HyperTalk™ Beginner's Guide for the Macintosh

Scripting

Limited Warranty on Media and Replacement

If you discover physical defects in the manuals distributed with an Apple product or in the media on which a software product is distributed, Apple will replace the media or manuals at no charge to you, provided you return the item to be replaced with proof of purchase to Apple or an authorized Apple dealer during the 12-month period after you purchased the software. In addition, Apple will replace damaged software media and manuals for as long as the software product is included in Apple's Media Exchange Program. While not an upgrade or update method, this program offers additional protection for two years or more from the date of your original purchase. See your authorized Apple dealer for program coverage and details. In some countries the replacement period may be different; check with your authorized Apple dealer.

ALL IMPLIED WARRANTIES ON THE MEDIA AND MANUALS, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO the Twelve (12) month period FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though Apple has tested the software and reviewed the documentation, **APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESSED OR IMPLIED, WITH RESPECT TO SOFTWARE, ITS QUALITY, PERFORMANCE, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS SOFTWARE IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND PERFORMANCE.**

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT IN THE SOFTWARE OR ITS DOCUMENTATION, even if advised of the possibility of such damages. In particular, Apple shall have no liability for any programs or data stored in or used with Apple products, including the costs of recovering such programs or data.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESSED OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

HyperTalk™ Beginner's Guide for the Macintosh

Scripting

Contents

About This Book

This book shows you how to start using HyperTalk™, the language that's built into HyperCard™. With HyperTalk, you can write your own instructions, called *scripts*, for HyperCard to carry out. Writing scripts is called *scripting*.

You can create, customize, and personalize HyperCard stacks without learning how to write scripts; but scripting with HyperTalk gives you even more control over your computer.

If writing scripts sounds a lot like programming to you, you're right—they are very similar. However, you do not need *any* previous experience with programming to write scripts. If you can read this paragraph, you can write a script.

What you need to know to use this book

To get the most out of this book, you should already know the basics of using a Macintosh™ computer; for instance, how to use the mouse, menus, and icons on the screen. You should also know how to find your way around in a HyperCard stack. If you have gone through the first five or six chapters of *Getting Started With HyperCard,* you probably know enough to begin.

Specifically, you should know how to use buttons to get around in stacks and how to use the HyperCard menus and tools. You should have browsed through some stacks, looked through part of the HyperCard Help stack, and perhaps started using a stack for your own purposes—for example, you might have used the Addresses stack to store some information.

If you already have experience with programming in another language, you might want to go directly to the *HyperCard Script Language Guide*, published by Addison-Wesley.

This book is intended to help you get started and let you get a feel for scripting on your own. You won't find long, technical explanations of HyperTalk concepts here, but you will be able to see clearly how specific scripts work.

How to use this book

The chapters in this book include exercises made up of numbered steps. Each step consists of a short instruction in boldface type followed (usually) by further explanation in plain type. Depending on your level of expertise with HyperCard, you may find that you can speed through some of the exercises by reading just the boldface steps. You can stop and read the more detailed explanations in plain type whenever you need to.

Each chapter builds on what you've done in previous chapters, so it's important that you start with Chapter 1 and work through the book sequentially.

- In Chapter 1, "Getting Started," you'll create a practice stack, which you'll use for scripting throughout this book. You'll make some buttons for the stack and complete their scripts.
- In Chapter 2, "Fields and Other Containers," you'll write some simple scripts that explore the way HyperCard stores and retrieves information.
- In Chapter 3, "Scripts That Make Decisions," you'll write some more powerful scripts.
- In Chapter 4, "Handling Messages," you'll explore how buttons and other objects receive and send messages.
- In Chapter 5, "More Scripting Ideas," you'll look at other ways you can use scripts in stacks. You'll see how to create a presentation stack, animation, and a stack just for fun.
- The Appendix, "HyperTalk Summary," contains a complete list of HyperTalk commands, functions, and other elements.

Terms appear in *italics* in this book when they are defined. These words are also defined in the glossary.

You'll also find an index and a HyperTalk Quick Reference Card, which you can remove from this book and keep handy.

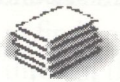
At the end of the book is a Tell Apple card. By answering the questions and mailing the card to Apple, you help us improve our products and documentation. Fill the card out after you've worked with this book.

Other sources of information

Because this book is intended as an introduction to scripting for beginners, it is not comprehensive. HyperTalk comprises many commands, functions, keywords, and other elements that are not explained in this book.

The HyperCard package includes the following reference materials:

HyperTalk Reference: A stack that provides easy access to information about HyperTalk. You will find this stack indispensable as you begin to learn scripting.



HyperCard Help: A stack that answers your questions about HyperCard's menus and tools.

HyperCard Reference: A book that contains reference information about all aspects of HyperCard other than scripting.

You may also want to consult the following books, which were written at Apple and are published by Addison-Wesley as part of the Apple Technical Library:

HyperCard Stack Design Guidelines: A book that provides information about how to design and build stacks. Its focus is the look and behavior of stacks (for example, navigation methods and card layouts) rather than the mechanics of scripts.

HyperCard Script Language Guide: A book that provides detailed reference information about scripts and HyperTalk. This book is for people with some programming or scripting experience.

Several other excellent books have been written about HyperTalk. Check with your favorite bookseller to see what titles are currently available.

Getting Started

Have you ever wished your computer could do things your way? Most application programs are designed to perform one type of task, like word processing or creating graphics. But what about all the things you do that don't fit neatly into other people's categories?

HyperCard™ lets you create your own ways of doing things on your computer. If you have read *Getting Started With HyperCard*, you already know how to work with HyperCard tools such as the Button and Field tools. This book introduces you to *scripting*—writing sets of instructions called *scripts* that give you even more control over the way HyperCard stacks work.

HyperCard scripts are written in the HyperTalk™ language, which is similar to English in many ways. HyperTalk uses common words such as *go*, *put*, and *it* in much the same way that people use these words in everyday life. In this book you'll learn how to combine these words with other words to form instructions that HyperCard can understand. As you work through the book, you'll build your vocabulary of HyperTalk words. You'll also learn how to write larger, more powerful sets of instructions.

You do not need any prior experience with computer languages to use this book. You should, however, know how to get around in HyperCard stacks, and how to use some of the tools described in *Getting Started With HyperCard.*

What you will build

In this book you'll learn scripting by building a practice stack from scratch and writing scripts for it. If you work through Chapters 1–4 in order, you'll end up with a stack you can use to catalog a collection of record albums, cassettes, or compact disks. Figure 1-1 shows a sample card from a completed version of the practice stack.

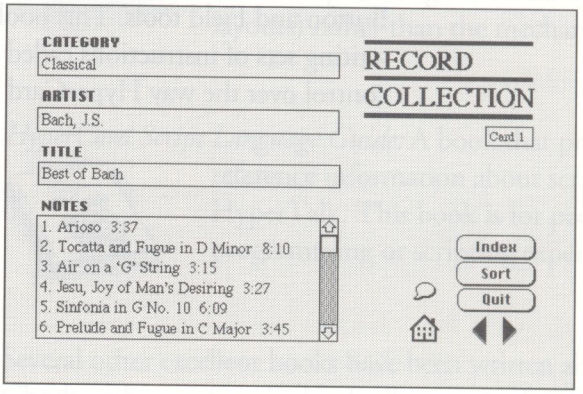


Figure 1-1 Sample card from the practice stack

If you don't feel like cataloging a collection of recordings, don't worry. In Chapter 5 you'll learn how to modify the practice stack for other purposes. You can modify it to keep track of books, baseball cards, computer software, your favorite restaurants, the inventory for a business, or anything else you might want to catalog. Here are some examples:

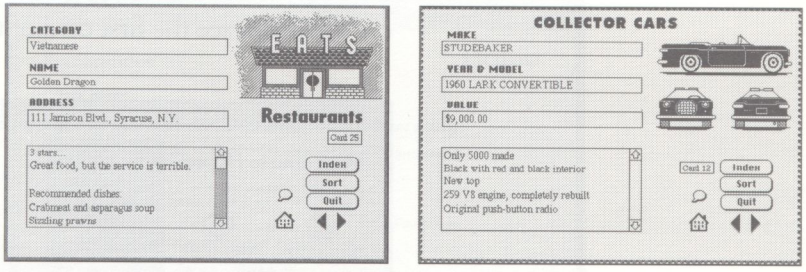


Figure 1-2 Some variations on the practice stack

More importantly, you'll learn basic scripting concepts and techniques as you build the practice stack. By the time you're finished with this guide, you'll know enough to create stacks for your own purposes.

Each chapter in this book builds on material you've completed in previous chapters, so you should go through the chapters sequentially. In this chapter you'll create the practice stack and write some simple scripts to control the actions of buttons.

Starting up HyperCard

This book is meant to be used with HyperCard "up and running" on your Macintosh™ computer. You'll need to perform the steps as directed in the sections that follow to get the most out of the material.

First, start up HyperCard as you normally would. (The *HyperCard Reference* includes instructions if you need them.) If you already have HyperCard running, go to the Home stack. You're ready to go on when you see the first card of the Home stack on your screen.

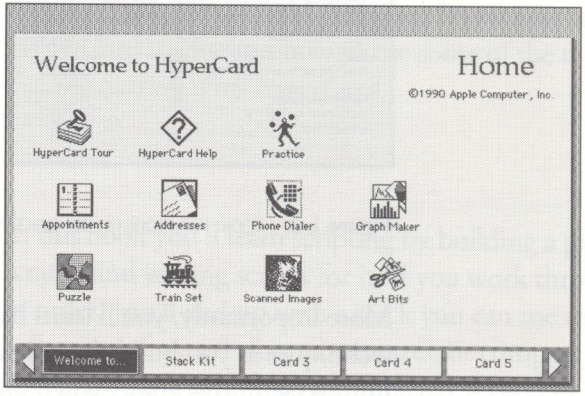


Figure 1-3 The first card of the Home stack

Setting your user level

To work with scripts, your user level must be set to Scripting. Start from the first card of the Home stack and set your user level as described in the following steps:

1. **Click the left arrow button.**

   Or choose Preferences from the Home menu. The Preferences card appears.

2. **Click the Scripting button on the Preferences card.**

   For now, leave the check box options Blind Typing, Power Keys, and Arrow Keys in Text unchecked. (For more information about these options, see the *HyperCard Reference*.)

   Figure 1-4 shows the Preferences card with Scripting selected.

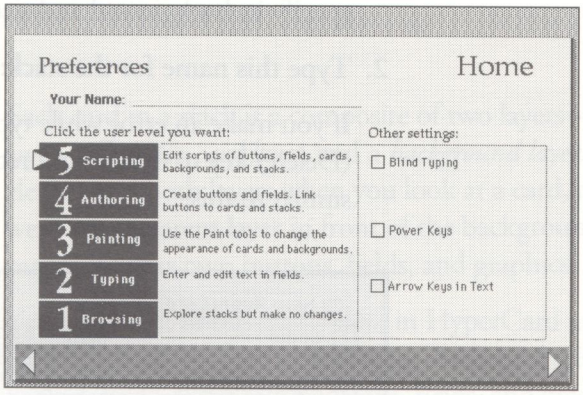


Figure 1-4 The Preferences card of the Home stack

When the user level is set to Authoring or Scripting, a new menu title, Objects, appears in the menu bar. Commands in this menu allow you to get information about and change properties of HyperCard *objects*—buttons, fields, cards, backgrounds, and stacks. (You'll learn more about objects later.) The user level must be set to Scripting before you can look at, write, or change these objects' scripts.

Creating a practice stack

Now that you've set the user level to Scripting, the next task is to create the stack that you'll work with throughout this book. You can make a new stack at any time from anywhere in HyperCard; you don't have to go back to the Home stack. Just follow these steps:

1. **Choose New Stack from the File menu.**

   A dialog box appears in which you can name the stack and specify its background and size.

2. **Type this name for the stack:** `Collection`

   If you make an error while typing the name, use the Backspace (Delete) key to erase it and retype. The dialog box should look similar to this:

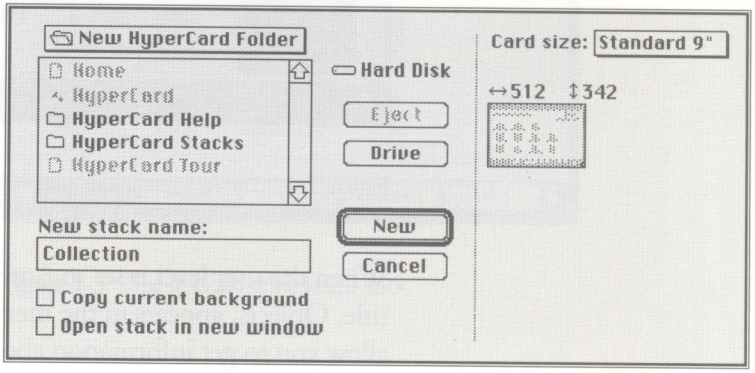


Figure 1-5 The New Stack dialog box

   From now on your practice stack will be referred to as the Collection stack.

3. **When you're ready, click New (or press Return).**

   You should see a completely blank card on your screen with the menu bar at the top. This card is the first—and right now the only—card of your Collection stack.

When you create a new stack, you automatically get three things: the stack itself, a background, and the first card. If you were to select the "Copy current background" option in the New Stack dialog box, you would also get the background pictures, fields, or buttons of the card you were on when you chose the New Stack command. Otherwise, as in this case, you have a blank card to work with.

Working in the background

Each card in a stack is a composite of two layers—a foreground layer, called the *card layer*, and a *background layer*. You see the elements of both layers when you look at a card, as if the card layer were a transparent layer in front of the background layer. Each layer can contain its own buttons, fields, and graphics.

You can think of the *background* in HyperCard as a kind of "holding area" for general elements. Each card with the same background has the same buttons, fields, and graphics in its background layer. If you put a button in the background, for example, you will have that button constantly available throughout a number of cards—you don't need to re-create it on every card. So far the Collection stack has only one background, so all the cards you create will share that background.

In the rest of this chapter you'll add buttons and fields to the background of the Collection stack.

Before you go on,

- **Choose Background from the Edit menu.**

  Stripes appear in the menu bar, indicating that you're working in the background:

Figure 1-6 Working in the background

You can also work in the background by pressing ⌘-B. (The ⌘ key is called the Command key and is located to the left of the Space bar on the keyboard.) Keyboard shortcuts like ⌘-B can save you a lot of time while you're creating a stack. You'll have many opportunities to practice HyperCard's most useful keyboard shortcuts as you work through this book.

When you're working in the background, you can't see any of the elements in the card layer. You can toggle back and forth between the background layer and the card layer by pressing ⌘-B repeatedly.

And now . . . a little scripting

In this section you will create a button and write a script for it. You may already know how to copy and paste buttons with prewritten scripts. In this book you'll complete the scripts yourself.

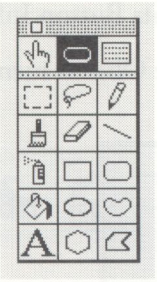
Creating a Home button

Whenever you see a small picture of a house in HyperCard, you can be pretty sure that clicking it will take you to the Home stack. In this section you'll add a Home button to your stack and complete its script.

First you'll create the button. You may already know how to create buttons by choosing New Button from the Objects menu. In this book you'll use a keyboard shortcut to create buttons. Follow these steps:

1. **Make sure you're working in the background.**

   You should see stripes in the menu bar. If you don't see stripes, press ⌘-B.



The Button tool

2. **Choose the Button tool from the Tools menu.**

   The pointing hand (Browse tool) on the screen changes to an arrow pointer.

   If you prefer to work with a palette, you can turn the Tools menu into a palette by dragging past its bottom edge to "tear" it off the menu bar.

   ❖ *A useful shortcut:* You can also choose the Button tool by pressing ⌘-Tab-Tab. ❖

3. **With the pointer anywhere on the card, hold down the ⌘ key.**

   The arrow pointer changes to a crosshair.

4. **While holding down the ⌘ key, press the mouse button and drag to create a small button about half an inch square.**

   Release the mouse button and the ⌘ key when the button is approximately the correct size. The new button is automatically selected—you can tell by the moving dashed line around its edges. (This effect is sometimes called "marching ants.") While it's selected, you can stretch or shrink the button by dragging a corner.

5. **Move the button to the lower-right corner of the background.**

   To move the button, position the pointer near the center of the button and drag.

   Because the button is in the background, it will appear in this position on every card in the stack, so you can always go Home.

6. **Double-click the button to see its Button Info dialog box.**

   Or choose Button Info from the Objects menu.

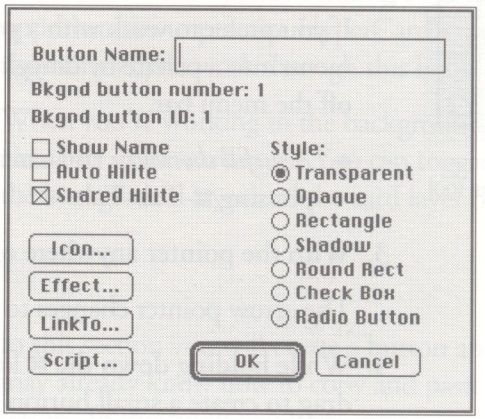


Figure 1-7 The Button Info dialog box

HyperCard buttons have a variety of styles and features from which to choose. You customize a button's appearance and actions through the Button Info dialog box.

A blinking vertical bar marks the insertion point in the Button Name box, ready for you to type a name.

7. **Type `Home` to name the button (but don't press Return).**

8. **Click the Auto Hilite check box to select it.**

   The Auto Hilite option causes the button to flash when it's clicked.

   Leave the Show Name check box deselected. You'll assign an icon to this button instead of showing its name.

9. **Click Icon.**

   Another dialog box appears in which you can select an icon for the button.

Some house icons

10. **Choose one of the house icons.**

    Scroll through the window until you find the house icons and click the one you want.

11. **Click OK.**

    The dialog boxes disappear. Your new button now has the house icon on it.

Next you'll write a script for this button.

Writing the script

You create and change scripts in a special window called the *script editor.* To see the script for the new Home button, make sure the Button tool is selected and follow these steps:

1. **Double-click your Home button.**

   Or click the button to select it and choose Button Info from the Objects menu. The Button Info dialog box appears.

2. **Click the Script button.**

   The script editor for the Home button appears (see Figure 1-8). There are two lines of text already in the window.

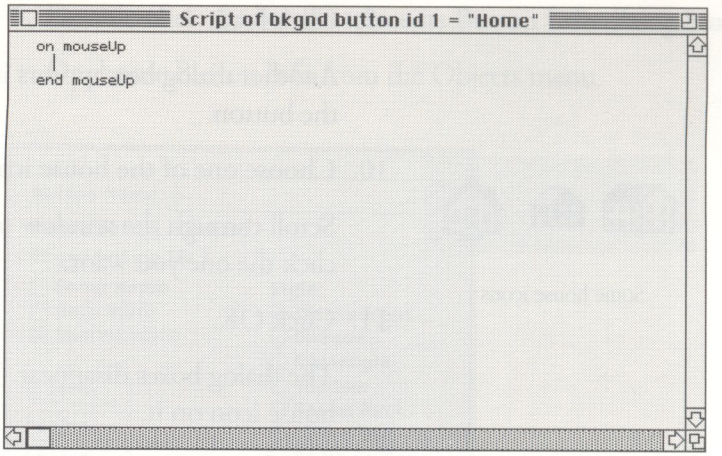


Figure 1-8 The script editor

Notice that the title bar at the top of the script editor identifies this script as the script of the background button with the ID number 1 and the name Home—your new button.

Also notice the Script menu that appears to the right of the other menus in the menu bar. (You'll learn more about this menu later.)

The next step is to type a statement that defines the button's action.

3. **Type** `go to stack "Home"`

   These three lines make up the complete script for the Home button:

```
on mouseUp
  go to stack "Home"
end mouseUp
```

   If you make a mistake, use the Backspace (Delete) key to erase the mistake and finish typing the script correctly.

The script editor automatically indents lines within scripts. This indenting helps you check your scripts. `On` and `end` should always line up at the leftmost edge of the script editor window when you're finished typing a script; if they don't line up, press the Tab key to check the script's formatting. If they still don't line up, you may have left out something important; check the script again.

❖ *By the way:* It doesn't matter how you capitalize HyperTalk words. Words that are formed from two words (such as `mouseUp`) are usually typed in small letters with a capital in the middle likeThis to make them more readable. ❖

To save your script and leave the script editor:

4. **Press the Enter key.**

   The script editor disappears, and you're looking at the Collection stack again. By pressing Enter, you save any changes you made to the script and return to the stack you're working on.

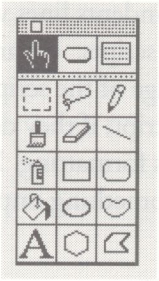
   If you click the Close box in the upper-left corner of the script editor, HyperCard will ask you if you want to save changes to the script before closing the script editor.

Trying out the Home button

Now see if the Home button works as it's supposed to.

1. **Press ⌘-B to return to the card layer.**

   The stripes in the menu bar should disappear. If they don't, press ⌘-B again.



The Browse tool

2. **Choose the Browse tool from the Tools menu.**

   The arrow pointer changes to a pointing hand—the Browse tool.

   ❖ *A useful shortcut:* You can also choose the Browse tool by pressing ⌘-Tab. ❖

3. **Click the Home button.**

   The Home stack appears. Welcome Home!

If something else happened, such as a dialog box saying "Can't understand," you may have made a typing mistake. Switch to the Button tool, double-click the Home button, and click Script in the Button Info box to check the script. Make sure everything is correct; then press Enter and try out the Home button again.

How the script works

As you might guess, the script you wrote describes what should happen when someone clicks the Home button.

Whenever you move the mouse, your Macintosh computer and HyperCard software track the movement electronically. You see the movement as a change in the position of the pointer on the screen. When you press and release the mouse button, the mouse sends an electrical signal to the computer, much the way a light switch works when you turn it on or off. The same thing is true when you press different keys on the keyboard. The HyperCard software interprets these signals from the system and translates them into HyperTalk *system messages.*

`MouseUp` is a system message that means the mouse button has been released; an on-screen HyperCard button receives this message when someone clicks it (that is, positions the Browse tool on it and then presses and releases the mouse button).

Whether something happens when the on-screen button receives the `mouseUp` message depends on whether the button's script contains any instructions for that message.

The first line of your script, `on mouseUp`, signals HyperCard that instructions for the `mouseUp` message exist. The next line, `go to stack "Home"`, tells HyperCard to go to the Home stack.

The word `go` is a HyperTalk *command;* it means what you might expect. `Go` must be followed by a destination—a description of a stack or a card. In this case, you used the name of the stack `Home`.

Each line in a script is called a HyperCard *statement.* In a more complicated script, the instructions signalled by `mouseUp` could consist of many statements. The last line of your script, `end mouseUp`, indicates the end of the instructions for the `mouseUp` message.

Translated into English, the instructions in your script say: "When someone clicks this button, go to the Home stack. That's all."

Adding a button to the Home stack

Wouldn't it be convenient to have a button in the Home stack that would take you directly to your Collection stack? In this section you'll create one.

Make sure you're looking at the Home stack and follow these steps:

1. **Choose the Button tool.**
2. **While holding down the ⌘ key, drag to create a new button.**

   Make the button about an inch wide and half an inch high. Move it to any open space on the card.

