroll (fields) property 465–466
scroll (windows) property 466–467
searching, find command 212
second (ordinal) 84
seconds format 192
seconds function 333–334
seco...
INDEX

naming 91
referring to 90
stack builder, integrated stand-alone 14
stack name 90
stack properties 359–360
cantAbort 387–388
cantDelete 388–389
cantModify 389–390, 390–391
freeSize 408–409
name 439–441
reportTemplates 458–459
script 460–461
scriptingLanguage 462–463
size 473–474
version 494–495
stacks
current 25
defined 25
descriptors for 90, 91
stacks function 345–72
stacksInUse property 474
stackSpace function 346
stack window properties
rect 454–455
scroll 466–467
stand-alone application, building 14
startUp system message 138, 139
start using command 274–275
statements 26
defined 26
end 147
end repeat 154
exit 147
exit repeat 153
formatting 40
global 159–160
as messages 53
next repeat 154
pass 147
repeat 150–154
return 148
send 160–163
static path 67
stepping through scripts 44
stop using command 275, 276
structures
if 155–158
multiple-statement 156–158
single-statement 155–156
style of button property 20
style property 475
subroutine calls 71–72
calling handler 72
subroutine handler 72
subtract command 277
sum function 17, 346
suspendStack system message 138
suspend system message 138
syntax
notation for commands 166–167
notation for functions 290
summary for commands 574–581
summary for functions 581–587
system message order 138–139
system messages 52, 125–139
appleEvent 132
arrowKey 132
close 132
closeBackground 133
closeCard 133
closeField 129
closePalette 133
closePicture 133
closeStack 133
commandKey 133
contrlKey 133
cutCard 139
deleteBackground 133, 139
deleteButton 127
deleteCard 133, 139
deleteField 129
deleteStack 134, 139
doMenu 134
doInField 129
doKey 134
entry point in hierarchy 58
exitField 129
functionKey 134
help 134
hide menuBar 135
idle 135
is an 115
keyDown 135
mouseDoubleClick 127, 129, 136
mouseDown 127, 130, 136
mouseDownInPicture 136
mouseEnter 127, 130
mouseLeave 127, 130
mouseStillDown 127, 130, 136
mouseUp 128, 130, 136
mouseUpInPicture 136
mouseWithin 128, 130
moveWindow 136
newBackground 137, 139
newButton 128
newCard 137, 139
newField 130
newStack 137, 139
openBackground 137
openCard 137
openField 130
openPalette 137
openPicture 137
openStack 137
pasteCard 139
quit 137
resume 137, 139
resumeStack 137
returnInField 131
returnKey 137
show 137
sizeWindow 138
startUp 138, 139
suspend 138
suspendStack 138
tabKey 131, 138
systemVersion function 17, 347

T

tab constant 554
tabKey command 277–278
tabKey. See also tabKey system message
tabKey system message 131, 138
tab order 87
tan function 347–348
target function 61, 348–349
temporary checkpoints 45
ten (constant) 84
tenth (ordinal) 84
textAlign property 477
textArrows property 478
textFont property 479–480
textHeight property 480–481
text operators 118
textSize property 481–482
textStyle property (buttons, fields, painting
environment) 482–484
textStyle property (menu items) 484–485
text tool name 178
there is an operator 116
there is a operator 116
there is not an operator 116
there is not a operator 116
the result function 144
the selection container 107
third (ordinal) 84
This 561
three (constant) 84
ticks function 349–350
time function 350–351
titleWidth property 20, 486
to (preposition) 121
tool function 351–352
. See also choose command
topleft property 20, 488–489
top property 20, 486–488
traceDelay property 489–490
tracing through scripts 45
true constant 554
trunc function 353–354
two (constant) 84
INDEX

type command 278–279

U
unlock command 279–280
unlock error dialogs command 279–280
unlock recent command 279–280
unlock screen
  with visual effect 280
unlock screen command 279–280
up constant 554
user-defined (custom) menus 93
user-defined message-passing hierarchy 62–65
  adding stacks 275
  deleting stacks 276
  handlers in 65
userLevel property 490–491
userModify property 491–492

V
value function 354–355
values 99–109
variable name 105
variables 105–106
  as containers 105
  defined 105
  global 106
  It 106
  local 106
  and numbers 102
  parameter variables 78, 106
  values stored in 102
Variable Watcher 47–48
Variable Watcher properties 376–377
variableWatcher property 492–493
Variable Watcher window properties
  hBarLoc 410
  rect 454–455
  vBarLoc 493–494
  vBarLoc property 493–494
  version property 494–495
  visible property 20, 496–497
  visual command 16, 282–284
  visual effects 283
  vocabulary list 589–621