   ❖ *By the way:* If you need room, you can move any of the buttons in the Home stack just by clicking them with the Button tool and dragging them to a new location. ❖

3. **Double-click the button to see its Info dialog box.**

   Or choose Button Info from the Objects menu.

4. **Name the button** `Collection`

5. **Click Show Name and Auto Hilite to select them.**

   When Show Name is selected, the button's name appears inside the button.

6. **Click Icon.**

   The list of icons appears.

Stack icon

7. **Choose the stack icon.**

8. **Click OK.**

   Your new button now has the stack icon on it along with the name of the button.

Writing the script

Now you're ready to write the script.

1. **Double-click the Collection button.**

   The Button Info dialog box appears.

2. **Click the Script button to see the script editor.**

3. **Type** `go to stack "Collection"`

   The completed script should look like this:

```
on mouseUp
  go to stack "Collection"
end mouseUp
```

4. **Press Enter.**

   The Home stack appears with the Collection button in place.

5. **Try out the new button by clicking it with the Browse tool.**

   If the Collection stack appears, congratulations!

   If something else happened, you may have misspelled a word or left out a space in the button's script. If you got a directory dialog box asking where the stack is, you may have typed the stack's name incorrectly.

The words `go to stack "Collection"` tell HyperCard to go to the Collection stack. HyperTalk is a flexible language; any of these statements would also have worked in the button's script:

```
go "Collection"
go to "Collection"
go to card 1 of stack "Collection"
```

Message handlers

Buttons, fields, backgrounds, cards, and stacks are known as HyperCard *objects*. An object can send and receive *messages*. (For example, when you click a button, the button receives a `mouseUp` message.) As you've seen, when an object receives a message, it can act on the message according to instructions in the object's script. More specifically, the object acts according to instructions in the message handler.

A *message handler* is a set of instructions to be carried out when a particular object receives a particular message. It's called a handler because it "handles" the message. Handlers always begin with the word `on` and end with the word `end`, and both words are followed by the name of the whatever message the handler deals with—for example, `on mouseUp`. Each of the scripts you have written so far contain only one message handler, but an object's script can contain a number of handlers, each one handling a different message. The word *script* therefore refers to all the handlers for a given object.

In some ways writing a script for a HyperCard object is like training a dog (see Figure 1-9). The dog is like a HyperCard object, and a spoken command is like a message. Each of the dog's tricks—the response of a particular dog to a particular command—is like a message handler. And the sum of all the dog's tricks represents the "script" for the dog. When the dog receives a message (for example, "fetch"), the dog searches through its script for the appropriate handler and then acts according to the instructions in that handler.

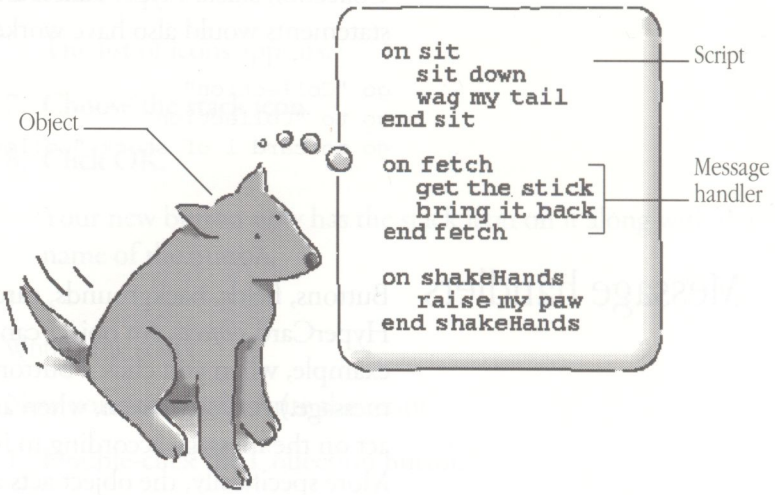


Figure 1-9 Message handlers: an analogy

❖ *By the way:* The words `on` and `end` belong to a special group of HyperTalk words known as *keywords*. Keywords are used to control which statements are executed in a script. ❖

Putting information into your stack

So far the Collection stack consists of a single card with a Home button. In this section you'll add fields to the background of the stack, type some text into the fields, and add some cards to the stack.

Adding fields to the background

First you'll add four fields to the background. When you're finished, the background will look similar to this:

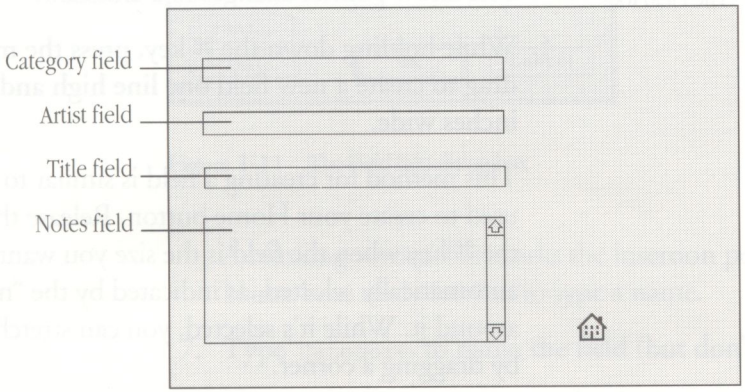


Figure 1-10 Background fields for the Collection stack

Because you'll place these fields in the background, they will appear on every card in your stack. The text contained in the fields can be different on every card.

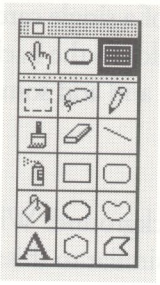
Creating the Category field

You can always get a new field by choosing New Field from the Objects menu. In this book you'll use a keyboard shortcut to make fields.

Follow these steps:

1. **Press ⌘-B to work in the background.**

   Stripes appear in the menu bar.



The Field tool

2. **Choose the Field tool from the Tools menu.**

   The pointing hand (Browse tool) on the screen changes to an arrow pointer.

   ❖ *A useful shortcut:* You can also choose the Field tool by pressing ⌘-Tab-Tab-Tab. ❖

3. **With the pointer anywhere on the card, hold down the ⌘ key.**

   The arrow pointer changes to a crosshair.

4. **While holding down the ⌘ key, press the mouse button and drag to create a new field one line high and about three inches wide.**

   This method for creating a field is similar to the method you used to create your Home button. Release the mouse button and the ⌘ key when the field is the size you want. The new field is automatically selected, as indicated by the "marching ants" around it. While it's selected, you can stretch or shrink the field by dragging a corner.

5. **Move the field to the top of the background (like the Category field in Figure 1-10).**

   To move the field, position the pointer near the center of the field and drag. Because the field is in the background, it will appear in this position on every card in the stack.

6. **Double-click the field to see its Field Info dialog box.**

   Or choose Field Info from the Objects menu.

   HyperCard fields have a variety of styles and options from which to choose. You customize a field's appearance and actions through the Field Info dialog box.

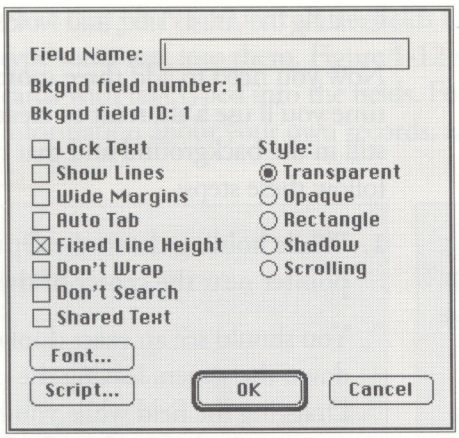


Figure 1-11 The Field Info dialog box

A blinking vertical bar marks the insertion point in the Field Name box, ready for you to type a name.

7. **Type `Category` to name the field (but don't press Return).**

   If you press Return prematurely, don't worry; just double-click the field again to get back to the Field Info dialog box.

8. **Click Rectangle to set the field's style.**

9. **Click the Font button.**

   The Text Style dialog box appears.

10. **Select a font and size.**

    Select a font that's easy to read, such as Times 14.

11. **Click OK.**

    The Text Style dialog box closes, and you return to the background.

Creating the Artist, Title, and Notes fields

Now you need to add three more fields to the background. This time you'll use a shortcut to create each field. Make sure you're still in the background and that the Field tool is still selected; then follow these steps:

1. **While holding down the Option and Shift keys, position the pointer near the center of the Category field and drag down.**

   You should see an exact duplicate of the Category field move down the screen, leaving the original Category field in place. Dragging the field while you hold down the Option key creates an exact duplicate of the field. Dragging the field while you hold down the Shift key restricts your movement of the field to either straight up and down or straight left and right. Dragging the field while you hold down both keys produces both effects simultaneously. (Both of these shortcuts also work for buttons.)

   Now all you need to do is change the name, and you'll have a new field with the same size and other characteristics as the Category field.

2. **Double-click the new field to see its Field Info dialog box, and name the field** `Artist`

   Except for its name, number, and ID, the Artist field will have all the same characteristics as the Category field.

3. **Using the same shortcut, Option-Shift-drag the Artist field to create a third background field, and name it** `Title`

4. **Option-Shift-drag the Title field to create a fourth background field, and name it** `Notes`

   In addition to changing the name of this field, you should enlarge it by dragging a corner until it's a few inches high, as shown in Figure 1-10. After you rename the field, change the field's style to Scrolling.

Typing in the fields

Now that you've created all the fields for your stack, you're ready to type some text into them. Figure 1-12 shows some examples of cards with text typed into the fields. For your own stack, type in information about your own records, tapes, or compact disks.

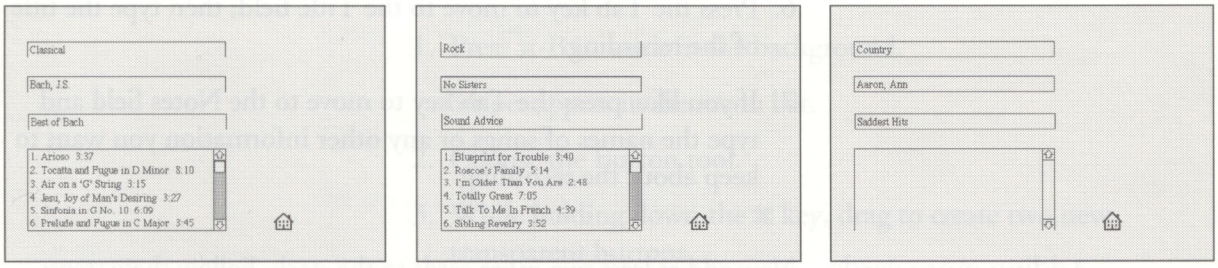


Figure 1-12 Sample record cards

To type text into the fields, follow these steps:

1. **Press ⌘-B to return to the card layer.**

   You see the first and only card in the stack. Make sure there are no stripes in the menu bar.

2. **Choose the Browse tool.**

3. **Click inside the Category field and type the category of music to which the recording belongs.**

   Type "Rock," "Jazz," "Classical," "Country," or any other category you want to use. Don't press Return.

4. **Press the Tab key.**

   The insertion point moves to next field you created—in this case the Artist field.

5. **Type the name of the artist featured on the recording.**

   Type the name as you would like it to be sorted alphabetically. For example, "Johann Sebastian Bach" should be entered as "Bach, Johann Sebastian."

6. **Press the Tab key to move to the Title field; then type the title of the recording.**

7. **If you like, press the Tab key to move to the Notes field and type the names of songs or any other information you want to keep about the recording.**

Adding more cards to the stack

Now add at least two more cards to the stack. Follow these steps:

1. **Choose New Card from the Edit menu.**

   Or press ⌘-N. A new card appears on the screen.

   ❖ *Disappearing fields:* If a field disappears when you create a new card, you probably placed the field in the card layer rather than the background layer. To move a field from the card layer to the background, go back to the card where you last saw the field; then click the field with the Field tool to select it. Press ⌘-X to cut the field, press ⌘-B to go to the background, and press ⌘-V to paste the field in the background. (You'll also have to return to the card layer and retype the contents of the field.) ❖

2. **Type information about another recording into the fields on the new card.**

Repeat these steps to add as many cards as you want to your stack.

Buttons for traveling

In this section you'll create two buttons that allow you to move forward and backward between cards in your stack.

Creating Next and Previous buttons

To make the buttons, use the same steps you followed when you made the Home button:

1. **Press ⌘-B to work in the background.**

   Stripes appear in the menu bar.

2. **Choose the Button tool.**

3. **While holding down the ⌘ key, drag to create two new transparent buttons.**

   Make each new button about the same size as the Home button.

4. **Position the two new buttons side-by-side, near the Home button.**

   Drag each button by its center to move it.

Customizing the button on the right

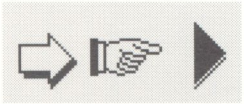
Make the button on the right into a Next button:

1. **With the Button tool still selected, double-click the button on the right.**

   The Button Info dialog box appears.

2. **Name the button** `Next`

3. **Click Icon to see the available icons.**



Some Next icons

4. **Choose any icon that points to the right.**

   You can choose any size arrow or pointing finger. Click the one you want.

5. **Click OK.**

   The dialog boxes disappear. You should see the icon you chose on the button.

Customizing the button on the left

Make the button on the left a Previous button:

1. **With the Button tool still selected, double-click the button on the left.**

   The Button Info dialog box appears.

2. **Name the button** `Previous`

3. **Click Icon to see the available icons.**

4. **Choose an icon that points to the left.**

   It's best to use the same kind of arrow or pointing finger that you chose for the first button, but pointing the opposite way.

5. **Click OK.**

   The two buttons now have matching icons pointing away from each other.

Writing the scripts

You want the Next button (the button on the right) to take you to the next card in the stack, and you want the Previous button (the button on the left) to take you to the previous card. Put your instructions into the buttons' scripts:

1. **With the Button tool still selected, double-click the Next button.**

   The Button Info dialog box appears.

2. **Click the Script button.**

   The script editor appears.

3. **Type** `go to next card`

   The completed script should look like this:

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

4. **Press Enter.**

   The script editor disappears.

Now you need to repeat the steps for the Previous button using a slightly different instruction.

1. **Double-click the Previous button to see its Info dialog box.**
2. **Click Script.**
3. **Type** `go to previous card`

   The script should look like this:

```
on mouseUp
  go to previous card
end mouseUp
```

4. **Press Enter.**

You have completed the scripts for both buttons. Now try them out.

5. **Press ⌘-B to return to the card layer.**
6. **Choose the Browse tool and click the Next button.**

   You go to the next card in the stack.

   ❖ *Disappearing buttons:* If a button disappears when you go from one card to another, you probably placed the button in the card layer rather than the background layer. To move a button from the card layer to the background, go back to the card where you last saw the button; then click the button with the Button tool to select it. Press ⌘-X to cut the button, press ⌘-B to go to the background, and press ⌘-V to paste the button in the background. (Return to the card layer when you're finished.) ❖

7. **Click the Previous button.**

   You go to the previous card.

You can use the Next and Previous buttons to move among the cards in the Collection stack. Cards in a stack are arranged in a circle, so the first card is the next one after the last card.

Moving to adjacent cards isn't the only possibility, of course; you can create other buttons to take you to any card of any stack you want by specifying in a script where you want to go.

Adding visual effects

Visual effects can make the movement from one card to another more obvious and interesting. In this section you'll learn two ways to add visual effects to buttons.

Adding a visual effect to the Next button

First you'll use a dialog box to add a visual effect to the Next button. Follow these steps:

1. **Choose the Button tool and double-click the Next button.**

   The Button Info dialog box appears.

2. **Click the Effect button.**

   The Effect dialog box appears.

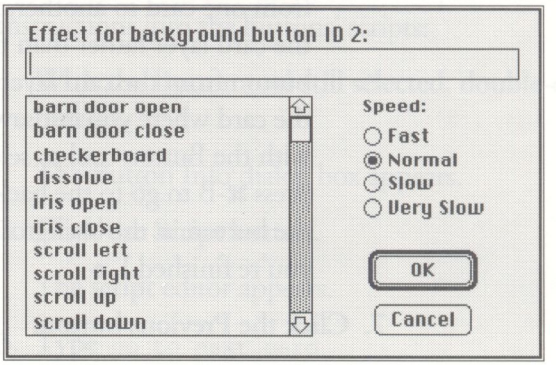


Figure 1-13 The Effect dialog box

This dialog box lets you choose which visual effect you want to display, as well as how fast you want the effect to take place. The scrolling field contains an alphabetical list of all the visual effects you can choose, and the buttons to the right list the speeds you can choose.

3. **Select "scroll left" from the list of visual effects.**

   The words `visual effect scroll left` appear at the top of the dialog box.

4. **Click the Fast button to select it.**

   The line at the top of the dialog box should now say `visual effect scroll left fast`.

5. **Click OK to close the dialog box.**

6. **To see how the visual effect works, watch the screen and click the Next button with the Browse tool.**

   The next card appears to slide in from the right.

What you have done is added a `visual effect` command to the script for the Next button. You can see for yourself by opening the button's script; it should now look like this:

```
on mouseUp
  visual effect scroll left fast
  go to next card
end mouseUp
```

Adding a visual effect to the Previous button

Using the Effect dialog box is one way of adding a visual effect to a script. Another way is to type the `visual effect` command yourself. You'll do that in this section.

1. **Choose the Button tool and double-click the Previous button.**

   The Button Info dialog box appears.

2. **Click Script.**

   The script editor appears showing the button's script.

3. **Click before the word `go` to set the insertion point.**

   Now you're ready to type the statement that creates a visual effect.

4. **Type `visual effect scroll right fast` and press Return.**

   The script should now look like this:

```
on mouseUp
  visual effect scroll right fast
  go to previous card
end mouseUp
```

5. **Press Enter.**

   The script editor disappears.

6. **Choose the Browse tool and click the Previous button.**

   The previous card appears to slide in from the left of the screen.

❖ *By the way:* Notice that you use `scroll left` for the *right* arrow and `scroll right` for the *left* arrow to simulate page turning. ❖

The syntax of the visual command

All languages—for people and computers—have rules of *syntax*. Syntax is a description of the way in which words are combined to form meaningful statements. For example, in English the statement "Go to the store" makes sense because it follows the rules of English syntax. However, the statement "The go store to" doesn't make sense because it doesn't use proper syntax.

HyperTalk syntax is much like English syntax, which makes HyperTalk an easy language to use. It's not always true, however, that a statement that makes sense in English will make sense in HyperTalk. For example, if you were to write the command

```
visual effect slow dissolve to black
```

HyperCard would not be able to understand the command because the words are in the wrong order. (The correct order is `visual effect dissolve slow to black`.) You would see a "Can't understand" dialog box like this:

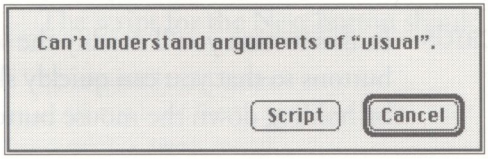


Figure 1-14 A "Can't understand" dialog box

Clicking Script in a "Can't understand" dialog box opens the script editor and places the insertion point at the beginning of the statement HyperCard can't understand. You can then correct any errors in syntax or spelling and try your script again.

The syntax of a HyperTalk statement describes the general, underlying structure that a statement must follow. In order for HyperCard to understand a statement, it must contain the correct elements in the correct order. Certain conventions are used to show the syntax of HyperTalk statements. For example, here's the syntax of the `visual` command:

`visual [effect]` *effect* `[`*speed*`] [to` *image*`]`

Syntax elements in `this kind of type` are typed exactly as they appear.

Elements in *italics* are placeholders. In an actual statement, you would replace *effect* with the name of an actual visual effect, such as `zoom out`.

Syntax elements enclosed in brackets [ ] are optional. (You don't include the brackets in an actual command.)

Knowing a command's syntax is as important as knowing what it does. But don't worry—you don't have to memorize syntax. A reference section, "Syntax Summaries," at the end of each chapter in this book describes the syntax of the commands you've learned. The Appendix and the HyperTalk Quick Reference Card at the end of the book list the syntax of every HyperTalk command, and the HyperTalk Reference stack describes the syntax of every command in detail.

Skimming cards

In this section you'll modify the scripts for the Next and Previous buttons so that you can quickly skim through a number of cards by holding down the mouse button.

Modifying the Next button's script

First modify the script for the Next button. You'll use a shortcut for seeing a button's script.

1. **Choose the Browse tool and then hold down the ⌘ and Option keys.**

   Pressing these two keys lets you see the outline of all buttons on the card.

2. **While holding down ⌘ and Option, click the Next button.**

   The script editor appears showing the button's script. (Release the keys after the script editor appears.)

   The ⌘-Option-click shortcut allows you to go directly to a button's script without switching to the Button tool first—a handy feature when you're doing a lot of scripting.

   ❖ *By the way:* Even though you had to switch to the background when you created this button, you do not have to switch to the background to change its script. ❖

3. **Select the word `mouseUp` in the first line of the script.**

   Drag across the word to select it, or double-click the word.

4. **Type `mouseStillDown`**

   You are changing the handler from a `mouseUp` handler to a `mouseStillDown` handler. The `mouseUp` message is sent only once—when the mouse button is released. The `mouseStillDown` message is sent continuously—as long as the mouse button is held down.

5. **Change the word `mouseUp` in the last line to `mouseStillDown`.**

   The script for the Next button should now look like this:

```
on mouseStillDown
  visual effect scroll left fast
  go to next card
end mouseStillDown
```

   In English, the script says, "As long as the mouse button is pressed, display a visual effect and go to the next card. That's all."

6. **Press Enter.**

   The Browse tool is still selected.

Now see how the Next button behaves.

7. **Click the Next button—making sure to release the mouse button immediately.**

   You go to the next card, just as you did before.

8. **Click the Next button again—this time holding down the mouse button for several seconds.**

   You skim through a number of cards until you release the mouse button.

As long as you hold down the mouse button, HyperCard repeatedly sends `mouseStillDown` messages to the button. The handler beginning `on mouseStillDown` executes again and again until you release the mouse button.

Modifying the Previous button's script

Now you will modify the Previous button. This time you'll use one of the commands in the Script menu.

1. **Hold down the ⌘ and Option keys and click the Previous button.**

   The script for the Previous button appears.

2. **Choose Replace from the Script menu.**

   The Replace dialog box appears. You can use this dialog box to find and replace text within a script. In this case, you're going to replace all instances of the word `mouseUp` with the word `mouseStillDown`.

3. **In the box labeled "Find," type the word `mouseUp` (but don't press Return).**

4. **In the box labeled "Replace with," type the word `mouseStillDown` (but don't press Return).**

   The dialog box should look like this:

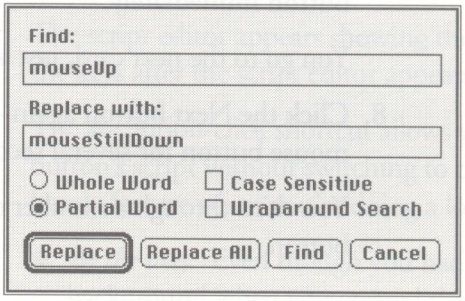


Figure 1-15 The Replace dialog box

5. **Click Replace All.**

   HyperCard replaces all instances of the word `mouseUp` in the script with `mouseStillDown`.

   The script should now look like this:

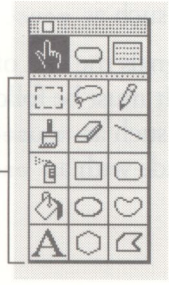
```
on mouseStillDown
  visual effect scroll right fast
  go to previous card
end mouseStillDown
```

6. **Press Enter.**

You can now use the Previous button to skim backward through the cards in your stack.

Adding graphics

If you'd like to give your stack a distinctive look, you can take some time now to design graphics for the background. Well-designed graphics can make your stack easier to use, as well as more appealing visually.



The Paint tools

You can create graphics by using the Paint tools, or you can copy clip art from the Art Bits stack. You may also want to change the fonts in the background fields or the position of the fields and buttons. You'll be adding more buttons to the background later, so be sure to leave space for them.

(For instructions on how to use Paint tools, see the *HyperCard Reference*. For tips on how to design stacks, see the *HyperCard Stack Design Guidelines*, published by Addison-Wesley.)

You can leave your stack as it is, copy one of the designs suggested in Figure 1-16, or have fun creating a design of your own. When you're satisfied with the way your stack looks, you can move on to Chapter 2.

Figure 1-16 Some possible designs

What you've done so far

In this chapter you've created a stack in which you can practice scripting in the rest of this book. You've created fields and added cards to the stack. You've also created some buttons and written their scripts.

Here's a list of the HyperTalk words you have learned:

Keywords

`on` — This word signals the beginning of a set of instructions. It must be followed by the name of a message, such as `mouseUp`.

`end` — This word signals the end of a set of instructions. It must be followed by the name of a message, such as `mouseUp`. All HyperTalk scripts conclude with an `end` statement.

System Messages

`mouseUp`	When you click something, such as a button on the screen, the system sends `mouseUp` when the mouse button is released. (If the pointer is moved off the screen button before the mouse button is released, `mouseUp` is not sent.)
`mouseStillDown`	When you hold the mouse down, the system sends `mouseStillDown` repeatedly until you release the mouse button.

Commands

`go`	This command is used to move from one card to another, within a stack or between stacks.
`visual [effect]`	Causes the visual effects you specify. A `visual` command must eventually be followed by a `go` command.

Miscellaneous

`scroll left` `scroll right`	Names of visual effects.

Syntax summaries

This section describes the basic structure of the two HyperTalk commands you've learned so far.

Go

The `go` command takes you to the specified card or stack. If you name a stack without specifying a card, you go to the first card in the stack. If you don't name a stack, you go to the specified card in the current stack. 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.

Syntax:

```
go [to] stack
go [to] background [of stack]
go [to] card [of background] [of stack]
```

The words *stack*, *background*, and *card* are placeholders. You would replace them with a word or phrase that describes a stack, a background, or a card.

Examples:

```
go "Home"
go to first card
go to card 3 of background 2 of "Presentation"
```

Visual

The `visual` command lets you display visual effects while going from one image to another. The `visual` command must eventually be followed by a `go` command.

Syntax:

`visual [effect]` *effect* `[`*speed*`] [to` *image*`]`

Effect is one of the following:

```
barn door close          shrink to top
barn door open           stretch from bottom
checkerboard             stretch from center
dissolve                 stretch from top
iris close               venetian blinds
iris open                wipe down
plain                    wipe left
scroll down              wipe right
scroll left              wipe up
scroll right             zoom close
scroll up                zoom in
shrink to bottom         zoom open
shrink to center         zoom out
```

Speed is one of the following:

```
fast                     very fast
slow[ly]                 very slow[ly]
```

Image is one of the following:

```
black                    inverse
card                     white
gray
```

Examples:

```
visual effect barn door open
visual dissolve slowly to white
```

Fields and Other Containers

In everyday life, a container is something you can put things into. In HyperTalk, a *container* is a place in the computer's memory where you can put a *value*, such as a number or some text. You can put values into containers; you can also get values out of containers to use elsewhere as needed.

In this chapter you'll learn about three different kinds of containers: the Message box, fields, and variables. You'll also learn how handlers can work with values in containers. You'll add some more features to your Collection stack, and you'll increase your vocabulary of HyperTalk commands.

If you took a break at the end of the previous chapter, start up HyperCard and go to the Collection stack before you read on.

Putting values into containers

You use the `put` command to put a value into a container. In this section, you'll practice using the `put` command to put values into the Message box. Later in this chapter you'll use the `put` command in scripts.

Putting values into the Message box

First open the Message box.

1. **Press ⌘-M to open the Message box.**

   Or choose Message from the Go menu.

   The vertical bar that marks the insertion point should be blinking inside the Message box, ready for you to type. If for any reason you previously typed something into the box, the earlier entry will still be there. When you start typing, whatever you type will replace the old text.

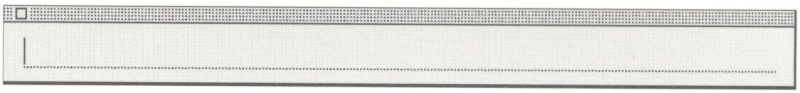


Figure 2-1 The Message box

2. **Type** `put "Hello" into the message box` **and press Return.**

   The word `Hello` appears in the Message box.

The `put` command does what you would expect—it puts a value where you want it to go. In its most basic form, the syntax of the `put` command is

`put` *expression* `[into` *container*`]`

The placeholder *expression* is a word or phrase that specifies a value. For example, the expression `2 + 2` specifies the value `4`.

The placeholder *container* can be a field, a variable, or the Message box. If you don't specify a container, the value is put into the Message box.

❖ *By the way:* After you press Return, you can start typing a new message into the Message box right away, even though you can't see the blinking vertical bar. Whatever you type will replace the old text. ❖

3. **Type `put 2 + 2` and press Return.**

   `4` appears in the Message box.

Including quotation marks around text characters tells HyperTalk to interpret literally whatever is inside. It treats what's inside the quotation marks as a string of text characters.

If you don't include quotation marks, HyperTalk evaluates the expression. That is, it replaces the expression with *the value of* the expression. For example, it replaces `2 + 2` with `4`.

4. **Type `put "It is now" && the time` and press Return.**

   A text string giving the current time appears in the Message box. For example:

   ```
   It is now 12:00 PM
   ```

   The double ampersand (`&&`) joins two pieces of text together with a space in between. In this case, it joins the words `It is now` and the current system time. (If you wanted to join two pieces of text together without a space, you would use a single ampersand.)

5. **Close the Message box.**

   Click the close box in the upper-left corner, or press ⌘-M.

Now that you've seen how the `put` command works, you'll use it in a script.

Fields as containers

Fields are objects. They can receive and send messages and have scripts. Fields are also containers that can contain text and numbers.

In Chapter 1 you put text into fields by typing in the fields. In this section you'll write a script that puts text into a field.

Putting values into a field

First you need to create a background field named `Label`. This field will display the number of each card in your stack, so you can easily tell where you are in the stack.

Creating the Label field

To create the field, follow these steps:

1. **Press ⌘-B to work in the background.**

   Stripes appear in the menu bar.

2. **Choose the Field tool.**

3. **Hold down the ⌘ key and drag to create a field one line high and about half an inch long.**

   Move the field to any available space in the background by dragging its center.

4. **Double-click the field to see its Info dialog box.**

   Or choose Field Info from the Objects menu.

5. **Name the field `Label` and set the field's characteristics.**

   Select Rectangle for the field's style. Choose any font you'd like.

6. **Click OK.**

Writing a script for the stack

You could label all cards in your stack by going to each one and typing its number into the Label field. But you can also write a script telling HyperCard to do it for you.

You'll write a script that puts a description of each card into the Label field. The field will contain a text string with two pieces: the word "Card" and the number of the current card.

To write the script, follow these steps:

1. **Choose Stack Info from the Objects menu.**

   The Info dialog box for the stack appears.

2. **Click the Script button.**

   The script editor for the stack appears. The line at the top of the script editor identifies it as the stack script.

3. **Type the following script:**

```
on openCard
  put "Card" && the number of this card into background field "Label"
end openCard
```

   In English, the script says, "When a card opens, put the word 'card' and the number of the card into the background field named Label. That's all."

4. **Press Enter.**

5. **Press ⌘-B to return to the card layer.**

6. **Choose the Browse tool and click the Next button several times.**

   Each time you go to another card, you should see in the Label field the word `Card` followed by the number of the current card.

   ❖ *If something else happened:* Check the script for spelling errors and make sure the `openCard` handler is in the stack script. Also make sure that the Label field is in the background and that its name matches the name you used in your script. ❖

How the script works

Just as HyperCard sends the system message `mouseUp` every time you click the mouse button, it sends the message `openCard` every time you go to a different card in the stack. When you open any card in the Collection stack, the `openCard` message handler executes and puts the number of the current card into the Label field. Because the `openCard` handler is in the script for the *stack*, it affects every card in the stack—not just a particular card.

The advantage of using a script to label cards is that you won't have to worry about labeling the cards yourself, even if you add or delete cards. HyperCard will take care of it for you.

❖ *Dealing with long statements:* The script editor doesn't wrap long lines or let you scroll to see text that extends beyond the window. In order to fit a long statement into the script editor window, you must manually break the statement into more than one line. You do this by pressing Option-Return at the end of a line. This inserts a "soft" Return character (¬) in your script. Although the statement appears on more than one line, HyperCard treats it as a single statement. ❖

Creating a pop-up field

Now it's time to give yourself a well-deserved pat on the back; you'll create a field that displays the credits for your stack. You'll create a button that makes the field appear and write a script that makes the field disappear when you click it.

Making the Credits field

You can start from any card in the Collection stack. Create the field by following these steps:

1. **Press ⌘-B to work in the background.**

   Stripes appear in the menu bar.

2. **Create a new field.**

   Use any method you want. Make the field about two inches long and an inch high. It's OK if the field covers other fields or buttons.

3. **Double-click the field to see its Info dialog box.**

4. **Name the field** `Credits`

5. **Click Shared Text.**

   Background fields with shared text contain the same text on every card.

6. **Click Shadow to set the field's style.**

7. **Choose a font for the field, if you'd like to.**

8. **Click OK.**

Now you'll type your message into the field. Because it's a background field with shared text, the message you type will appear on every card in the stack.

9. **Choose the Browse tool.**

10. **Click inside the Credits field to set the insertion point; then type the credits for your stack.**

    Figure 2-2 shows an example.

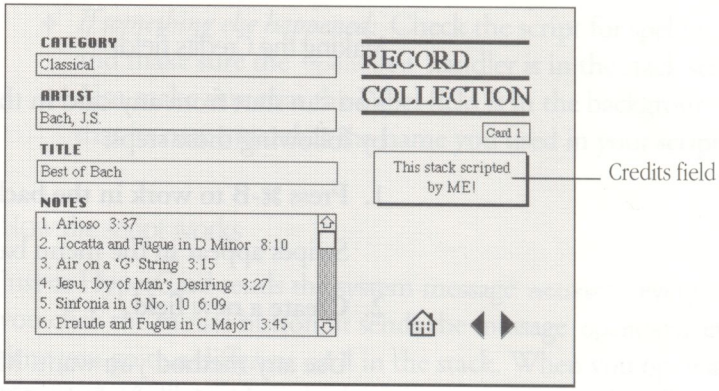


Figure 2-2 A card displaying the Credits field

Making an About button

Next you'll create a button that makes the Credits field appear. Make sure you're still working in the background, and follow these steps:

1. **Choose the Button tool and create a new button about half an inch square.**

   Drag the button to any available space in the background.

2. **Name the button `About` and select Auto Hilite.**

3. **Select a cartoon balloon icon from the Icon dialog box.**

   A cartoon balloon icon identifies an "About this stack" button.

Cartoon balloon icon

Now you'll write the script for the About button.

4. **While holding down the ⌘ and Option keys, click the About button.**

   The script editor appears.

5. **Type** `show bg field "Credits"`

   The completed script should look like this:

```
on mouseUp
  show bg field "Credits"
end mouseUp
```

The letters `bg` are an abbreviation for the word `background`. The Appendix includes a complete list of HyperTalk abbreviations.

In HyperTalk, you must use `card` or `cd` in front of `field` to specify a card field. If you leave out `card`, HyperCard assumes you mean a background field. Conversely, you must use `background`, `bkgnd`, or `bg` in front of `button` to specify a background button. Otherwise HyperCard assumes you mean a card button. To avoid confusion, it's a good idea to always use `card` or `background` when referring to fields and buttons.

6. **Press Enter.**

Writing a script for the Credits field

Next you'll write a script that makes the Credits field disappear when you click it.

1. **Choose the Field tool and double-click the Credits field to see its Info dialog box.**

2. **Click Script.**

   The script editor for the Credits field appears.

3. **Type the following script:**

```
on mouseDown
  hide me
end mouseDown
```

`MouseDown` is a system message that's sent as soon as the mouse button is pressed.

The HyperTalk word `me` refers to the object in whose script the word appears. In this case, `me` refers to the Credits field.

4. **Press Enter.**

Trying out the scripts

Now see how the About button and Credits field work.

1. **Press ⌘-B to return to the card layer.**
2. **Choose the Browse tool and click the Credits field.**

   The field disappears.

3. **Click the About button.**

   The Credits field reappears.

   ❖ *If something else happened:* Make sure the script for the About button is spelled correctly. Also make sure that the name of the Credits field is spelled correctly.

   If the Credits field still won't appear when it is supposed to, open the Message box and type `show last bg field`. Then check the spelling of the Credits field in the Field Info dialog box. ❖

4. **Click the Credits field to make it disappear.**

How the scripts work

When you click the About button, the `mouseUp` handler in the button's script executes. The statement `show bg field "Credits"` makes the Credits field visible.

When you press the mouse down on the Credits field, the `mouseDown` handler in the field's script executes. The statement `hide me` makes the Credits field disappear.

To be able to send messages to a field by clicking it, the field must *be locked or be a background field with shared text. Otherwise,* clicking the field merely places the insertion point inside the field.

You can use the `hide` command to make invisible a field, a button, a window (such as the Message box), the menu bar, the background picture (graphics in the background), or the card picture (graphics on the card that aren't in the background). The `show` command does just the opposite; you use `show` to reveal hidden elements.

Variables

A *variable* is a named container that can have any value you choose to put into it. In this section you'll create a button that uses a variable in its script.

Creating a Sort button

You'll create a button that sorts all the cards in your stack alphabetically. When a user clicks the Sort button, a dialog box appears that asks the question "Sort by what?" and presents three possible replies: Category, Title, or Artist. When the user chooses, HyperCard sorts the cards in the stack according to the contents of the chosen field.

Follow these steps to create the Sort button:

1. **Press ⌘-B to work in the background.**
2. **Choose New Button from the Objects menu.**

   A new button appears. When you choose the New Button command, you automatically switch to the Button tool, and the new button is automatically selected.

   Drag the button to any available space in the background.

3. **Name the button `Sort` and select Auto Hilite.**

   The Show Name option is already checked. You'll show the name of this button rather than assigning an icon to it.

4. **Click Script to see the script editor.**

5. **Type the following script:**

```
on mouseUp
  answer "Sort by what?" with "Category" or "Title" or "Artist"
  put it into sortReply
  sort by background field sortReply
end mouseUp
```

6. **Press Enter.**

Now try the Sort button to see how it works.

7. **Press ⌘-B to return to the card layer.**

8. **Choose the Browse tool and click the Sort button.**

   The following dialog box appears:

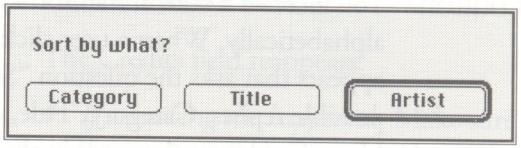


Figure 2-3 Dialog box displayed by the Sort button

9. **Click Artist.**

   HyperCard reorders the cards in the stack alphabetically according to the contents of the Artist field. Browse through your stack with the arrow buttons to see that the names of the artists are in alphabetical order.

If you would rather sort your cards by category or title, you can use the Sort button to do that, too.

How the script works

The `answer` command asks the user of your stack a question, and presents up to three possible replies in the form of buttons in a dialog box. In this case it asks the question `"Sort by what?"` and presents three possible replies: `"Category"`, `"Title"`, and `"Artist"`. (The `answer` command always highlights the last reply, so it's a good idea to list the safest or "most correct" answer last.)

When someone clicks a reply in the dialog box, that reply is put into a special HyperTalk variable named `it`. For example, when you click Artist, the value `Artist` is put into `it`.

The next statement in the script, `put it into sortReply`, puts the contents of `it` into another variable, which you've named `sortReply`. The names of variables can be almost anything you choose, but it's a good idea to name them something that describes what's contained in them.

If you clicked Artist, the variable `sortReply` would then contain the value `Artist`. Therefore, the statement

```
sort by background field sortReply
```

is evaluated as

```
sort by background field Artist
```

And HyperCard sorts all the cards in your stack according to the contents of the Artist field.

- *Local versus global:* The variables discussed here are *local variables;* that is, they and their values exist only within the handler in which they're created. HyperCard also has *global variables*, whose values are available to all handlers everywhere. You declare a variable to be a global variable by using the `global` keyword. For information about global variables and the `global` keyword, see the HyperTalk Reference stack or the *HyperCard Script Language Guide.* ❖

Putting comments in the handler

The following version of the handler for the Sort button shows comments that describe the action of the handler's statements. *Comments* are text lines typed into a script that are not part of the instructions. In HyperTalk, a comment must be preceded by two hyphens (--); the double hyphens indicate to HyperCard that the text following is a comment and should be ignored.

You don't have to type these comments into your own script; they are shown only as an example.

```
-- This button sorts the stack according to a field chosen by the user
on mouseUp
  answer "Sort by what?" with "Category" or "Title" or "Artist"
  -- The user's response is now in the variable it
  put it into sortReply   -- Response is now in sortReply
  sort by background field sortReply  -- Sorts the stack
end mouseUp
```

As you see, comments can be placed either at the beginning of a line or following a statement.

Although HyperCard ignores comments, other scripters generally appreciate them. Adding comments to your scripts is an excellent way to document what your scripts do. Comments not only help other scripters understand what you've done, but also help *you* remember when you look at old scripts long after you've written them.

What you've done in this chapter

You've learned how to use fields, variables, and the Message box as containers for text and numbers.

You've also added some features to your Collection stack: a handler that automatically numbers the cards in the stack, a pop-up field, and a Sort button.

Here's a list of the new HyperTalk words you have learned:

System messages

`openCard`	A message sent by HyperCard when a card is opened.
`mouseDown`	A message sent by HyperCard when the mouse button is pressed down.

Commands

`answer`	Puts a box on the screen containing a question and up to three response buttons.
`hide`	Makes buttons, fields, windows, and pictures invisible.
`put`	As you might guess, this command takes something and puts it somewhere. The word `put` must be followed by the name of the thing you want to put somewhere and the name of the place you want to put it.
`show`	Causes hidden buttons, fields, windows, and pictures to become visible.
`sort`	Sorts all the cards in a stack.

Miscellaneous

`&`	(Ampersand) This symbol joins two pieces, or strings, of text together with no space between them.
`&&`	(Double ampersand) This combination symbol joins two pieces of text with a space between them.
¬	("Soft" return character—produced by pressing Option-Return at the end of a line) Breaks long statements into more than one line in the script editor window.
`--`	(Double hyphen) Indicates that what follows is a comment and should be ignored by HyperCard.

`bg`	Abbreviation for `background`.
`me`	This word refers to the object in whose script it appears.
`it`	The name of a special HyperTalk variable. Certain commands, such as `answer`, put a value into `it`.

Syntax summaries

This section describes the syntax of the commands you used in this chapter.

Answer

The `answer` command displays a dialog box with a question and up to three buttons for the user to choose from, each representing a different reply. If you don't specify a reply, HyperCard displays a single OK button in the box.

HyperCard puts the label of whatever button gets clicked into a variable named `it`.

Syntax:

```
answer text
answer text with reply
answer text with reply1 or reply2
answer text with reply1 or reply2 or reply3
```

Text can be any text you like—usually a question that invites the user to answer. *Reply1, reply2,* and *reply3* are the labels for buttons representing the choices. The size limit for a reply is about 13 characters, depending on the width of the characters.

Example:

```
answer "Pick a color:" with "Red" or "Blue" or "Green"
```

Hide

The `hide` command makes invisible a button, field, picture, or window. (See also "Show," later in this section.)

Syntax:

```
hide button
hide field
hide window
hide card picture
hide background picture
hide picture of card
hide picture of background
hide menuBar
hide titleBar
```

Button, *field*, *card*, and *background* are expressions identifying objects (for example, `background button 1`.)

Window stands for the card window, one of the palettes (Tools or Patterns), or the Message box. Replace *window* with one of these names:

```
card window
tool window
pattern window
[the] message [box]
```

`Card picture` consists of all elements in the card layer created with a Paint tool. `Background picture` consists of all graphic elements in the background layer.

`menuBar` is the HyperTalk name for the menu bar.

`titleBar` is the HyperTalk name for the title bar.

Examples:

```
hide background field "About"
hide Message box
hide picture of card 1
```

Put

The `put` command places the value of an expression into a container.

Syntax:

```
put expression
put expression into [chunk of] container
put expression after [chunk of] container
put expression before [chunk of] container
```

Expression can be any description of a text string or a number.

Chunk consists of the words `character`, `word`, `item`, or `line` preceded by an ordinal or followed by a number, range of numbers, or another chunk expression.

Container is an expression that identifies a field, variable, the Message box, or the selection.

The preposition `into` causes anything already in the destination container to be replaced by the expression. The preposition `before` places the expression at the beginning of what's in the container (if anything), and `after` puts the expression at the end.

If you don't specify a destination, the expression is put into the Message box.

Examples:

```
put 256
put 256 into Total
put 256 into line 1 of card field 3
put 256 before word 4 of line 1 of card field 3
put 256 after word 3 of line 1 of card field 3
```

Show

The `show` command makes visible a button, field, picture, or window.

Syntax:

```
show button [at point]
show field [at point]
show window [at point]
show card picture
show background picture
show picture of card
show picture of background
show menuBar
show titleBar
```

See "Hide," earlier in this section, for a description of the placeholders.

Point consists of the horizontal and vertical coordinates of a point on the screen, separated by a comma. The optional phrase `at` *point* lets you place the window or object wherever you want it. If you don't include it, the window or object appears wherever it was before it was hidden.

Examples:

```
show background field "About"
show background field "About" at 10,20
show Message box
show picture of card 1
```

Sort

The `sort` command allows you to reorder all the cards in a stack from within a script.

Syntax:

`sort` [*sortDirection*] [*sortStyle*] `by` *expression*

SortDirection is `ascending` or `descending`. If you don't specify a direction, the direction is `ascending`. *SortStyle* is `text`, `numeric`, `dateTime`, or `international`. If you don't specify a style, the style is `text`.

Expression is any expression. The `sort` command reorders all the cards in a stack according to the value of *expression*, which is evaluated individually for each card in the stack.