W
wait command 284–285
watch cursor 395
wideMargins property 498
width property 20, 499
window layers defined 543
window properties 374–375
  bottom 383–384
  bottomRight 384–385
  height 411
  ID 416–418
  left 421–422
  location 423–425
  owner 443
  rectangle 455–458
  right 459–460
  top 486–488
  topleft 488–489
  visible 496–497
windows 28, 81
windows function 355
within operator 116
words as chunk expressions 120
write command 285–287
  See also close file command; open file command; read command
write to file command 16

X, Y
XCmdBlock parameter block 509
XCMDs and XFCNs 503
  accessing 504
attaching the resource 508
callback fields 511
callback procedures and functions 512–534
  AbortScript 533
  BeginXSound 520
  BeginXWEdit 529
  BoolToStr 515
  CloseXWindow 524
  CountHandlers 532
  EndXSound 520
  EndXWEdit 530
  EvalExpr 513
  ExtToStr 515
  FormatScript 530
  FrontDocWindow 520
  GetCheckPoints 531
  GetFieldByID 518
  GetFieldByName 518
  GetFieldByNum 518
  GetFieldTE 519
  GetFilePath 521
  GetGlobal 514
  GetHandlerInfo 533
  GetNewXWindow 522
  GetObjectName 532
  GetObjectScript 532
  GetStackCrawl 534
  Get_VarValue 533
 GetXResInfo 521
  GoScript 534
  HCWordBreakProc 530
  HideHCPalettes 525
  LongToStr 516
  NewXWindow 523
  Notify 521
  NumToHex 516
  NumToStr 516
  PasToZero 516
  PointToStr 516
  PrintTEHandle 530
  RectToStr 516
  RegisterXWMenu 525
  ReturnToPas 517
  RunHandler 513
  SaveXWScript 531
  ScanToReturn 514
  ScanToZero 514
  SendCardMessage 513
  SendHCEvent 521
  SendHCMassage 513
  SendWindowMessage 522
  SetCheckPoints 532
  SetFieldByID 519
  SetFieldByName 519
  SetFieldByNum 519
  SetFieldTE 519
  SetGlobal 514
  SetObjectScript 532
  SetVarValue 533
  SetXWIdleTime 526
  ShowHCPalettes 526
  StackNameToNum 522
  StepScript 534
  StringEqual 515
  StringLength 515
  StringMatch 515
  StrToBool 517
  StrToExt 517
  StrToLong 517
  StrToNum 517
  StrToPoint 517
  StrToRect 517
  TraceScript 534
  XWAllowReEntrancy 528
  XWAlwaysMoveHigh 527
  XWHasInterruptCode 526
  ZeroBytes 514
  ZeroTermHandle 515
  ZeroToPas 518
debugger callbacks 533
entryPoint 511
external window callbacks 522–529
field callbacks 518–519
guidelines for writing 507
HyperTalk callbacks 513
inArgs 512
invoking 505
maximum parameters 505
INDEX

memory callbacks 514
in the message-passing hierarchy 505
miscellaneous utility callbacks 520–522
outArgs 512
paramCount field 510
parameter block 509
params array 510
passFlag 511
passing back results 510
passing information 509
passing parameters 510
request 511
result 511
returnValue 510
script editor callbacks 530–532
special XCmdBlock values 540
  Debugger 542
  Message Watcher 541
  script editor 541
  Variable Watcher 541
string callbacks 514–515
string conversion callbacks 515–518
text editing callbacks 524–529, 529–530
uses for 504
  Variable Watcher callbacks 533
XCmdBlock 509
XTalkObject fields
  bkgndID 542
  buttonID 543
  cardID 543
  fieldID 543
  objectKind 542
  stackNum 542
XTalkObject structure 542–543

Z

zero..ten constant 554
This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Proof pages were created on an Apple LaserWriter IIINTX printer. Final page negatives were output directly from the text and graphic files. Line art was created using Adobe™ Illustrator. PostScript™, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier.

WRITERS
Julie Callahan, Cheryl Chambers, Steve Schwander, and Alan Spragens

DEVELOPMENTAL EDITORS
Jeanne Woodward and Beverly Zegarski

ILLUSTRATOR
Barbara Carey

PRODUCTION EDITOR
Rex Wolf

COVER DESIGNER
Barbara Smyth

PROJECT MANAGER
Patricia Eastman