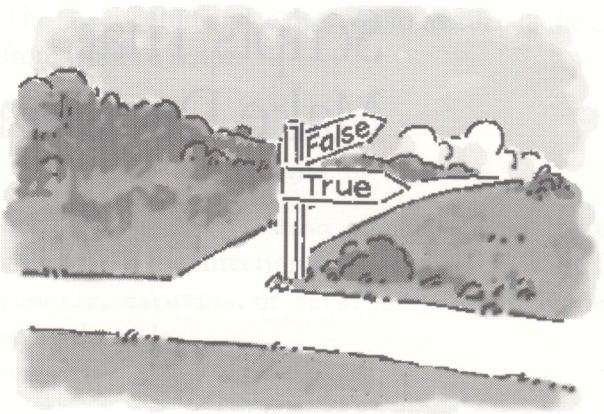
Examples:

```
sort by card field 1
sort descending numeric by card field 1
```

Scripts That Make Decisions

In this chapter you'll learn how to control which statements are executed in a handler, as well as the order in which they are executed. You'll create some buttons for your Collection stack and write scripts that use the HyperTalk words `if` and `repeat`. Using `if` and `repeat`, you can write scripts that are more responsive and efficient.

If you took a break at the end of Chapter 2, start up HyperCard and go to the Collection stack before you go on.



If structures

In English, we use the word *if* to talk about an action that depends on a certain condition. For example, we might say "If I am hungry, then I'll eat dinner." If the condition "I am hungry" is true, then the action "I'll eat dinner" will be performed.

In HyperTalk, the word `if` is used in much the same way. `If` and `then` are HyperTalk keywords that work together in arrangements called `if` structures. `If` structures are used to test conditions and to specify different actions, depending on the results.

`If` structures come in a few varieties; the most basic version is

```
if condition then
  action
end if
```

The placeholder *condition* stands for the thing being tested. It's an expression that HyperCard can evaluate as either true or false. The placeholder *action* stands for the instruction lines that follow if the condition is true. The last line, `end if`, signals the end of the `if` structure.

Here's how the English example would look if it could be written in HyperTalk:

```
if I am hungry then
  I'll eat dinner
end if
```

In English, the word *then* is often implied; in HyperTalk, you must always include it.

Creating a Quit button

In this section you'll create a button that uses an `if` structure in its script. When a user clicks the button, a dialog box will appear asking the user whether he or she wants to quit HyperCard. The dialog box will display two options: OK and Cancel. If the user clicks OK, he or she quits HyperCard. If the user clicks Cancel, the dialog box disappears and nothing else happens.

Follow these steps to make the Quit button:

1. **Press ⌘-B to work in the background.**
2. **Choose New Button from the Objects menu, and move the new button to any available space in the background.**
3. **Name the button `Quit` and select Auto Hilite.**
4. **Click Script to see the script editor.**
5. **Type the following script:**

```
on mouseUp
  answer "Quit HyperCard?" with "OK" or "Cancel"
  if it is "OK" then
    doMenu "Quit HyperCard"
  end if
end mouseUp
```

   Notice that the contents of the `if` structure are automatically indented. The statements beginning with `if` and `end if` should always line up. If they don't line up, you may have misspelled a word or left out something.

6. **Press Enter.**

Trying out the Quit button

Now try the Quit button to see how it works:

1. **Press ⌘-B to return to the card layer.**

2. **Choose the Browse tool and click the Quit button.**

   This dialog box appears:

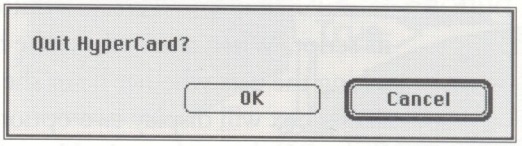


Figure 3-1 Dialog box displayed by the Quit button

3. **Click Cancel.**

   The text string `Cancel` is put into the variable `it`.

   Because the condition `it is "OK"` is not true, HyperCard doesn't execute the action specified within the `if` structure. The dialog box disappears, and nothing else happens.

4. **Click the Quit button again.**

   The dialog box appears again.

5. **Click OK.**

   The text string `OK` is put into the variable `it`.

   The condition `it is "OK"` is true, so HyperCard executes the statement within the `if` structure—and you quit HyperCard.

The `doMenu` command lets you execute any of HyperCard's menu commands from within a script. In this case it executes the Quit HyperCard command. (Be sure to put quotes around the name of the menu command when you use `doMenu`.)

To continue in this chapter you'll need to start up HyperCard again and return to the Collection stack.

Adding an action

An `if` structure can specify not only an action to be taken when a condition is true, but also an alternative action to be taken when the condition is false. `If` structures of this type have the general form

```
if condition then
   action
else
   anotherAction
end if
```

The placeholder *anotherAction* stands for an alternative instruction line or lines. An example in English might be something like this: "If I am hungry, then I'll eat dinner; otherwise [else] I'll go to the movies." Here's how it would look if it could be written in HyperTalk:

```
if I am hungry then
   I'll eat dinner
else
   I'll go to the movies
end if
```

Modifying the Quit button's script

In this section you'll add two statements to the script for the Quit button. You'll add an `else` statement and a statement that specifies an alternative action for when a user clicks Cancel.

1. **Open the script for the Quit button.**
2. **Click before `end if` to position the insertion point at the beginning of the next-to-last line.**
3. **Type the following lines (press Return after each line):**

```
else
answer "Glad you reconsidered." with "No problem"
```

The lines will automatically indent. When you press Return for the final time, `end mouseUp` should line up at the leftmost margin.

Here's the completed script (the two new statements are shown in boldface type):

```
on mouseUp
  answer "Quit HyperCard?" with "OK" or "Cancel"
  if it is "OK" then
    doMenu "Quit HyperCard"
  else
    answer "Glad you reconsidered" with "No problem"
  end if
end mouseUp
```

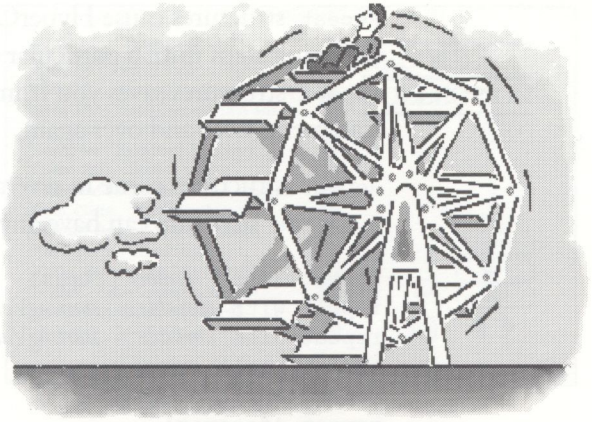
4. **Press Enter.**
5. **Try the Quit button.**

   When you click the Quit button with the Browse tool, you get the alert box, just as before. Clicking Cancel (the choice represented by `else`) makes another dialog box appear with a friendly comment and reply—just for fun. (No further instructions are specified for the "No problem" button.)

Decisions within decisions

It's possible to specify more than two separate actions by nesting `if` structures inside other `if` structures. Here's how an English example might look if it could be written in HyperTalk:

```
if I am hungry then
  if there's some food in the house then
    I'll cook
  else
    I'll order a pizza
else
  if there's a good movie at the theater then
    I'll go to the movies
  else
    I'll watch television
end if
```



Repeat structures

`Repeat` is a keyword that tells HyperCard to perform a command or series of commands over and over again. Suppose you wanted to create a sequence in which your stack moved through a series of five cards, with a one-second pause between cards. You could write the instructions this way:

```
wait 1 second
go to next card
wait 1 second
go to next card
wait 1 second
go to next card
wait 1 second
go to next card
wait 1 second
go to next card
```

Or you could write a `repeat` structure, like this:

```
repeat 5 times
  wait 1 second
  go to next card
end repeat
```

`Repeat` structures cause HyperCard to go around in a "loop," repeating steps until a particular endpoint occurs. Being able to use `repeat` structures saves you from having to retype or duplicate statements over and over again.

`Repeat` structures come in several varieties. The first line of a `repeat` structure can have any of these general forms:

```
repeat [for] number [times]
repeat with variable = startingValue to endingValue
repeat with variable = startingValue down to endingValue
repeat until condition
repeat while condition
repeat [forever]
```

The statement or list of statements that you want to have repeated can follow any of these first lines. You must include `end repeat` to indicate the end of the list. (For more information about variations of the `repeat` structure, see the end of this chapter.)

Creating an Index button

In this section you'll create a button that uses a repeat structure to generate an index of your stack. Each index entry will include the name of the recording artist and the title of the record. (Figure 3-2 shows what the index might look like.)

Later in this chapter you'll write a script that lets you go to a card by simply clicking an index entry.

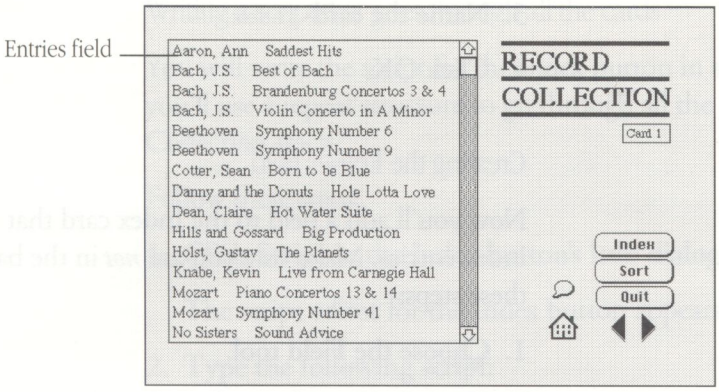


Figure 3-2 A sample index card

Creating the Index card

First you need to add a new card to your stack. Make sure you're looking at the Collection stack, and follow these steps:

1. **Choose New Card from the Edit menu.**

   Or press ⌘-N. A new card appears.

2. **Choose Card Info from the Objects menu.**

   The Card Info dialog box appears.

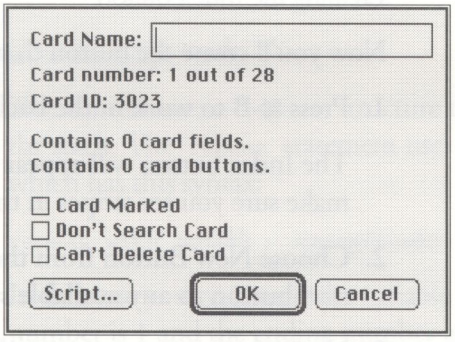


Figure 3-3 The Card Info dialog box

3. **Name the card** `Index`
4. **Click OK.**

Creating the Entries field

Now you'll add a field to the Index card that will display the list of index entries. Make sure you are *not* in the background, and follow these steps:

1. **Choose the Field tool.**
2. **While holding down the ⌘ key, drag to create a large field (like the scrolling field shown in Figure 3-2).**

   Make the field large enough to cover the Category, Artist, Title, and Notes fields.
3. **Double-click the field to see its Info dialog box.**

   Notice that the field is a card field (not a background field). The field appears only on this card.
4. **Name the field** `Entries`
5. **Select Scrolling for the field's style.**
6. **Click OK.**

Creating the Index button

Now you'll create the button that automatically generates the index.

1. **Press ⌘-B to work in the background.**

   The Index button will appear on every card in your stack, so make sure you see stripes in the menu bar.
2. **Choose New Button from the Objects menu and move the new button to any available space in the background.**
3. **Name the button** `Index` **and select Auto Hilite.**

Writing a script that goes through all the cards

You will write the script for the Index button in several stages. First you'll use a repeat structure to go through all the cards in the Collection stack.

Follow these steps:

1. **Click Script in the Index button's Info dialog box.**

   The script editor for the Index button appears.

2. **Type the following script:**

```
on mouseUp
  repeat with countVar = 1 to the number of cards
    go to card countVar
  end repeat
end mouseUp
```

   The contents of a `repeat` structure are automatically indented. The `repeat` and `end repeat` statements should always line up.

3. **Press Enter.**

4. **Return to the card layer and click the Index button with the Browse tool.**

   HyperCard goes to the first card in the stack, then to the second, then to the third, and so on—until it reaches the last card in the stack.

How the script works

The handler uses a `repeat` structure to go through all the cards in the stack. The `repeat` statement uses the `repeat with` form, which has this syntax:

`repeat with` *variable* `=` *startingNumber* `to` *endingNumber*

In this case you've named the variable `countVar`. The starting number is 1 and the ending number is the number of cards in the stack.

The first time through the loop, `countVar` equals 1. Therefore HyperCard evaluates the statement

```
go to card countVar
```

as

```
go to card 1
```

The next time through the loop, `countVar` equals 2, so HyperCard goes to card 2 of the stack. The process continues and HyperCard goes to card 3, and 4, and so on—until `countVar` equals the number of cards in your stack. At that point, the loop finishes and the handler moves on to the next statement, which is `end mouseUp`.

Adding statements that compile the index

Now you will add statements to the script that put information into the index. As you go to each card, you'll put the contents of the Artist field and Title field into a variable. Each time the handler goes to another card, it will put the entry for that card *after* what is already in the variable. In this way the variable will accumulate all of the index entries. Finally the handler will put the contents of the variable into the Entries field.

Follow these steps:

1. **Open the script for the Index button.**

2. **Type the statements shown in bold in the following script:**

```
on mouseUp
  put empty into indexVar
  repeat with countVar = 1 to the number of cards
    go to card countVar
    put bg field "Artist" & "  " & bg field "Title" & Return after indexVar
  end repeat
  go to card "Index"
  put indexVar into card field "Entries"
end mouseUp
```

Make sure everything is spelled correctly and that the statements are in the right order.

3. **Press Enter.**

4. **Try out the Index button.**

   You go to the first card in the stack, then the second, and so on until you reach the end of the stack. Then you go back to the Index card, where a list of recordings appears inside the Entries field.

   ❖ *If something else happened:* Check your spelling and try the script again. Make sure that the names of the Index card and Entries field are spelled correctly and match the names you used in your script. ❖

Each index entry consists of the contents of the Artist field, followed by four spaces and the contents of the Title field.

Because you put a Return character at the end of each entry, all the entries begin on new lines. Some entries may take up more than one line. Entries take up more than one line if they are long and "wrap" onto a second line or if you typed Return characters when you entered text into the Artist and Title fields.

Some finishing touches

The index includes an entry for every card in your stack—including the Index card itself. Because the Index card has nothing in its Artist and Title fields, the entry for the Index card is a blank line. If you have any blank cards in your Collection stack, they also appear as blank lines in the index.

Now you'll add an `if` structure to the script that checks each card to make sure that something has been typed into the Artist field. If the Artist field is blank, the index won't include that card.

You'll also add a `lock screen` command at the beginning of the handler to freeze the screen while HyperCard goes from card to card "behind the scenes."

Follow these steps:

1. **Open the script for the Index button.**

2. **Type the statements shown in bold in the following script:**

```
on mouseUp
  lock screen
  put empty into indexVar
  repeat with countVar = 1 to the number of cards
    go to card countVar
    if bg field "Artist" is not empty then
      put bg field "Artist" & "  " & bg field "Title" & Return after indexVar
    end if
  end repeat
  go to card "Index"
  put indexVar into card field "Entries"
  unlock screen
end mouseUp
```

The `put` statement should indent and the `if` statement should line up with the `end if` statement.

3. **Press Enter.**

4. **Try the Index button.**

After a pause, the Index card appears with the list of recordings in the Entries field. The list should contain only cards for which you typed something into the Artist field.

As you probably guessed, the `lock screen` command locks the screen. When you lock the screen, the screen image won't change until either an `unlock screen` command is executed or all handlers have finished executing. Because HyperCard doesn't have to redraw the screen every time the script goes to another card in your stack, it can compile the Index more quickly.

Adding a keyboard shortcut

Every time you click the Index button, HyperCard recompiles the index for your stack. This process can be time-consuming, especially if your stack contains many cards. Now you'll modify the Index button's script so that by holding down the Option key as you click the button, you go directly to the Index card, without compiling the index.

Follow these steps:

1. **Open the script for the Index button and type the statements shown in bold in the following script:**

```
on mouseUp
  if the optionKey is down then
    go to card "Index"
  else
    lock screen
    put empty into indexVar
    repeat with countVar = 1 to the number of cards
      go to card countVar
      if bg field "Artist" is not empty then
        put bg field "Artist" & "    " & bg field "Title" & Return after indexVar
      end if
    end repeat
    go to card "Index"
    put indexVar into card field "Entries"
    unlock screen
  end if
end mouseUp
```

2. **Press Enter.**
3. **Go to any card other than the Index card.**
4. **While holding down the Option key, click the Index button with the Browse tool.**

   You go directly to the Index card without recompiling the index.

When you click the Index button, HyperCard tests the condition `the optionKey is down`. If the Option key is pressed, you go directly to the Index card. Otherwise HyperCard compiles the index.

Setting properties

The *properties* of a HyperCard object are characteristics of the object that you can set. For example, every button has a `name` property that specifies the name of the button, a `hilite` property that specifies whether the button is highlighted, a `location` property that specifies where the button appears on the screen, and so on.

The characteristics of the overall HyperCard environment are called *global properties*. For example, the `userLevel` is a global property that determines the current user level.

Usually you set properties by choosing options in the object's Info dialog box or on the User Preferences card of the Home stack. However, you can also set properties by using the `set` command.

The syntax of the `set` command is:

`set [the]` *property* `[of` *object*`] to` *expression*

The placeholder *property* is a HyperCard property. (A complete list of properties appears in the appendix.) What *expression* may be depends on the property. Here are some examples of the `set` command with comments describing what each command does:

```
set the hilite of bg button "Quit" to true -- highlights the "Quit" button
set the textFont of bg button "Quit" to "Times" -- changes the button's font to "Times"
set the userLevel to 5 -- sets user level to scripting
set the cursor to "watch" -- changes the cursor to a wristwatch
```

Now you'll add a `set` command to the script for the Index button.

Changing the cursor

Busy cursor

When someone clicks the Index button, there is a momentary delay while HyperCard compiles the index for the stack. In this section you'll write a command that displays the busy cursor or "spinning beach ball" while the index is being generated. By displaying the busy cursor, you let the user of your stack know that HyperCard is still working even though the screen isn't changing.

Follow these steps:

1. **Open the script for the Index button.**
2. **Type the statement shown in bold in the following script:**

```
on mouseUp
  if the optionKey is down then
    go to card "Index"
  else
    lock screen
    put empty into indexVar
    repeat with countVar = 1 to the number of cards
      set the cursor to busy
      go to card countVar
      if bg field "Artist" is not empty then
        put bg field "Artist" & "  " & bg field "Title" & Return after indexVar
      end if
    end repeat
    go to card "Index"
    put indexVar into card field "Entries"
    unlock screen
  end if
end mouseUp
```

3. **Press Enter.**
4. **Click the Index button with the Browse tool. (Don't hold down the Option key.)**

   The pointing finger (Browse tool) cursor changes to the busy cursor. The cursor spins for a moment, the Index card appears, and the cursor changes back to the Browse tool.

The `cursor` is a global property that determines which cursor is displayed on the screen. As soon as all handlers have finished executing, the cursor changes back to the cursor for the current tool—in this case, the Browse tool.

❖ *By the way:* To make the busy cursor spin, the `set cursor to busy` command must be inside a `repeat` structure. ❖

Using functions

HyperTalk contains both commands and functions. A *command* (such as `go` or `put`) carries out an action, whereas a *function* returns a value of some sort. For example, `the time` is a HyperCard function that returns the current time set in your Macintosh. You can use names of functions in commands to get values without having to figure out how to write the formula as part of your handler.

The Appendix contains a complete list of HyperTalk functions. The HyperTalk Reference stack describes how to use each function.

❖ *By the way:* In addition to using HyperTalk's built-in functions, you can make up functions of your own. For instructions on how to write your own functions, see the HyperTalk Reference stack or the *HyperCard Script Language Guide.* ❖

Going from an index entry to a card

In this section you'll write a script for the Entries field. The script will use a HyperTalk function named `the clickLine` to determine which line a user has clicked in the Entries field and find the appropriate card.

You will write this script in stages to get a better idea of how it works. You'll begin by writing a script that tells you which line of the Entries field has been clicked.

Go to the Index card (if you're not already there) and follow these steps:

1. **Choose the Field tool.**
2. **Double-click the Entries field to see its Info dialog box.**
3. **Click the Lock Text and Don't Search options to select them.**

   Selecting the Lock Text option locks the field so you can't type in it. When a field is locked, clicking the field with the Browse tool doesn't place the insertion point in the field; instead, it sends a `mouseUp` message to the field.

Selecting Don't Search tells HyperCard not to search this field when you execute a `find` command. If you are searching for a particular record, you would want to find the card for that record, not its index entry.

4. **Click the Script button and type the following script for the Entries field:**

```
on mouseUp
  put the clickLine
end mouseUp
```

When you're finished, press Enter to close the script editor.

5. **Try out the script by clicking any index entry with the Browse tool.**

A description of the line you clicked appears in the Message box. For example, if you clicked the second line, the message would say:

```
line 2 of card field 1
```

❖ *If something else happened:* Check your spelling and try the script again. Also make sure that the Entries field is locked. ❖

Because the Entries field is locked, clicking the field sends a `mouseUp` message to the field, causing the `mouseUp` handler in the field's script to execute.

The function `the clickLine` returns a description of the line that you click in a field. The statement `put the clickLine` puts that description into the Message box.

The script now knows which line you clicked. But what does that line contain?

6. **Open the script for the Entries field and type the boldface words in the following script:**

```
on mouseUp
  put the value of the clickLine
end mouseUp
```

When you're finished, press Enter.

7. **Click an index entry with the Browse tool.**

The contents of the line you clicked should appear in the Message box.

`The value of` is a function that returns the value of any expression. In this case it returns a text string consisting of the contents of the line you clicked—that is, the index entry for that line.

Now that your handler knows which recording you're interested in, the next step is to find the right card. You'll use HyperCard's `find` command to do that.

8. **Open the script for the Entries field, select the word `put`, and change it to `find`.**

The completed script should look like this:

```
on mouseUp
  find the value of the clickLine
end mouseUp
```

9. **Try out the script by clicking an index entry with the Browse tool.**

If you went to the correct card, congratulations! You're doing great.

The `find` command tells HyperCard to search through the fields in the stack for a specific string of text. In this case, it searches for the index entry that you clicked. (Because you selected the "Don't Search" option for the Entries field, it won't search the Entries field.)

- *If something else happened:* Check your spelling and try the script again. Make sure that the Lock Text and Don't Search options are selected in the Entries field's Info dialog box. ❖

What you've done in this chapter

In this chapter you learned how to use `if` structures and `repeat` structures, how to set properties, and how to use functions. You added a Quit button to your stack, and you wrote a script that compiles an index for your stack and a script that lets you go to a card by clicking an index entry.

Here's a list of the new HyperTalk words you have learned:

Keywords

`if`	Begins an `if` structure.
`then`	Used in `if` structures to mark the beginning of a list of actions to be carried out.
`else`	Used when you want to specify an alternative action in an `if` structure.
`repeat`	Begins a `repeat` structure.

Commands

`doMenu`	Lets you execute a menu command from within a script.
`find`	Searches all the cards in a stack for a text string.
`lock screen`	Prevents HyperCard from updating the screen until an `unlock screen` command is encountered or until all handlers have finished executing.
`set`	Changes the value of HyperCard properties.

Properties

`cursor`	Determines which cursor HyperCard displays.
`userLevel`	Determines the current user level. The possible settings range from 1 (for browsing) to 5 (for scripting).

Functions

`the clickLine`	Returns a description of the line in a field that has been clicked.
`the value`	Returns the value of an expression.
`the optionKey`	Returns the value `down` when the Option key is pressed and `up` when it is not.

Miscellaneous

`empty`	A text string with nothing in it; the same as `""`

Syntax summaries

This section describes the syntax of the commands you used in this chapter, along with the syntax of the `if` and `repeat` keywords.

DoMenu

The `doMenu` command lets you execute any of HyperCard's menu commands from within a script.

Syntax:

`doMenu` *itemName*

ItemName can be the name of a menu command or the name of a desk accessory in the Apple menu.

Include three typed periods if that's how a command is shown in the menu; for instance, `"Card Info..."`. You must type the three periods; don't use the ellipsis character (Option-semicolon).

Examples:

```
doMenu "New Card"
doMenu "Print Stack..."
```

Find

The `find` command searches for a text string in all the card and background fields (visible or not) of the current stack. You can limit the search to a specific background field by specifying a field.

Syntax:

`find` *text* `[in` *field* `]`

Text can be any text string. *Field* is an expression that identifies a background field.

When HyperCard finds a word beginning with *text*, it stops searching and places a rectangle around the word.

Examples:

```
find "Moz"
find "Mozart" in background field "Artist"
```

If

The `if` keyword is used to begin an `if` structure. An `if` structure tests a condition and then executes one or more statements if the condition is true. If the condition is false, statements following the optional `else` keyword are executed.

Syntax:

`if` *trueOrFalse* `then` *statement*

`if` *trueOrFalse* `then` *statement* `else` *statement*

`if` *trueOrFalse* `then`
  statements
`else`
  statements
`end if`

`if` *trueOrFalse* `then`
  statements
`end if`

TrueOrFalse is an expression that evaluates to either `true` or `false`. *Statement* is a single HyperTalk statement. *Statements* can be one or more statements.

Example:

```
if Response = "Correct" then
  answer "That's correct!"
  else
    answer "Sorry, try again."
end if
```

Lock screen and unlock screen

The `lock screen` command prevents HyperCard from updating the screen until HyperCard encounters an `unlock screen` command or all handlers have finished executing.

Syntax:

```
lock screen
unlock screen
unlock screen with visualEffect
```

VisualEffect is any of the forms of the `visual` command.

Examples:

```
lock screen
unlock screen with visual effect zoom out slowly
```

Repeat

A `repeat` statement is used to identify the first line of a `repeat` structure.

Syntax:

```
repeat [forever]
```

This loop repeats forever, or until an `exit` statement is encountered.

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

Number specifies how many times the loop is executed.

```
repeat until trueOrFalse
repeat while trueOrFalse
```

TrueOrFalse is an expression that evaluates to true or false. The `repeat until` loop is repeated as long as *trueOrFalse* is false. The `repeat while` loop is repeated as long as *trueOrFalse* is true.

`repeat with` *variable* = *start* `to` *finish*
`repeat with` *variable* = *start* `down to` *finish*

Variable is a variable name, and *start* and *finish* are integers. The value of *start* is assigned to *variable* at the beginning of the loop, and is increased by 1 with each pass through the loop. In the `down to` form, the value of *variable* is decreased by 1 with each pass through the loop. Execution ends when the value of *variable* equals the value of *finish*.

Example:

```
repeat for 100 times
  add 1 to message box
end repeat
```

Set

The `set` command allows you to change various HyperCard properties from within a script.

Syntax:

`set [the]` *property* `[of` *object*`] to` *expression*

Property stands for a changeable characteristic of the HyperCard environment or of an object.

Object is an identifier for an object, such as its number, ID, or name.

What *expression* is depends on the property. Some properties, such as `hilite`, have the values `true` or `false`. Others, such as `userLevel`, have numeric values. Still others—such as the `name` property of a button—have a string of characters as their value.

Examples:

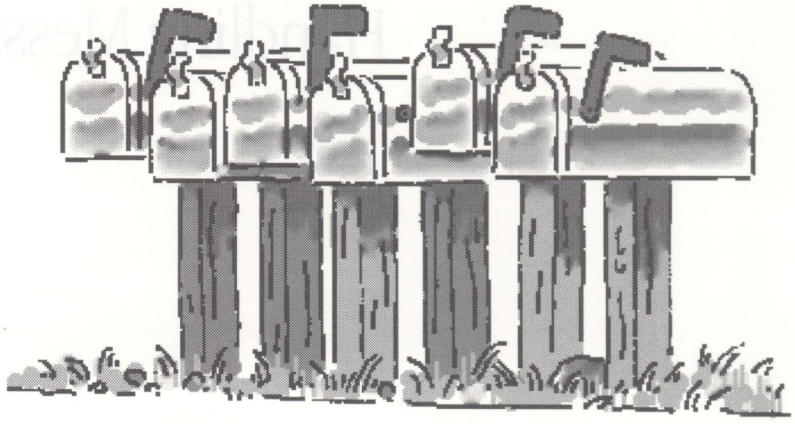
```
set the userLevel to 5
set the hilite of card button 1 to true
set the name of card field 1 to "Horse"
```

Handling Messages

As you know, a *message handler* is a group of HyperTalk statements beginning with an `on` statement, such as `on mouseUp` and ending with an `end` statement. All the scripts you've written so far contain only one message handler, but scripts often contain more than one handler.

In this chapter you'll write new handlers and explore the way messages travel between objects. You will add another feature to your Collection stack—a button that plays sound when you click it.

If you took a break at the end of Chapter 3, start up HyperCard and go to your Collection stack before you go on.



How messages travel

HyperCard can send system messages to a button, a field, or the current card. For example, if you click a button, HyperCard sends a `mouseUp` message to the button. If you click a locked field, HyperCard sends `mouseUp` to the field. If you click anywhere else on a card, HyperCard sends `mouseUp` directly to the card.

A message can travel from one HyperCard object to another—until it is handled. For example, when someone clicks a button, a `mouseUp` message is sent to the button. If the button's script doesn't have a handler for `mouseUp`, the message is passed to the current card, then to the background, then to the stack, then to the Home stack, and finally to HyperCard itself. This sequence is called the *message-passing order;* it's illustrated in Figure 4-1.

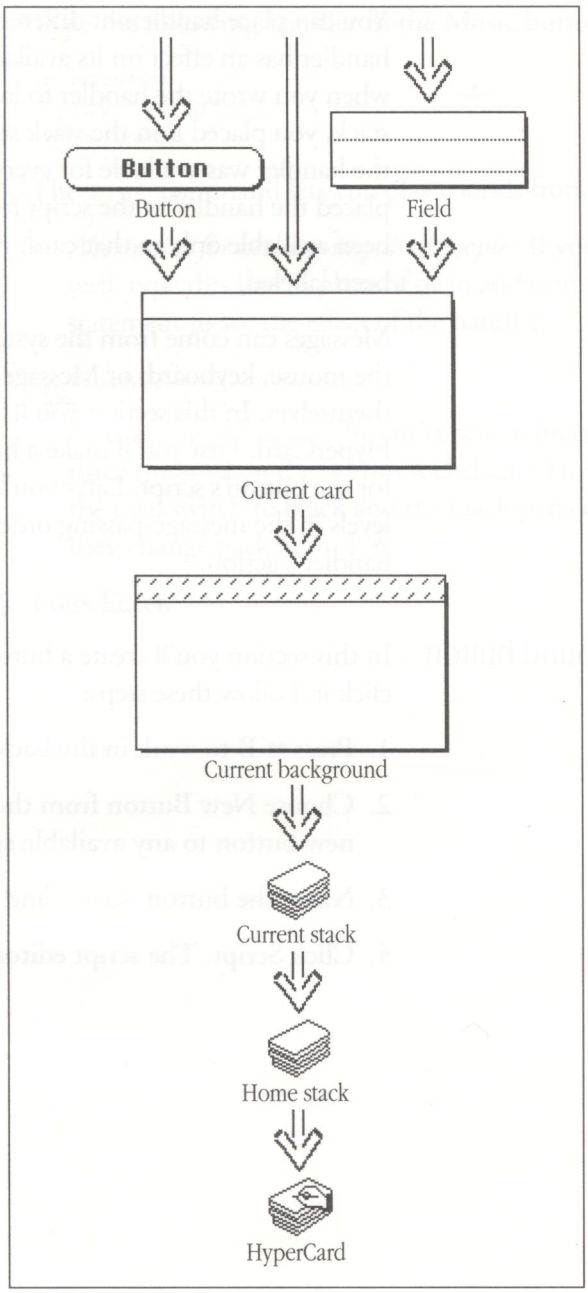


Figure 4-1 The message-passing order

You can place handlers at different levels. Where you place a handler has an effect on its availability. For example, in Chapter 2, when you wrote the handler to label all cards of the Collection stack, you placed it in the stack script; that placement meant that the handler was available for every card in the stack. If you had placed the handler in the script for one of the cards, it would have been available only to that card; no other cards would have been labeled.

Messages can come from the system; from your actions with the mouse, keyboard, or Message box; or even from handlers themselves. In this section you'll see how messages move around in HyperCard. First you'll make a button and write a message handler for the button's script. Later you'll move the handler to different levels in the message-passing order and observe the difference in the handler's action.

Creating a Sound button

In this section you'll create a button that plays sound when you click it. Follow these steps:

1. **Press ⌘-B to work in the background.**
2. **Choose New Button from the Objects menu and move the new button to any available space in the background.**
3. **Name the button `Sound` and select Auto Hilite.**
4. **Click Script. The script editor appears.**

7. **Type the following handler in the Music button's script:**

```
on mouseUp
  play "boing"
end mouseUp
```

The `play` command lets you play sounds from within scripts.

❖ *Alternative for hearing-impaired people:* If you can't hear well, type this line in place of or in addition to the `play` statement to see the effect of the handler:

```
flash 3
```

This command causes the entire screen image to flash rapidly three times when the button is clicked. (The white parts of the card switch to black and the black parts to white; then they change back again.) ❖

6. **Press Enter.**

7. **Press ⌘-B to return to the card layer and click the Sound button with the Browse tool.**

   You hear a "boing" sound

   ❖ *If something else happened:* Check the script's spelling and make sure you have included quotation marks in the right places. If the script is correct, make sure you have the Speaker Volume in the Control Panel set to a value greater than zero. ❖

When you click the button, a `mouseUp` message is sent to the button. This causes the `mouseUp` handler to execute, and the "boing".

Moving the handler to the card level

Where you place a handler in HyperCard affects its action. A handler at the "top" level, namely, in a button script or a field script, can respond only to a message received by that button or field. The same handler further "down" in the message-passing order—that is, at the card, background, or stack level—can respond to a the message sent by any objects higher up, unless those objects intercept the message with their own handlers. (See Figure 4-1, earlier in this chapter.)

What the message-passing order means to you is that you can control whether your handlers act very locally—only for a particular button, for example—or more globally, for an entire card, background, or stack.

In this section you'll move the `mouseUp` handler of the Sound button to a different level in the message-passing order to experience the change in its response.

First notice that the handler works only if you click the Sound button. If you click anywhere else on the card, you won't hear tanything.

The next step is to move the handler to the script for one particular card. You'll cut the `mouseUp` handler from the Sound button's script and paste it into the script for the Index card that you created in Chapter 3. Follow these steps:

1. **Go to the Index card.**
2. **While pressing the ⌘ and Option keys, click the Sound button with the Browse tool.**

   You see the script for the Sound button.
3. **Select the `mouseUp` handler.**

   Drag the mouse across the entire handler to select it.
4. **Choose Cut from the Edit menu to cut the handler and place it on the Clipboard.**

   The script editor should now have nothing in it. If you still see the handler there, try steps 2 and 3 again. Every object has a script, even if there's nothing in it. Scripts with nothing in them are called empty scripts.
5. **Press Enter to close the script for the Sound button.**

Now you'll open the script for the Index card.

6. **Choose Card Info from the Objects menu.**

   The Card Info dialog box appears.
7. **Click Script.**

   The script for the Index card appears.
8. **Choose Paste from the Edit menu to paste the handler into the script for the Index card.**

   The `mouseUp` handler appears in the script for the Index card.
9. **Press Enter to close the script for the Index card.**

Trying out the card script

Now test the effects of moving the handler to the card level.

1. **Click the Sound button with the Browse tool.**

   The "boing" plays just as it did before. The `mouseUp` message passes through the empty button script and goes on to the card script.

2. **Click anywhere else on the card (except on a button or field).**

   The "boing" plays because whenever you click the card, `mouseUp` goes directly to the card, which now contains the handler for `mouseUp` in its script.

3. **Click an entry in the index.**

   You go to the card for the entry you clicked, as usual—and you don't hear the boing.

   The reason you don't hear the boing when you click the Entries field is that the field has a `mouseUp` handler in its script. The `mouseUp` message is intercepted by the field script, so the `mouseUp` handler in the card script doesn't execute.

Now that you are on a card other than the Index card, notice what happens when you click the card.

4. **Click anywhere on the card (except on a button or field with a `mouseUp` handler).**

   Nothing happens because there is no `mouseUp` handler in this card's script.

Moving the handler to the background level

Now you'll take the handler out of the card script and move it to the background script:

1. **Go to the Index card.**
2. **Choose Card Info from the Objects menu.**

   The Card Info dialog box appears.
3. **Click Script to see the script editor.**

   The script for the Index card appears.

   ❖ *Keyboard shortcut:* You can press ⌘-Option-C to see the script editor of the current card without having to go through the card's Info dialog box. ❖
4. **Drag the mouse across the entire handler to select it.**
5. **Choose Cut from the Edit menu to cut the script and place it on the Clipboard.**

   The card script should now be empty.
6. **Press Enter to close the card script.**
7. **Choose Bkgnd Info from the Objects menu.**

   The Background Info dialog box appears.
8. **Click Script.**

   The script for the current background appears.

   ❖ *Keyboard shortcut:* You can press ⌘-Option-B to see the script for the current background. ❖
9. **Choose Paste from the Edit menu to paste the handler into the background script.**
10. **Press Enter to close the background script.**

Trying out the background script

Now test the effects of moving the handler to the background level.

1. **Click the Sound button with the Browse tool.**

   You hear the "boing" play. The `mouseUp` message passes through the empty button script and empty card script to the background script, which now contains the handler.

2. **Click anywhere else on the card (except on a button or field with a `mouseUp` handler).**

   The `mouseUp` message passes to the background script and the boing plays.

3. **Go to any other card in the stack and click anywhere on the card (except on a button or field with a `mouseUp` handler).**

   You should still hear the "boing" splay. The handler is now available to any card sharing the background.

If you moved the handler to the stack level, the same thing would happen because the Collection stack has only one background; however, if a stack has more than one background, only a handler at the stack level or above would be available to all cards of all backgrounds.

Handlers calling handlers

All the handlers you've written so far respond to system messages sent by HyperCard (such as `mouseUp` and `openCard`). HyperCard sends system messages in response to events such as mouse clicks, keyboard actions, and the creation or deletion of objects. (The appendix contains a list of all HyperCard system messages.) But there are other ways for handlers to "get the message."

Each time HyperCard executes a statement within a handler, it sends that statement as a message. A message sent from one handler can cause another handler to execute. It's as though the handlers are talking to each other—with one handler telling the other to begin executing.

In this section you'll write a handler that "calls" another handler. First you'll write a handler that sends a message, then you'll write a handler that responds to that message.

Writing the "calling" handler

You will write a script for the Sound button so that a message named `playSound` is sent whenever someone clicks the button. Later you'll change the `mouseUp` handler in the background script so that it responds to the `playSound` message. Follow these steps:

1. **Open the script for the Sound button.**
2. **Type** `playSound`

   The completed script should look like this:

   ```
   on mouseUp
     playSound
   end mouseUp
   ```

   In English, the script says, "When someone clicks this button, send a message named `playSound`. That's all."

   The message name `playSound` is just a made-up word. You could use any other word (except a HyperTalk keyword); this name seems appropriate because it describes the action of the handler.

   ❖ *Alternative for hearing-impaired people:* If you are using the `flash 3` alternative instead of the "boing", you could use a different name, such as `razzleDazzle` (but don't use `flash`). Be sure, however, that you use your alternative name in the steps that follow. ❖
3. **Press Enter.**

You will need to write a handler that handles the `playSound` message. But for now, see how the script works so far.

4. **Click the Sound button with the Browse tool.**

   You see a "Can't understand" dialog box. HyperCard can't understand the `playSound` message because it can't find a `playSound` handler anywhere. In other words, there is no handler that begins with the statement `on playSound`.

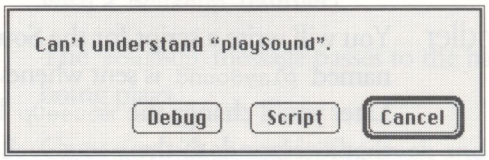


Figure 4-2 "Can't understand" dialog box

5. **Click Cancel to close the "Can't understand" dialog box.**

❖ *By the way:* Clicking the Debug button opens the script editor window and puts a box around the statement HyperCard can't understand. It also adds a new menu, named the Debugger menu, which is sometimes helpful in figuring out where mistakes or "bugs" are in a script. For more information about the Debugger menu, see the *HyperCard Script Language Guide.* ❖

Writing the "called" handler

Now you'll create a handler that responds to the `playSound` message that's sent when someone clicks the Sound button. You could write a handler from scratch, but in this case you'll simply change the `mouseUp` handler in the background script to a `playSound` handler.

1. **Choose Bkgnd Info from the Objects menu.**
2. **Click the Script button.**

   The script for the background appears.

3. **Select the word `mouseUp` in the first line of the handler.**

   Drag across the word as you would when selecting any text, or just double-click the word.

4. **Replace it by typing the word `playSound`**

5. **Select the word `mouseUp` in the last line of the handler amd replace it by typing the word `playSound`**

   The completed handler looks like this:

```
on playSound
  play "boing"
end playSound
```

   The name used after `on` must match the name after `end`.

You have now changed the handler from a `mouseUp` handler to a `playSound` handler. It now responds to the message `playSound` instead of the message `mouseUp`.

6. **Press Enter.**

You have created a handler that sends a message named `playTune`, as well as a handler that responds to `playTune`. Now see how the two handlers work together.

7. **Click the Sound button with the Browse tool.**

   When the Sound button receives `mouseUp`, its handler in turn sends the message `playSound`. That message goes through the message-passing order until it's intercepted by the `playSound` handler in the background script. The `playSound` handler executes, and you hear the `"boing."` Figure 4-3 shows the path taken by the `playSound` message.

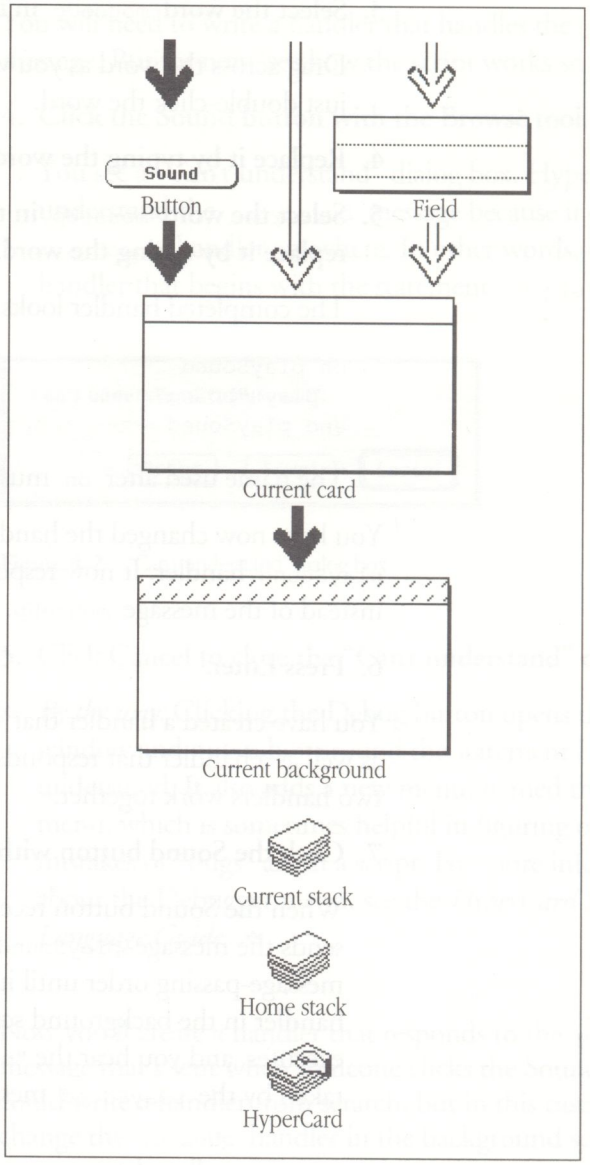


Figure 4-3 A message passing from a button to the background script

Clicking anywhere else on the card won't cause the sound to play, because the background handler isn't a `mouseUp` handler any more.

In this section you've essentially defined a new command, which is named `playSound`. The `playSound` command plays a boing. That's really all there is to defining your own commands. Think of what you want a command to do, think of a name for it, and write a handler that uses the name after `on` and `end`, with the appropriate HyperTalk statements in between. To make the command work, send the command's name as a message to the object that has the handler in its script.

❖ *By the way:* It's generally best to avoid using the name of an existing HyperTalk command or function as the name of a command you create. See the *HyperCard Script Language Guide* for details on naming commands. ❖

Intercepting a message

When HyperCard sends a statement within a handler as a message, the message goes first to the object that contains the handler being executed. (For example, when the Sound button sends a `playSound` message, the message first goes to the button itself.) If the object's script doesn't have a handler for the message, the message travels next to the current card. If the script for the current card doesn't have an appropriate handler, the message continues through the message-passing order, as shown in Figure 4-1.

Once a message is handled, it does not continue passing through the message-passing order. Therefore it's possible for an object at the "top" of the message-passing order to intercept a message before the message can travel to objects at the "bottom."

In this section, you'll write a `playSound` handler for the script of the Index card. This card-level handler will make the Sound button play a different sound when you're on the Index card.

Follow these steps to write the script:

1. **Go to the Index card.**
2. **Choose Card Info from the Objects menu; then click the Script button to see the card's script.**

   Or press ⌘-Option-C.

3. **Type the following handler:**

```
on playSound
  play "harpsichord" "c e g"
end playSound
```

   This handler plays three notes using the `harpsichord` sound.

4. **Press Enter.**
5. **Click the Sound button with the Browse tool.**

   The `playSound` handler in the card script executes, and you hear the three harpsichord notes.

6. **Go to any other card in the stack and click the Sound button.**

   The `playSound` handler in the card script executes, and you hear the "boing".

How the handlers work

When you click the Sound button the button's `mouseUp` handler sends a `playSound` message. Because there is no `playSound` handler in the button's script, the message passes to the script for the current card.

When the Index card is the current card, the `playSound` handler in the card script handles the `playSound` message. The card script intercepts the message before it can pass to the background script. Figure 4-4 shows the path taken by the `playSound` message when the Index card is the current card.

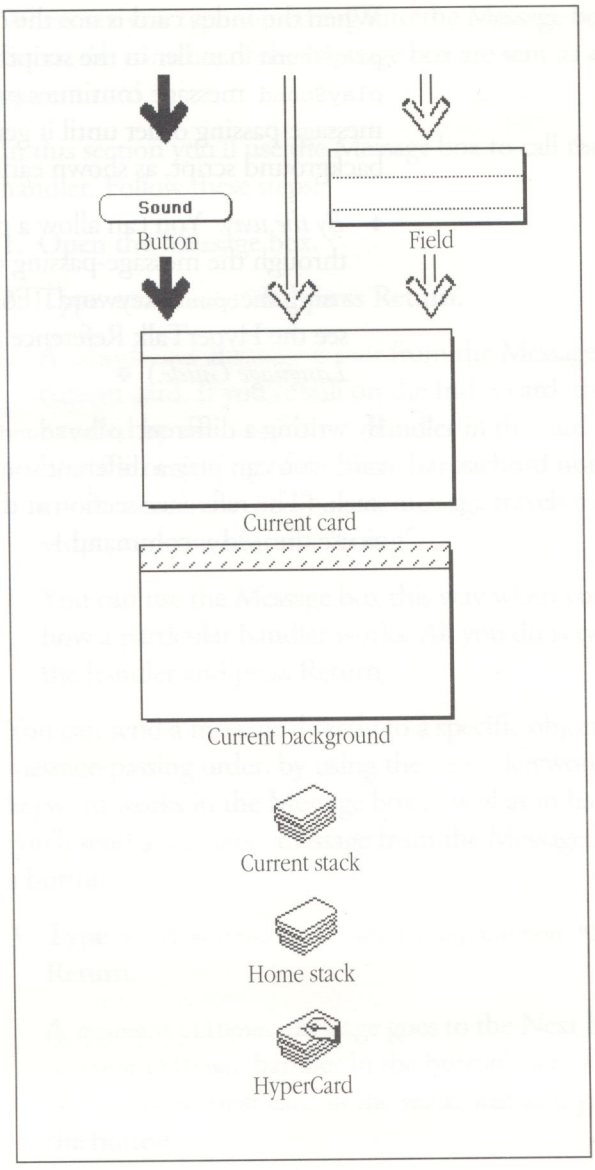


Figure 4-4 A message being intercepted by the Index card's script.

When the Index card is not the current card (and there is no `playSound` handler in the script for the current card), the `playSound` message continues passing from object to object in the message-passing order until it gets to the `playSound` handler in the background script, as shown earlier in Figure 4-3.

❖ *By the way:* You can allow a message to continue passing through the message-passing order after it has been handled by using the `pass` keyword. (For more information about `pass`, see the HyperTalk Reference stack or the *HyperCard Script Language Guide.*) ❖

By writing a different `playSound` handler for the script of each card, you can play a different sound on each card in your Collection stack. (The reference section at the end of this chapter explains how to use the `play` command.)

Calling handlers from the Message box

Whenever you type something into the Message box and press Enter, the contents of the Message box are sent as a message to the current card.

In this section you'll use the Message box to call the `playSound` handler. Follow these steps:

1. **Open the Message box.**

2. **Type `playSound` and press Return.**

   A `playSound` message is sent from the Message box to the current card. If you're still on the Index card, the message is handled by the `playSound` handler in the card script, and you hear the beginning of the three harpsichord notes. If you're on another card, the `playSound` message travels to the background script, and you hear the "boing".

   You can use the Message box this way when you want to test how a particular handler works. All you do is type the name of the handler and press Return.

You can send a message directly to a specific object, bypassing the message-passing order, by using the `send` keyword. The `send` keyword works in the Message box as well as in handlers. Now you'll send a `mouseUp` message from the Message box directly to a button:

3. **Type `send mouseStillDown to bg button "Next"` and press Return.**

   A `mouseStillDown` message goes to the Next button. The `mouseStillDown` handler in the button's script executes, and you go to the next card in the stack, just as if you had clicked the button.

The `send` keyword lets you send messages against the normal flow of the message-passing order—for example, from a stack script to a button or from one button to another button.

4. **Close the Message box.**

Handlers as building blocks

In some ways getting things done in HyperTalk is no different from getting things done in everyday life. When you want to perform a complex procedure, you can divide the procedure into smaller, more easily manageable parts. These smaller parts of a complex procedure are sometimes called *subprocedures.*

For example, suppose you want to cook spaghetti with meat sauce. You might divide the main procedure, "cook spaghetti," into three subprocedures: "cook pasta," "cook sauce," and "add sauce to pasta." If you could describe the procedure of making spaghetti as a HyperTalk script, it would look something like this:

```
on makeSpaghetti
  cookPasta
  cookSauce
  addSauceToPasta
end makeSpaghetti
```

The handler for the main procedure (`makeSpaghetti`) calls handlers for three subprocedures (`cookPasta`, `cookSauce`, and `addSauceToPasta`).

HyperCard handlers can be used as subprocedures in much the same way. 0Understanding how handlers can call other handlers will be a big help to you as you learn to write longer, more complex scripts.

What you've done in this chapter

In this chapter you have demonstrated the three ways that HyperCard can send messages:

- *System messages* (such as `mouseUp`) are sent in response to some event, such as a mouse or keyboard action.
- *Statements within handlers* (such as `playSound`) are sent when the statements are executed.
- *The contents of the Message box* are sent when you type something in and press Return.

You've learned how a message handler can "call" other handlers, how messages can travel from one object to another, and how handlers can be used as subprocedures.

Here's a list of the new HyperTalk words you have learned:

Commands

`play`	Causes sounds to play.

Keywords

`send`	Sends messages directly to objects.

Miscellaneous

`harpsichord`	Names of sounds used with the `play` command.
`boing`	

Syntax summaries

This section describes the syntax of the `play` command and the `send` keyword.

Play

The `play` command lets you play sounds from within a script or from the Message box.

Syntax:

```
play sound
play sound notes
play sound tempo positiveInteger
play sound tempo positiveInteger notes
play stop
```

Sound is `harpsichord` or `boing`—which are included with HyperCard—or the name of a digitized sound from some outside source.

You can set the speed of play by including the word `tempo` followed by a positive integer. The value 100 is a medium speed; higher numbers play faster. If you don't specify a tempo, `tempo 100` is assumed.

Tempo is the word `tempo` followed by a positive integer that sets the speed of play. The value 100 is a medium speed; higher numbers play faster. If you don't specify a tempo, `tempo 100` is assumed.

Notes make up the melody sequence. Notes are represented by the letters A through G. Rests (or pauses) are represented by the letter R.

If you don't specify any notes, HyperCard plays a single note in the sound you specify. You should include quotation marks around the sound and the notes.

You can include further modifiers after the note name, such as an accidental (a sharp or flat), an octave specification, and a duration code. Here's the syntax for a note:

noteName `[`*accidental*`]` `[`*octave*`]` `[`*duration*`]`

Accidental is either `#` for sharp or `b` for flat.

Octave is a whole number that specifies the pitch range. For example, `g#4` would be the G-sharp note in the middle range, or what musicians call the *middle-C octave*. Higher numbers give higher ranges, and vice versa. If you don't specify a number, HyperCard uses 4.

Duration is a letter code indicating how long to hold the note before the next note sounds. Here are the codes for note duration:

`w`	whole note (four counts)
`h`	half (two counts)
`q`	quarter (one count)
`e`	eighth (one-half count)
`s`	16th (one-fourth count)
`t`	32nd (one-eighth count)
`x`	64th (one-sixteenth count)

If you don't specify a duration code, HyperCard assumes a quarter note.

A period (.) after the duration code indicates a dotted note, which means a note with a duration value of half again as much; that is, `w.` would indicate six counts (four plus half of four).

The codes for octave and duration carry over to subsequent notes unless you change them; this feature saves you from having to type numbers and letters over and over.

Here are some examples of notes with modifiers:

Note specification	Meaning
`d#5w`	D-sharp above high C held for four counts
`Bb4q`	B-flat above middle C held for one count
`e5h.`	E above high C held for three counts (because of the period after the duration code `h`)

Example:

```
play "boing" tempo 300 "ch d#e ge c5w"
```

Dealing with long lines

You can put a long sequence of notes into a script; however, the script editor doesn't wrap lines or let you scroll to see lines that extend beyond the window. You can press Return or Option-Return to wrap a long line temporarily while you type the notes; however, if you use this method you *must* delete the Returns to "unwrap" the lines when you're finished. If you don't the script won't work properly. HyperCard doesn't understand a line break of any sort inside quotation marks.

Another alternative is to wrap a long line permanently by inserting a closing quotation mark and the double ampersand (`&&`) followed by an Option-Return (¬):

```
play "harpsichord" "c3 d e f g a b c4 d e" &&" ¬
"f g a b c5 d e f g a b c6 d e f g a b c7"
```

Notice that you must also begin the wrapped line with a quotation mark.

Send

The `send` keyword directs a message to any object in the current stack or to another stack, but not to a specific object in another stack. It sends a message directly to the specified object, bypassing any other objects in the usual message-passing hierarchy.

Syntax:

```
send "messageName"
send "messageName" [to object]
```

The quotation marks around the name of the message aren't needed if the message is a single word, like `mouseUp`.

Object is an identifier for any object, such as its number, ID, or name. If you use the name, you must enclose it in quotation marks.

Example:

```
send "mouseUp" to background button "About"
```

More Scripting Ideas

As you built your Collection stack, you learned some of the basic methods you can use for HyperTalk scripting. In this chapter you'll learn other ways of using scripts in stacks.

This chapter explains how to modify the Collection stack for other purposes. It also describes some other simple stacks you can build—including a presentation stack, animation stacks, and a stack for fun—and explains the basic steps involved in building and scripting these stacks. You can try building the stacks if you wish, or you can use them as a source of ideas for creating stacks on your own.

Customizing your Collection stack

You can easily modify the Collection stack to catalog things other than records. For instance, you could modify the stack along the lines shown in Figure 5-1.

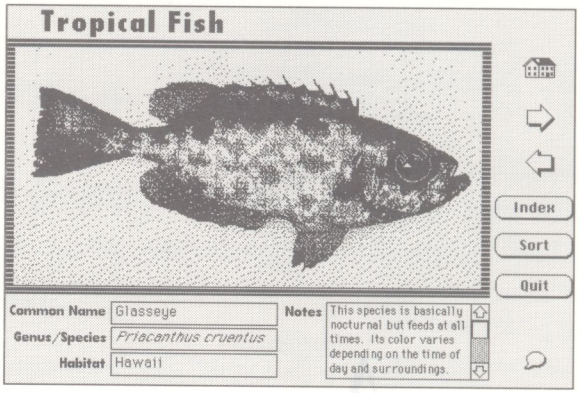


Figure 5-1 Another variation on the records stack

To modify the Collection stack for some other purpose, follow these basic steps:

1. **Save a copy of the Collection stack by choosing Save a Copy from the File menu.**
2. **Change the names of the Category, Artist, and Title fields to better indicate the new contents of the fields.**
3. **Change the scripts of the Sort and Index buttons, replacing all references to the Category, Artist, and Title fields with the new field names.**

You might also want to substititute more appropriate graphics.

Presentation stacks

You can use HyperCard to combine text, graphics, animation and sound into a dazzling presentation. This section shows you how to create a basic presentation stack. You can fill in the contents of the presentation (and the dazzle) yourself.

There are many ways you can organize a presentation. One way is to tell a story from beginning to end by having users go forward or backward from card to card. In most stacks, though, users have opportunities to branch to different parts of the stack, depending on what interests them. (See the *HyperCard Stack Design Guidelines*, published by Addison-Wesley, for a discussion of different ways to structure a stack and how to make stacks easy to navigate.)

The stack described in this section uses a simple tree structure that users can easily navigate. The first card of the stack lists the topics of the presentation. A user chooses a topic of interest by clicking a button. Once a topic has been chosen, the user can navigate through a series of cards about that topic. The user can also return to the list of topics at any time. Figure 5-2 illustrates a stack with a tree structure.

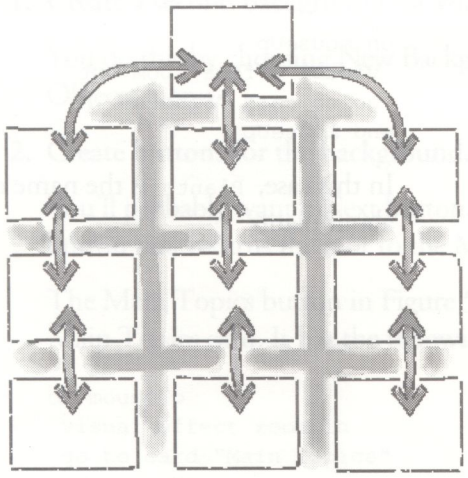


Figure 5-2 Stack with a tree structure

Creating a Main Topics card

Here is an example of a Main Topics card.

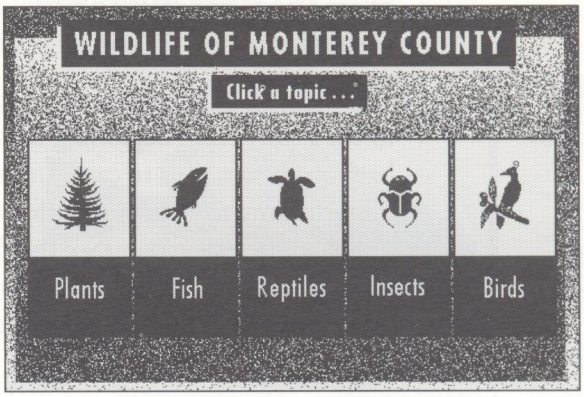


Figure 5-3 Main Topics card with art the user can click

The card shows several pictures, each corresponding to a topic the user can pick. Each picture is covered with a transparent button, which takes the user to a card about the chosen topic. For example, the button covering the picture of a tree has this script:

```
on mouseUp
 visual effect zoom out
 go to card "Plants"
end mouseUp
```

In this case, `Plants` is the name of the first card in a series of cards about plants.

Creating cards about a topic

Once you have decided what the topics of your presentation are, you can create a series of cards about each topic. Figure 5-4 shows an example of a card about a topic.

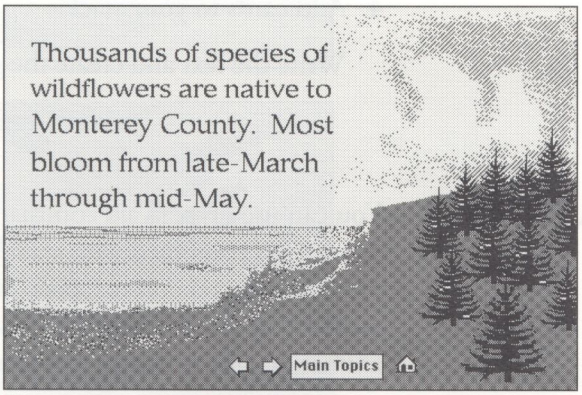


Figure 5-4 A card about a topic

You can create cards about a topic by following these basic steps:

1. **Create a second background for your stack.**

   You do this by choosing New Background from the Objects menu.

2. **Create buttons for the background.**

   You'll probably want a Next button, a Previous button, and a button that returns the user to the Main Topics card.

   The Main Topics button in Figure 5-4 takes the user back to the Main Topics card. It has the following script.

```
on mouseUp
 visual effect zoom in
 go to card "Main Topics"
end mouseUp
```

3. **Create background fields.**

   You will probably want a field for the the heading on each card, as well as a field for the text.

4. **Add cards to your stack.**

   Write the text and create the graphics for your presentation.

Animation

You can use HyperTalk commands to create animation effects. Animation combined with visual effects and sound can turn a presentation, a demonstration, or a training stack into an exciting multimedia production. This section explains two ways to create animation effects using HyperTalk commands.

Animating a series of cards

Creating animation on a series of cards is easy. You put slightly different images on successive cards and then show the cards rapidly—creating the appearance of movement. Figure 5-5 shows an example of a multiple-card animation sequence.

You can practice creating an animation sequence by following these steps:

1. **Create a new stack.**

   Name the stack Animation or any other name you'd like.

2. **Add a few cards to the stack, and create graphics for each card.**

   You can paint your own graphics, or you can copy them from the Art Bits stack.

   Each card should look slightly different from the card before it. To create each card, copy the image from the previous card; then change the image by moving graphics or adding graphics to the card.

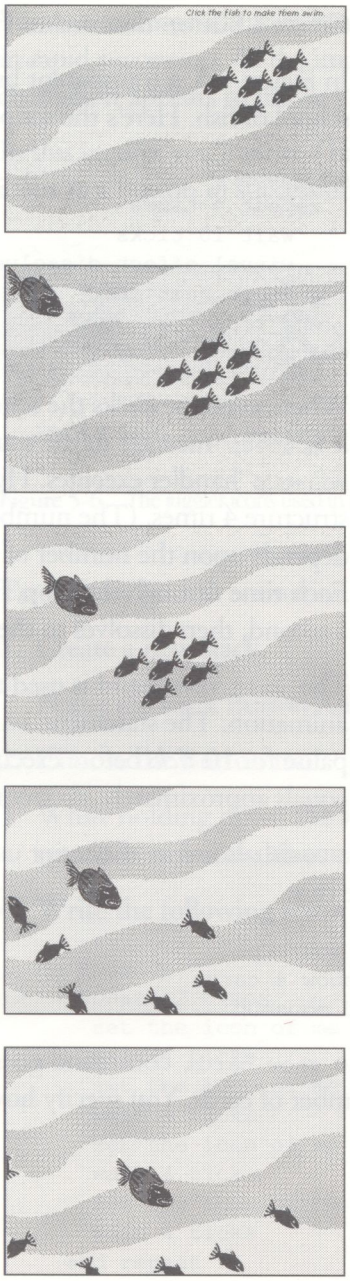


Figure 5-5 Example of multiple-card animation

3. **Create a button that makes HyperCard flip through the cards.**

   In Figure 5-5, a transparent button has been placed on top of the school of fish. Here's the script for the button:

```
on mouseUp
  repeat 4 times
    wait 10 ticks
    visual effect dissolve very fast
    go to next card
  end repeat
end mouseUp
```

   When someone clicks the school of fish, HyperCard sends a `mouseUp` message to the transparent button and the button's `mouseUp` handler executes. HyperCard loops through a repeat structure 4 times. (The number of times through the loop depends upon the number of cards in the animation sequence.) Each time through the loop, HyperCard pauses for a fraction of a second, then dissolves to the next card.

   The `wait` command is used to control the speed of the animation. The statement `wait 10 ticks` tells HyperCard to pause for 10 *ticks* before executing the next command. (A tick equals approximately 1⁄60 second.)

You could also write the script using the `show cards` command, like this:

```
on mouseUp
  show 4 cards
end mouseUp
```

The `show cards` command goes rapidly through a specified number of cards. You specify how many cards you want to show.

Animating an icon

You can animate a button by assigning it a series of different icons in rapid succession. As the image on the button (the button's icon) changes, it appears to move.

In this section you'll write a handler for assigning the following four icons to a button in succession:

Figure 5-6 The juggler icons used in the button animation exercise

To see how button animation works, follow these steps:

1. **Create a new stack.**

   Name the stack Button Animation.

2. **Choose the Button tool.**

3. **While holding down the ⌘ key, drag diagonally to create a transparent button about one inch square.**

4. **Write the following script for the button:**

```
on mouseUp
  repeat 12 times
    set the icon of me to "Juggler 4"
    wait 4 ticks
    set the icon of me to "Juggler 3"
    wait 4 ticks
    set the icon of me to "Juggler 2"
    wait 4 ticks
    set the icon of me to "Practice Stack"
    wait 4 ticks
  end repeat
end mouseUp
```

5. **Press Enter.**

6. **Click the button with the Browse tool.**

   The button comes to life as its icon changes quickly from one version of the juggler to another.

Each `set` command in this handler assigns a different juggler icon to the button. These commands refer to the icons by name, but you can also reference an icon by its ID.

The statement `wait 4 ticks` tells HyperCard to pause for 4/60 second before executing the next command.

The juggler icons come with HyperCard. You can see them by clicking Icon in the Button Info dialog box, then scrolling through the list in the Icon dialog box. To find out an icon's name or ID, click the icon to select it in the Icon dialog box—its name and ID appear at the top of the box.

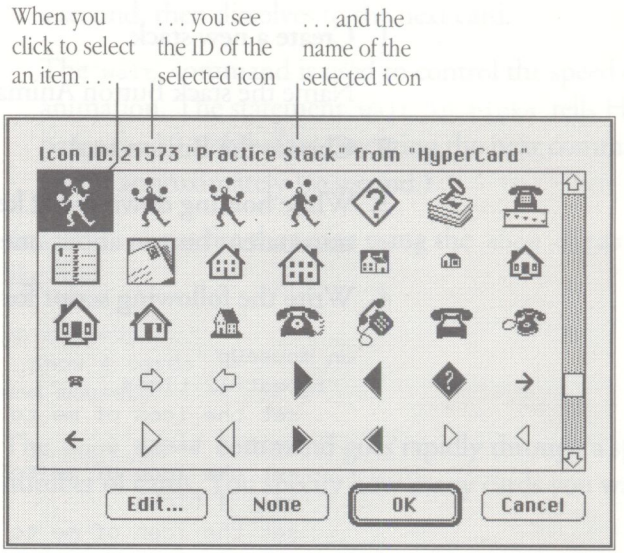


Figure 5-7 Finding out the name and ID of an icon in the Icon dialog box

Using the Icon Editor, you can create your own sets of icons and use them to animate buttons. A button animation works best when each icon in the set looks slightly different from the one before it. To create each icon, you can duplicate the previous icon and then change the image slightly using the commands and options in the Icon Editor. The *HyperCard Reference* explains how to create and modify icons with the Icon Editor.

Animation using Paint tools

Anything you can do with a HyperCard menu command you can also accomplish with a HyperTalk command in a handler. You can achieve an animated effect by writing a handler to select a picture and move it.

In this section you'll learn how to animate an image on a card. You'll create a practice stack, paint an image, and write a script that causes the image to spin.

Making something to animate

The first step is to create a graphic image to animate. Follow these steps:

1. **Create a new stack.**

   Name the stack Paint Animation or any other name you prefer.

2. **Paint a picture on the first card of the stack.**

   You can create the picture yourself using the Paint tools or simply paste some clip art onto the card The image can be simple, like the one shown in Figure 5-6. Make sure you paint the image in the card layer, not in the background.

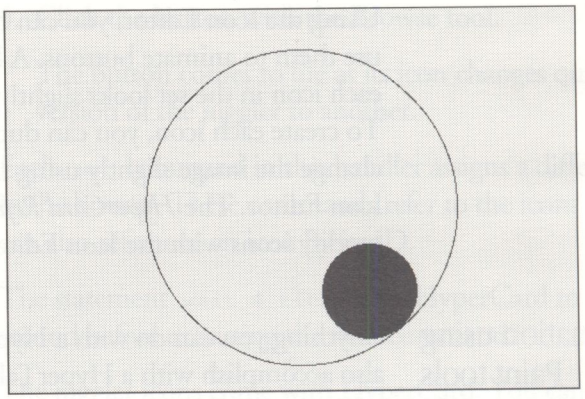


Figure 5-8 A picture that you can spin

To cause this picture to rotate, you could select it and choose Rotate Left or Rotate Right from the Paint menu. However, that action would rotate the image 90° only once. To make it spin, you'd have to choose a Rotate command repeatedly. You can have HyperCard perform this action with a HyperTalk handler.

You can select the image from a handler by using the `choose` command to choose the Selection tool and then the `drag` command to drag across the image. But first, you need to know the starting point and ending point for dragging.

Finding the starting point

If you were going to drag across the image to select it, you would probably begin by positioning the pointer above and to the left of the image. That point would be your starting point.

1. **Press ⌘-M to see the Message box.**
2. **Click a point on the card above and to the left of the graphic image.**

   Be sure to click somewhere higher than the top of the image and farther to the left than the left edge of the image.

3. **Type `the clickLoc` into the Message box.**

   The word `clickLoc` is short for "click location." You must include the word `the`.

4. **Press Return.**

Two numbers appear in the Message box. These numbers represent the horizontal and vertical position of the point where you last clicked on the screen. The first number tells you how far the point is from the left edge of the card window, and the second number tells you how far it is from the top of the window. The distances are measured in pixels. A *pixel* (short for "picture element") is the smallest dot you can draw on the screen.

For example, if `the clickLoc` has a value of `20,35` that means you clicked 20 pixels from the left edge of the card window and 35 pixels from the top of the window. The value of the upper-left corner of the screen is `0,0`.

5. **Make a note of the two numbers in the Message box; you'll need to put them into your animation handler.**

Finding the ending point

You use similar steps to find the ending point for dragging:

1. **Click somewhere below and to the right of the graphic image.**

   Make sure the point that you click is lower and farther right than the image. If necessary, move the Message box out of the way temporarily (drag it by the bar across the top).

2. **Type `the clickLoc` into the Message box.**

   Again, be sure to include `the`.

3. **Press Return.**

4. **Make a note of the new numbers in the Message box. These numbers will also go into your handler.**

5. **Close the Message box.**

Making a button and completing the handler

Next you'll create a button to hold the handler that will cause the animation:

1. **Choose New Button from the Objects menu and name the new button** `Spin`

2. **Type the following script for the button, substituting the numbers you made note of in the previous sections for the ones shown here:**

```
on mouseUp
  choose select tool
  drag from 10,10 to 490,340 with commandKey
  repeat for 16 times
    doMenu "rotate right"
  end repeat
  choose browse tool
end mouseUp
```

   When you type the line beginning with `drag`, be sure to substitute the numbers you got by using `the clickLoc` in the Message box. The first pair of numbers, the starting point for the drag, should follow the word `from`. The second pair of numbers, the ending point, should follow the word `to`. The phrase `with commandKey` has the same effect as dragging with the Selection tool while holding down the Command key. The selection is tightened to the perimeter of the image.

   `DoMenu` lets you choose any command from an available HyperCard menu. As with other names, it's a good idea to put the command name inside quotation marks.

3. **Press Enter.**

4. **Switch to the Browse tool and click the Spin button.**

   The picture turns through four complete rotations; that's because the `repeat` statement specifies 16 repetitions of the 90° Rotate Right command. (If you had not specified a number, the image would keep turning until you pressed ⌘-period.)

A stack for fun

Here's a stack that's easy to build and fun to play with. It randomly generates newspaper headlines from lists of words that you supply.

1. **Create a new stack.**

   Name the stack Headlines or whatever you like.

Next you'll add some fields and buttons to the first (and only) card in the stack.

2. **Create three scrolling fields named Man, Bites, and Dog.**

   Choose any font you want for these fields.

3. **Type some words or phrases into the scrolling fields.**

   In the field named Man, type the names of some friends. In the field named Bites, type some verbs. In the field named Dog, type some nouns. Press Return after each word or phrase to put it on a separate line.

   For now you can just type two or three lines into each field. It will be easy to add more words later. Figure 5-9 suggests some words you can type into these fields. Have fun making up your own.

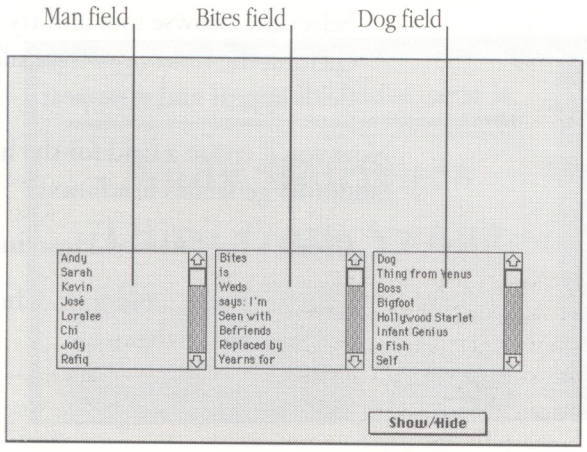


Figure 5-9 Some text for the Man, Bites, and Dog fields

4. **Create a button named Show/Hide that makes the scrolling fields appear and disappear.**

   Write the following script for the button:

```
on mouseUp
  if the visible of card field "Man" is false then
    show card field "Man"
    show card field "Bites"
    show card field "Dog"
  else
    hide card field "Man"
    hide card field "Bites"
    hide card field "Dog"
  end if
end mouseUp
```

   The `visible` property of a field determines whether the field is shown or hidden. When a field is shown, the `visible` property of the field has a value of `true`. When the field is hidden, the `visible` property has a value of `false`.

   This script tests whether the Man field is hidden. If the Man field is hidden, the script shows all three scrolling fields. Otherwise, if the Man field is shown, the script hides all three fields.

   Select the Browse tool and try out the Show/Hide button. By clicking the button, you should be able to make the scrolling fields appear and disappear.

Now you'll create a field for the headline and a button that randomly generates headlines.

5. **Create a field named Headline.**

   Choose a large, bold font. Choose Center so that the headlines are centered in the field.

6. **Create a button named Write Headline.**

   Write the following script for the button:

```
on mouseUp

  put the random of (number of lines in card field "Man") into randomNum
  put line randomNum of card field "Man" into manVar

  put the random of (number of lines in card field "Bites") into randomNum
  put line randomNum of card field "Bites" into bitesVar

  put the random of (number of lines in card field "Dog") into randomNum
  put line randomNum of card field "Dog" into dogVar

  put manVar && bitesVar && dogVar into card field "Headline"

end mouseUp
```

7. **Try out the Write Headline button.**

   Each time you click the button, a different headline appears. If something else happens, check your script and try again.

8. **Paint some graphics on the card to make it look like the front page of a newspaper.**

   Here is one possible design:

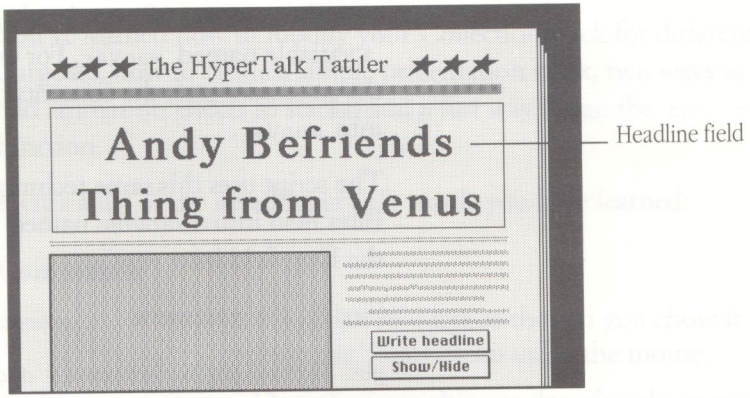


Figure 5-10 Sample graphics for the Man Bites Dog stack

How the Write Headline button's script works

This script uses HyperTalk's `random` function. The syntax of the `random` function is

```
the random of positiveInteger
```

The function returns an integer between 1 and *positiveInteger*. For example, `the random of 6` returns an integer between 1 and 6. Similarly,

```
the random of (number of lines in card field "Man")
```

returns an integer between 1 and the number of lines in the Man field. (The parentheses are included to make the expression easier to read.)

The statement

```
put the random of (number of lines in card field "Man") into randomNum
```

puts a random integer into a variable named `randomNum`. The random integer can be anything from 1 to the number of lines in the Man field.

The statement

```
put line randomNum of card field "Man" into manVar
```

puts the contents of a randomly chosen line in the Man field into a variable named `manVar`. For example, if `randomNum` equals 3, the statement puts the contents of the third line of the Man field into `manVar`.

The script uses this same technique to put a random line from the Bites field into a variable named `bitesVar` and a random line from the Dog field into `dogVar`.

Finally, the statement

```
put manVar && bitesVar && dogVar into card field "Headline"
```

combines the contents of the three variables into a single text string. It puts the three variables into the Headline field with spaces between them.

Where to go from here

Now that you're an experienced scripter, you can go on to other sources to learn more about HyperTalk and more ways of using HyperCard. Many people have written books on HyperCard and scripting that you might find helpful. The *HyperCard Script Language Guide* contains complete descriptions of HyperTalk elements. The HyperTalk Reference stack is also a good resource to consult while you're working.

Look again at the stacks that come with HyperCard, especially Readymade Buttons. See what you can observe about the way their scripts work, and how you might modify some of the scripts to suit your own ways of doing things. Create a stack you can use as a repository for buttons with prewritten handlers and other scripts that you can copy and paste when you want them. Talk to other HyperCard scripters about the stacks they've built and how they've built them.

Most of all, enjoy the creative environment that HyperCard provides. Experiment. Build your own stacks for your own purposes, learning more about HyperTalk as you need to. Your most valuable knowledge of scripting is likely to come from your own experience.

What you've done in this chapter

You've learned how to modify your Collection stack for different purposes, how to create a simple presentation stack, two ways to add animation effects to stacks, and a fun way to use the `random` function.

Here's a list of the new HyperTalk words you have learned:

Commands

`choose`	Chooses a tool just as though you chose it from the Tools menu using the mouse.
`drag`	Does the same thing as dragging the mouse.

`show cards`	Shows cards one after another on the screen. The cards to be shown (all or some number) must be in sequence.
`wait`	Causes HyperCard to pause for a specified length of time, or until some condition becomes true.

Properties

`visible`	A property of buttons and fields that has a value of `true` when the object is shown and `false` when the object is hidden.

Functions

`the clickLoc`	Gives the location where you last clicked on the screen in horizontal and vertical coordinates.
`the random`	Gives a random integer between 1 and a specified number.

Syntax summaries

This section describes the syntax of the commands you learned in this chapter.

Choose

The `choose` command allows you to select a HyperCard tool from within a script.

You can use the `choose` command only when the user level is set to Painting, Authoring, or Scripting. You can set and reset the `userLevel` property inside a handler with the `set` command, if you don't want to change the user level permanently in a stack.

Syntax:

`choose` *`toolName`* `tool`

ToolName is any one of the HyperCard tools from the Tools menu. You must always use `tool` after the name. Here are the HyperTalk names for the tools that you can use:

```
browse      field       reg[ular] poly[gon]
brush       lasso       round rect[angle]
bucket      line        select
button      oval        spray
curve       pencil      text
eraser      rect[angle]
```

The only tool you can't choose from within a script is the Polygon tool.

Example:

```
choose button tool
```

Drag

The `drag` command allows you to manipulate objects and graphics on a card from within a script. It has the same effect as dragging the mouse manually from one point to another.

Syntax:

```
drag from point to point
drag from point to point with key1
drag from point to point with key1, key2
drag from point to point with key1, key2, key3
```

Point consists of the horizontal and vertical coordinates of a point on the screen, separated by commas. You can find the coordinates of a point by placing the pointer there and typing `the mouseLoc` into the Message box.

Key1, *key2*, and *key3* can be `shiftKey`, `optionKey`, or `commandKey`.

Examples:

```
drag from 5,5 to 80,130
drag from 5,5 to 80,130 with commandKey
```

Show cards

The `show cards` command lets you quickly display a number of cards in sequence.

Syntax:

```
show all cards
show positiveInteger cards
```

PositiveInteger is the number of cards you want to show if you don't want to show all of them.

Examples:

```
show all cards
show 5 cards
```

Wait

The `wait` command causes HyperCard to pause for a specified period of time, or until a specified condition is true.

Syntax:

```
wait [for] positiveInteger [ticks]
wait [for] positiveInteger seconds
wait until trueOrFalse
wait while trueOrFalse
```

PositiveInteger specifies how long you want HyperCard to pause. If you want seconds, you must add `second`, `seconds`, or the abbreviation `sec` or `secs`; otherwise, HyperCard uses ticks, which have a value of 1/60 second. No other measurements (such as minutes) can be used.

TrueOrFalse is an expression that evaluates to `true` or `false`. The `wait until` form pauses until *trueOrFalse* is `true`. The `wait while` form pauses until *trueOrFalse* is `false`.

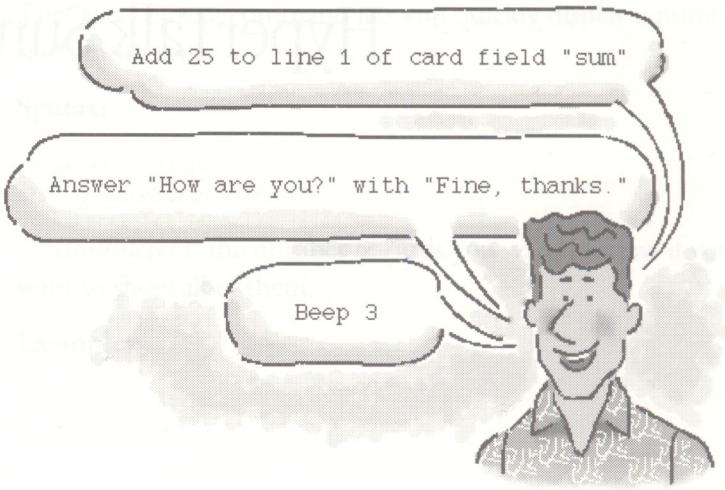
Examples:

```
wait 2 seconds
wait 30 -- waits 30 ticks (or one-half second)
wait until the mouse is down
wait while the mouse is up
```

HyperTalk Summary

This appendix contains

- Syntax statements for all built-in Hypertalk commands, functions, and keywords
- Lists of properties, system messages, and constants
- A table of operators and their order of precedence
- Script editor keyboard shortcuts
- Shortcuts for seeing scripts
- Synonyms and abbreviations
- A summary of the Script menu commands



Syntax statement notation

Syntax statements show the most general form of a command or function, with all elements in the correct order. The syntax statements in this book use the following typographic conventions. Words or phrases in `this kind of type` are HyperTalk language elements that you type exactly as shown. Square 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 command does; in other cases they simply make the command more readable. Words in *italic* are placeholders describing general elements, not specific names; you must replace them in an actual command. For example, *effectName* stands for any of the HyperTalk visual effect names, such as `barn door`, `checkerboard`, or `zoom out`.

It doesn't matter whether you use uppercase or lowercase letters in HyperTalk, but names formed from two words are often shown in small letters with a capital in the middle (`likeThis`) to make them more readable.

Commands

This section lists the syntax of all HyperTalk commands. For more information about HyperTalk commands, see the HyperTalk Reference stack or the *HyperCard Script Language Guide.*

```
add number to [chunk of] container

answer text
answer text with reply
answer text with reply1 or reply2
answer text with reply1 or reply2 or reply3
answer file text [of type fileType]

arrowKey direction

ask text [with defaultText ]
ask password text [with defaultText ]
ask file text [with filename]

beep
beep positiveInteger

choose toolName tool
choose tool positiveInteger
```

❖ *Note: positive integer* must be a number from 1 to 18. ❖

```
click at point
click at point with key1
click at point with key1, key2
click at point with key1, key2, key3

close file fileName

close printing

controlKey positiveInteger
```

❖ *Note: positiveInteger* must be a number from 0 to 255. ❖

```
convert date to formatName [and formatName]
convert [chunk of] container to formatName [and formatName]
```

```
create stack stackName [with background] [in a new window]
create menu menuName

debug checkpoint
debug timeShare on
debug timeShare off

delete chunk of container
delete menu
delete menuItem of menu
delete menuItem from menu

dial positiveInteger
dial positiveInteger with modem
dial positiveInteger with [modem] modemCommands

disable menuItem of menu
disable menu

divide [chunk of] container by number

doMenu itemName, menuName [without dialog]
doMenu itemName [without dialog]

drag from point to point
drag from point to point with key1
drag from point to point with key1, key2
drag from point to point with key1, key2, key3

edit [the] script of object

enable menuItem of menu
enable menu

enterInField

enterKey

export paint to file filename

find text [in field ]
find chars text [in field ]
find word text [in field ]
find whole text [in field ]
find string text [in field ]
```

```
functionKey positiveInteger
```

❖ *Note: positiveInteger* be a number from 1 to 15. ❖

```
get expression
get [the] property of object

go [to] stack
go [to] background [of stack ]
go [to] card [of background ] [of stack ]
go [to] [without dialog] [card of] [background of] stack [in a
  new window]

help

hide button
hide field
hide window
hide card picture
hide background picture
hide picture of card
hide picture of background
hide menuBar
hide titleBar

import paint from file fileName

lock screen

[un]mark [this] card
[un]mark card
[un]mark cards where trueOrFalse
[un]mark all cards
[un]mark cards by finding text [in field]
[un]mark cards by finding chars text [in field]
[un]mark cards by finding word text [in field]
[un]mark cards by finding whole text [in field]
[un]mark cards by finding string text [in field]

multiply [chunk of] container by number

open programName
open fileName with programName

open file fileName
```

```
open printing [with dialog]

open report printing [with dialog]
open report printing with template text

play sound
play sound notes
play sound tempo positiveInteger
play sound tempo positiveInteger notes
play stop

pop card
pop card into [chunk of] container
pop card after [chunk of] container
pop card before [chunk of] container

print card [from point1 to point2]
print card [from point1 to point2]
print positiveInteger cards
print all cards
print marked cards
print field
print fileName with programName
print expression

push card
push card [of stack]
push bacground [of stack]
push stack

put expression
put expression into [chunk of] container
put expression after [chunk of] container
put expression before [chunk of] container
put menuItemList preposition [menuItem of] menu
   [with menuMessages messageNameList]

read from file fileName until character
read from file fileName for positiveInteger

reset paint
reset menuBar
```

```
returnInField

returnKey

save [this] stack as [stack] stackName
save stack stackName as [stack] stackName

select empty
select button
select field

select text of container
select before text of container
select after text of container
select chunk of container
select before chunk of container
select after chunk of container
```

❖ *Note: container* must be a field or the message box. ❖

```
set [the] property to expression
set [the] property of object to expression
set [the] property of window to expression
set [the] property of menuItem of menu to expression
set [the] property of chunk of field to expression
```

❖ *Note: expression* must yield a valid setting for the specified property. ❖

```
show positiveInteger cards
show all cards
show marked cards

show button [at point]
show field [at point]
show window [at point]
show picture of card
show picture of background
show card picture
show background picture
show menuBar
show titleBar

sort [sortDirection] [sortStyle] by expression
```

```
start using stack

stop using stack

subtract number from [chunk of] container

tabkey

type text
type text with key1
type text with key1, key2
type text with key1, key2, key3

unlock screen
unlock screen with effect [speed] [to image]
unlock screen with visual [effect] effect [speed] [to image]

visual [effect] effect [speed] [to image]

wait [for] positiveInteger [ticks]
wait [for] positiveInteger seconds
wait until trueOrFalse
wait while trueOrFalse

write text to file fileName
```

Functions

This section lists the syntax for all of HyperTalk's built-in functions, as well as the value returned by the function.

When using functions in HyperTalk statements you must either use the word `the` before the function name or add parentheses after it (both forms are shown in the list that follows). The parentheses are used to enclose any values on which the function operates. These values are called *parameters*. If the function takes several parameters (for example, the `average` function), you must separate the parameters with commas. For a more complete discussion of functions and parameters, see the HyperTalk Reference stack or the *HyperCard Script Language Guide*.

Syntax of function	Value returned by function
`the abs of` *number* `abs(`*number*`)`	Absolute value of *number*
`annuity(`*rate*`,` *periods*`)`	Current or future value of an annuity
`the atan of` *number* `atan(`*number*`)`	Arc tangent of *number*, expressed in radians
`average(`*numberList*`)`	Average of the numbers in *numberList*
`the charToNum of` *character* `charToNum(`*character*`)`	ASCII value of a character
`the clickChunk` `clickChunk()`	Description of the chunk of text that the user last clicked
`the clickH` `clickH()`	Horizontal coordinate of the point where the user last clicked
`the clickLine` `clickLine()`	Description of the line that the user last clicked
`the clickLoc` `clickLoc()`	Horizontal and vertical coordinates of the point where the user last clicked
`the clickText` `clickText()`	The word (or string of grouped text) that user last clicked
`the clickV` `clickV()`	Vertical coordinate of the point where the user last clicked
`the commandKey` `commandKey()`	Position of the Command key (the ⌘ key): `up` or `down`
`compound(`*rate*`,` *periods*`)`	Present or future value of a compound interest-bearing account

Syntax of function	Value returned by function
`the cos of` *number* `cos(`*number*`)`	Cosine of *number*, expressed in radians

`the date`
`the long date`
`the abbreviated date`
`the short date`
`date()`
	Current date set in the Macintosh

`the diskSpace` `diskSpace()`	Amount of free space on the current disk
`the exp of` *number* `exp(`*number*`)`	Mathematical exponential (the constant *e* raised to the power of *number*)
`the exp1 of` *number* `exp1(`*number*`)`	1 less than mathematical exponential: `exp()-1`
`the exp2 of` *number* `exp2(`*number*`)`	The value of 2 raised to the power of *number*
`the foundChunk` `foundChunk()`	Description of where the text is found in a field
`the foundField` `foundField()`	Which field the found text is in
`the foundLine` `foundLine()`	Which line the found text is in
`the foundText` `foundText()`	Characters found by the `find` command
`the length of` *text* `length(`*text*`)`	Number of characters in a text string

Syntax of function	Value returned by function
`the ln of` *number* `ln(`*number*`)`	Base-*e* (natural) logarithm of *number*
`the ln1 of` *number* `ln1(`*number*`)`	Base-*e* (natural) logarithm of (1 + *number*)
`the log2 of` *number* `log2(`*number*`)`	Base-2 logarithm of *number*
`max(`*numberList*`)`	Highest number in *numberList*
`the menus` `menus()`	A list of the items currently in the menu bar
`min(`*numberList*`)`	Lowest number in *numberList*
`the mouse` `mouse()`	Position of the mouse button: `up` or `down`
`the mouseClick` `mouseClick()`	`True` or `false`, depending on whether the mouse button is clicked
`the mouseH` `mouseH()`	Horizontal position of the pointer on the screen
`the mouseLoc` `mouseLoc()`	Horizontal and vertical coordinates of the pointer
`the mouseV` `mouseV()`	Vertical position of the pointer
`the number of background buttons` `number(background buttons)`	Number of background buttons in the current background
`the number of [card] buttons` `number([card] buttons)`	Number of card buttons on the current card

Syntax of function	Value returned by function
`the number of cards [in` *background*`]` `number(cards [in` *background*`])`	Number of cards in a specified background
`the number of` *chunks* `in` *expression* `number(`*chunks* `in` *expression*`)`	Number of characters, words, items, or lines in a specified expression
`the number of [background] fields` `number([background] fields)`	Number of background fields in the current background
`the number of card fields` `number(card fields)`	Number of card fields on the current card
`the number of marked cards` `number(marked cards)`	Number of marked cards in the current stack
`the number of menuItems of` *menu* `number(menuItems of` *menu*`)`	Number of items in a specified menu
`the number of menus` `number(menus)`	Number of menus in the menu bar
`[the] number of windows` `number(windows)`	Number of all windows in Hypercard
`the numToChar of` *positiveInteger* `numToChar(`*positiveInteger*`)`	Character corresponding to an ASCII value
`offset(`*text1*`,` *text2*`)`	Number of characters between the beginnings of two strings
`the optionKey` `optionKey()`	Position of the Option key: `up` or `down`
`the param of` *positiveInteger* `param(`*positiveInteger*`)`	Value of a parameter in a list
`the paramCount` `paramCount()`	Total number of parameters

Syntax of function	Value returned by function
`the params` `params()`	Entire list of parameters
`the random of` *positiveInteger* `random (`*positiveInteger*`)`	Random integer from 1 to *positiveInteger*
`the result` `result()`	A text string if `find` or `go` is unsuccessful
`the round of` *number* `round(`*number*`)`	Nearest integer to *number* (odd integer plus 0.5 rounds up; even integer plus 0.5 rounds down)
`the screenRect` `screenRect ()`	Dimensions of the screen in pixels
`the seconds` `seconds()`	Number of seconds between midnight January 1, 1904, and the current time in your Macintosh
`the selectedChunk` `selectedChunk()`	Description of the location of the selected text
`the selectedField` `selectedField()`	Which field the selected text is in
`the selectedLine` `selectedLine()`	Which line the selected text is in
`the selectedText` `selectedText()`	Text currently selected
`the shiftKey` `shiftKey()`	Position of the Shift key: `up` or `down`
`the sin of` *number* `sin(`*number*`)`	Sine of *number*, expressed in radians

Syntax of function	Value returned by function
`the sound` `sound()`	Name of the sound resource currently playing (or `done` if none is playing)
`the sqrt of` *number* `sqrt(`*number*`)`	Square root of a number—if *number* is negative gives the result NAN(001) meaning "not a number"
`the tan of` *number* `tan(`*number*`)`	Tangent of *number*, expressed in radians
`the target` `target()`	Description of the original recipient of a message
`the ticks` `ticks()`	Number of ticks (1/60 second) since the Macintosh was last started

`the time`
`the long time`
`the abbreviated time`
`the short time`
`time()`
	Current time set in the Macintosh

`the tool` `tool()`	Name of the currently chosen tool
`the trunc of` *number* `trunc(`*number*`)`	Integer part of *number*
`the value of` *expression* `value(`*expression*`)`	Value of *expression*
`the windows` `windows()`	A list of all windows currently in use, including palettes and script windows.

Keywords

The following list includes HyperTalk keywords and their syntax. Keywords are predefined; you can't redefine them—for instance, you can't use a keyword as the name of a variable.

`Send` is the only keyword that can be used in the Message box.

```
do expression

else

end functionName
end messageName
end if
end repeat

exit functionName
exit messageName
exit repeat
exit to HyperCard

function functionName
function functionName parameterList

global variableList

if trueOrFalse then

next repeat

on messageName
on messageName parameterList

pass functionName
pass messageName

repeat [forever]
repeat [for] positiveInteger [times]
repeat until trueOrFalse
repeat while trueOrFalse
repeat with variable = start to finish
repeat with variable = start down to finish

return expression

send "messageName[parameterList]" [to object]
send "messageName[parameterList]" to HyperCard
```

System messages

HyperCard sends these messages to the objects specified to inform them of system events.

Messages sent to a button

```
deleteButton
mouseDown
mouseEnter
mouseLeave
mouseStillDown
mouseUp
mouseWithin
newButton
```

Messages sent to a field

```
closeField
deleteField
exitField
mouseDown
mouseEnter
mouseLeave
mouseStillDown
mouseUp
mouseWithin
newField
openField
```

Messages sent to the current card

```
closeBackground
closeCard
closeStack
deleteBackground
deleteCard
deleteStack
idle
mouseDown
mouseStillDown
mouseUp
newBackground
newCard
newStack
openBackground
openCard
openStack
quit
resume
resumeStack
startUp
suspend
suspendStack
```

Properties

This section lists the properties of the HyperCard environment and of objects.

Global properties

```
blindTyping
cursor
debugger
dragSpeed
editBkgnd
language
lockMessages
lockRecent
lockScreen
messageWatcher
messageWatching
numberFormat
powerKeys
printMargins
printTextAlign
printTextFont
printTextHeight
printTextSize
PrintTextStyle
scriptEditor
scriptTextFont
scriptTextSize
stacksInUse
suspended
textArrows
userLevel
userModify
variableWatcher
variableWatching
version
```

Button properties

```
autoHilite
hilite
icon
ID
loc[ation]
name
number
rect[angle]
script
sharedHilite
showName
style
textAlign
textFont
textHeight
textSize
textStyle
visible
```

Field properties

```
autoTab
dontSearch
dontWrap
fixedLineHeight
ID
loc[ation]
lockText
name
number
rect[angle]
script
scroll
sharedText
showLines
style
textAlign
textFont
textHeight
textSize
textStyle
visible
wideMargins
```

Card properties

```
cantDelete
cantModify
dontSearch
ID
marked
name
number
rect[angle]
script
showPict
```

Background properties

```
cantDelete
cantModify
dontSearch
ID
name
number
script
showPict
```

Stack properties

```
cantDelete
cantModify
cantPeek
freesize
name
script
size
```

Rectangle properties

```
bottom
bottomRight
height
left
right
top
topLeft
width
```

Painting properties

```
brush
centered
filled
grid
lineSize
multiple
multiSpace
pattern
polySides
textAlign
textFont
textHeight
textSize
textStyle
```

Window properties

```
loc[ation]
rect[angle]
scroll
visible
```

Menu properties

```
checkMark
commandChar
enabled
markChar
menuMessage
name
```

Constants

Constants are named values that never change. You can't use the name of a constant as a variable name.

Constant	Description
`down`	The value of the key functions for the Command, Option, and Shift keys and for the mouse button when pressed
`empty`	A string containing nothing (the *null* string)—same as `""`
`false`	The opposite of `true`
`formFeed`	The form feed character, ASCII 12
`lineFeed`	The line feed character, ASCII 10
`pi`	The value of pi to 20 decimal places
`quote`	The double quotation mark character
`return`	The return character, ASCII 13
`space`	The space character, ASCII 32—same as `" "`
`tab`	The horizontal tab character, ASCII 9
`true`	The opposite of `false`
`up`	The value of the key functions for the Command, Option, and Shift keys and for the mouse button when not currently pressed
`zero..ten`	The numbers 0 through 10

Operator precedence

The table below shows the order of precedence of HyperTalk operators. The order of precedence determines which operation HyperCard performs first when evaluating an expression. Operators are evaluated from left to right, except for exponentiation, which is from right to left. Parentheses force evaluation in a certain order; for example, `2*3+5` yields `11`, but `2*(3+5)` yields `16`.

Order	Operators	Type of operator
1	`( )`	Grouping
2	`-`	Minus sign for numbers
	`not`	Logical negation for Boolean values

	`there is a`
	`there is not a`
	`there is no`
		Tests whether or not an object, window, menu, or menu item exists.

3	`^`	Exponentiation for numbers
4	`* / div mod`	Multiplication and division for numbers
5	`+ -`	Addition and subtraction for numbers
6	`& &&`	Concatenation of text
7	`> < <= >= ≤ ≥`	Comparison for numbers or text

	`is in`
	`contains`
	`is not in`
		Comparison for text

	`is a`	Tests whether or not a value is a certain type, such as a number, integer, point, rectangle, Boolean, or date.
8	`= is is not <> ≠`	Comparison for numbers or text
9	`and`	Logical for Boolean values
10	`or`	Logical for Boolean values

Script editor keyboard shortcuts

The following table lists keyboard combinations used to edit and format scripts.

Key combination	Effect
⌘-A	Select entire script
⌘-C	Copy selection to Clipboard
⌘-D	Set a checkpoint
⌘-F	Find text
⌘-G	Find next occurrence of same text
⌘-H	Find current selection
⌘-P	Print selection or (if no selection) entire script
⌘-R	Replace text
⌘-S	Save script
⌘-T	Replace next occurrence of same text
⌘-V	Paste Clipboard contents at insertion point
⌘-W	Close script
⌘-X	Cut selection to Clipboard
⌘-period	Close script without saving changes
⌘-(hyphen)	Comment current line or selected lines
⌘-(equals)	Uncomment current line or selected lines
Enter	Close script and save changes
Return	Return character—indicates end of HyperTalk statement
Option-Return	Wrap line without return character ("soft" return—symbolized by ¬ in scripts. Don't use a "soft" return inside quotation marks.)
Tab	Format script

Shortcuts for seeing scripts

The following table lists shortcuts for displaying the scripts of HyperCard objects.

Script	Shortcut(s)
Button script	Click button while pressing Option and ⌘ keys
	Double-click button with Button tool while pressing Shift key
Field script	Click field while pressing Option, ⌘, and Shift keys
	Double-click field with Field tool while pressing Shift key
Card script	Press ⌘-Option-C
Background script	Press ⌘-Option-B
Stack script	Press ⌘-Option-S

Synonyms and abbreviations

This table lists synonyms and abbreviations that you can use in scripts.

Term	Synonym or abbreviation
`abbreviated`	`abbr`
	`abbrev`
`background`	`bg`
	`bkgnd`
`backgrounds`	`bgs`
	`bkgnds`
`button`	`btn`
`buttons`	`btns`
`card`	`cd`
`cards`	`cds`
`character`	`char`

Term	Synonym or abbreviation
characters	chars
commandKey	cmdKey
field	fld
fields	flds
gray	grey
location	loc
message box	message msg box msg
middle	mid
picture	pict
polygon	poly
previous	prev
rectangle	rect
regular	reg
round rectangle	round rect
second (*time unit*)	sec secs seconds
spray can	spray
ticks	tick

The Script menu

When the script editor is open, the Script menu appears in the menu bar.

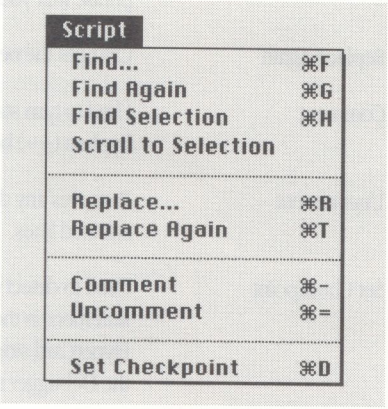


Figure A-1 The Script menu

The Script menu contains commands that are helpful in editing scripts and checking scripts for mistakes. Here's a summary of the Script menu commands:

Command	Description
Find	Searches the script for a word or phrase that you specify.
Find Again	Searches the script for the next occurrence of the text.
Find Selection	Searches the script for the next occurrence of the text that is currently selected.
Scroll to Selection	In long scripts that don't fit in the script editor window, displays the portion of the script where the insertion point is placed or where text is selected.

Command	Description
Replace	Changes a word or phrase in a script to another word or phrase that you specify.
Replace Again	Changes the next occurrence of the text.
Comment	Used to turn statements into comments. Places a double hyphen (--) before the selected text or selected lines.
Uncomment	Removes any double hyphens from the selected text or selected lines.
Set Checkpoint	Used to detect errors in scripts. Places a checkpoint before a statement in the script. When the statement executes, HyperCard stops at that statement and displays the script and the Debugger menu.

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.

ASCII Acronym for *American Standard Code for Information Interchange,* pronounced "ASK-ee." A standard that assigns a unique number to each text character and control character. ASCII code is used for representing text inside a computer and for transmitting information between computers and other devices.

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 layer The layer behind the **card layer,** containing all the elements of the **background.** You see the elements of both layers when you look at a **card,** as if the card layer were a transparent layer in front of the background layer. The **background button** or **background field** created most recently is the topmost object in the background layer (that is, closest within the background layer to the front of the screen). The **background picture** is behind (farther from the front of the screen) the objects in the background layer.

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.**

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.

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 card is a composite of two layers—a foreground layer, called the **card layer,** and a **background layer.** You see the elements of both layers when you look at a card, as if the card layer were a transparent layer in front of the background layer. Each layer can contain its own buttons, fields, and graphics.

card button A button in the **card layer** of a single card. Compare with **background button.**

card field A field in the **card layer** of a specific card; its size, position, text attributes, and contents are limited to the card on which the field is created. Compare with **background field.**

card layer The layer in front of the **background layer.** You see the elements of both layers when you look at a **card,** as if the card layer were a transparent layer in front of the background layer. The **card button** or **card field** created most recently is the topmost object in the card layer (that is, closest within the card layer to the front of the screen). The **card picture** is behind (farther from the front of the screen) the objects in the card layer and in front of all the elements in the background layer.

card picture The graphics in the **card layer** of a single 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.**

command A response to a particular message; a built-in message handler residing in HyperCard. Compare with **function** and **keyword.** See also **external command.**

command-key (⌘-key) equivalent The combination of the ⌘ key and another key on the keyboard that you can press instead of choosing a command from a menu.

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.

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 `""`. HyperCard contains a number of constants, such as `true`, `false`, `up`, `down`, and `pi`. 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.

current (adj.) The card, background, or stack you're looking at now. For example, the current card is the one you see in the active window 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.

empty Used to describe scripts that contain no handlers. Every HyperCard object has a script, even if the script is empty. See also **null.**

expression A HyperTalk description of how to get a **value;** a **source of value** or complex expression built from sources of value and **operators.**

external command (Also known as 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 (Also known as 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.**

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.**

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.** Sometimes a script must supply a function with starting values or **parameters.** Compare with **command** and **keyword.**

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.

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 **object hierarchy.**

Home cards The first five cards in the standard Home stack, designed to hold buttons that take you to stacks, applications, and documents. Choose Home from the Go menu (or press ⌘-H) to get to the card in the standard Home stack that you've seen most recently. You can also type `go home` in the Message box or include it as a statement in a handler.

HyperTalk HyperCard's built-in script language for HyperCard users.

identifier A character string of any length, beginning with an alphabetic character, containing any alphanumeric character and, optionally, the underscore character. Identifiers are used for variable and handler names.

keyword Any one of the 14 words that have a special meaning in HyperTalk statements. Examples of keywords are `end`, `if`, `on`, `repeat`, and `send`.

link A short script, usually in a button but potentially in any HyperCard **object,** that allows you to move immediately to a specific card in a stack, to an application, or to a document. For example, clicking a button that contains a link to your Addresses stack takes you immediately to the first card of that stack.

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). Compare 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** and **object hierarchy.**

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 order The order in which a **message** is passed between objects. For example, a message that goes first to a button, such as `mouseUp`, would go next to the card, then to the background, then to the stack, and finally to HyperCard itself, unless intercepted and acted upon by a **handler.** See also **object hierarchy.**

metasymbol See **syntax.**

null Having no value at all, not even zero. The HyperTalk constant `empty` is defined as a string containing nothing—that is, a null string. A string containing zero would not be empty.

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 descriptor A HyperTalk description that specifies a unique 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.

object hierarchy The hierarchy of objects according to their **message-passing order.** For example, for a message such as `mouseUp`, the button that first receives the message is higher in the object hierarchy than the background, the stack, or HyperCard itself.

object properties The properties that determine how HyperCard objects look and act. For example, the `autohilite` property of a button determines whether or not the button will highlight when clicked.

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.

painting properties The properties that control aspects of HyperCard's painting environment, which is invoked when you choose a Paint tool. For example, the brush property determines the shape of the Brush tool.

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 In printing, the unit of measurement of the height of a text character; one point is about 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. Also, a location on the screen described by two integers, separated by a comma, that represent horizontal and vertical offsets measured in pixels from the upper-left corner of the card window or (in the case of the card window itself) of the screen.

Preferences card The last card in the Home stack, where you can set your **user level** and select or deselect the Blind Typing, Power Keys, and Arrow Keys in Text options.

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. See also **global properties, object properties, painting properties,** and **window properties.**

Recent A special dialog box that holds miniature representations of the last 42 unique cards viewed. Choose Recent from the Go menu to get the dialog box. Also, as in `recent card`, a HyperTalk adjective describing the card you were viewing immediately prior to the current card.

recursion The continuing repetition of an operation or group of operations. Recursion occurs when a handler calls itself.

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.

script editor A large 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 The act of writing **scripts;** writing programs in HyperTalk. Also refers to the **user level** that allows you to look at and change objects' scripts.

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 Paths cards Three cards in the Home stack used to store information about the location of stacks, douments, 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 HyperCard'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 **object hierarchy.**

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`.

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 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 remove from the menu bar by dragging the pointer beyond the menu's edge. HyperCard has two menus that can be torn off: the Tools menu and the Patterns menu. When torn off, these menus are referred to as palettes.

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`.

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 windows such as the Message box and the Tool and Pattern palettes are displayed. For example, the `visible` property of a window determines whether that window is displayed on the screen.

D

E

F

G

H

I, J

K

L

M

T

U

V

W, X, Y, Z

HyperTalk

System messages

HyperCard sends these messages to the objects specified to inform them of system events.

Messages sent to a button

```
deleteButton
mouseDown
mouseEnter
mouseLeave
mouseStillDown
mouseUp
mouseWithin
newButton
```

Messages sent to a field

```
closeField
deleteField
exitField
mouseDown
mouseEnter
mouseLeave
mouseStillDown
mouseUp
mouseWithin
newField
openField
```

Messages sent to the current card

```
closeBackground
closeCard
closeStack
deleteBackground
deleteCard
deleteStack
idle
mouseDown
mouseStillDown
mouseUp
newBackground
newCard
newStack
openBackground
openCard
openStack
quit
resume
resumeStack
startUp
suspend
suspendStack
```

If the Arrow Keys in Text option is turned on (on your Preferences card), hold down the Option key while you press this arrow key.

Keywords

The following list includes HyperTalk keywords and their syntax. Keywords are predefined; you can't redefine them—for instance, you can't use a keyword as the name of a variable.

`Send` is the only keyword that can be used in the Message box.

```
do expression

else

end functionName
end messageName
end if
end repeat

exit functionName
exit messageName
exit repeat
exit to HyperCard

function functionName
function functionName parameterList

global variableList

if trueOrFalse then

next repeat

on messageName
on messageName parameterList

pass functionName
pass messageName

repeat [forever]
repeat [for] positiveInteger [times]
repeat until trueOrFalse
repeat while trueOrFalse
repeat with variable = start to finish
repeat with variable = start down to finish

return expression

send "messageName[parameterList]" [to object]
send "messageName[parameterList]" to HyperCard
```

Commands

This section lists the syntax of all HyperTalk commands. For more information about HyperTalk commands, see the HyperTalk Reference stack or the *HyperCard Script Language Guide.*

```
add number to [chunk of] container

answer text
answer text with reply
answer text with reply1 or reply2
answer text with reply1 or reply2 or reply3
answer file text [of type fileType]

arrowKey direction

ask text [with defaultText ]
ask password text [with defaultText ]
ask file text [with filename]

beep
beep positiveInteger

choose toolName tool
choose tool positiveInteger
```

❖ *Note: positive integer* must must be a number from 1 to 18. ❖

```
click at point
click at point with key1
click at point with key1, key2
click at point with key1, key2, key3

close file fileName

close printing

controlKey positiveInteger
```

❖ *Note: positiveInteger* must be a number from 0 to 255. ❖

```
convert date to formatName [and formatName]
convert [chunk of] container to formatName [and formatName]

create stack stackName [with background] [in a new window]
create menu menuName
```

```
debug checkpoint
debug timeShare on
debug timeShare off

delete chunk of container
delete menu
delete menuItem of menu
delete menuItem from menu

dial positiveInteger
dial positiveInteger with modem
dial positiveInteger with [modem] modemCommands

disable menuItem of menu
disable menu

divide [chunk of] container by number

doMenu itemName, menuName [without dialog]
doMenu itemName [without dialog]

drag from point to point
drag from point to point with key1
drag from point to point with key1, key2
drag from point to point with key1, key2, key3

edit [the] script of object

enable menuItem of menu
enable menu

enterInField

enterKey

export paint to file filename

find text [in field ]
find chars text [in field ]
find word text [in field ]
find whole text [in field ]
find string text [in field ]

functionKey positiveInteger
```

❖ *Note: positiveInteger* be a number from 1 to 15. ❖

```
get expression
get [the] property of object

go [to] stack
go [to] background [of stack ]
go [to] card [of background ] [of stack ]
go [to] [card of] [background of] stack [in a new window]
  [without dialog]

help

hide button
hide field
hide window
hide card picture
hide background picture
hide picture of card
hide picture of background
hide menuBar
hide titleBar

import paint from file fileName

lock screen

[un]mark [this] card
[un]mark card
[un]mark cards where trueOrFalse
[un]mark all cards
[un]mark cards by finding text [in field]
[un]mark cards by finding chars text [in field]
[un]mark cards by finding word text [in field]
[un]mark cards by finding whole text [in field]
[un]mark cards by finding string text [in field]

multiply [chunk of] container by number

open programName
open fileName with programName

open file fileName

open printing [with dialog]

open report printing [with dialog]
open report printing with template text
```

```
play sound
play sound notes
play sound tempo positiveInteger
play sound tempo positiveInteger notes
play stop

pop card
pop card into [chunk of] container
pop card after [chunk of] container
pop card before [chunk of] container

print card [from point1 to point2]
print card [from point1 to point2]
print positiveInteger cards
print all cards
print marked cards
print field
print fileName with programName
print expression

push card
push card [of stack]
push bacground [of stack]
push stack

put expression
put expression into [chunk of] container
put expression after [chunk of] container
put expression before [chunk of] container
put menuItemList preposition [menuItem of] menu
   [with menuMessages messageNameList]

read from file fileName until character
read from file fileName for positiveInteger

reset paint
reset menuBar

returnInField

returnKey
```

```
save [this] stack as [stack] stackName
save stack stackName as [stack] stackName

select empty
select button
select field

select text of container
select before text of container
select after text of container
select chunk of container
select before chunk of container
select after chunk of container
```

❖ *Note:* *container* must be a field or the message box. ❖

```
set [the] property to expression
set [the] property of object to expression
set [the] property of window to expression
set [the] property of menuItem of menu to expression
set [the] property of chunk of field to expression
```

❖ *Note:* *expression* must yield a valid setting for the specified property. ❖

```
show positiveInteger cards
show all cards
show marked cards

show button [at point]
show field [at point]
show window [at point]
show picture of card
show picture of background
show card picture
show background picture
show menuBar
show titleBar

sort [sortDirection] [sortStyle] by expression

start using stack

stop using stack

subtract number from [chunk of] container
```

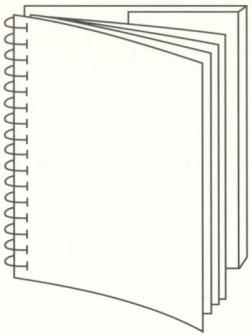
```
tabkey

type text
type text with key1
type text with key1, key2
type text with key1, key2, key3

unlock screen
unlock screen with effect [speed] [to image]
unlock screen with visual [effect] effect [speed] [to image]

visual [effect] effect [speed] [to image]

wait [for] positiveInteger [ticks]
wait [for] positiveInteger seconds
wait until trueOrFalse
wait while trueOrFalse

write text to file fileName
```

THE APPLE PUBLISHING SY

This Apple™ manual was w
and composed on a desktop
system using Apple Macinto
and Microsoft Word. Proof p
created on Apple LaserWrit
Final pages were produced
and output directly to film o
Imagesetters. Line art was c
Studio/8 and Adobe Illustra

Display type is Apple's corp
condensed version of Garar
is Adobe Garamond. Ornan
Zapf Dingbats. Some eleme
program listings, are set in A
fixed-width font.

POSTSCRIPT, the LaserWriter p
language, was developed by
Systems Incorporated.



Tuck end flap inside back cover when using manual.

Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, California 95014
(408) 996-1010
TLX 171-576

Z030-0865-A