



THE MACINTOSH PERFORMANCE LIBRARY

DISK
INCLUDED

COVERS THE NEW
HYPERCARD

COOKING WITH HYPERTALK 2.0

Foreword by Bill Atkinson

DAN WINKLER AND SCOTT KNASTER
Creator of HyperTalk Writer-Engineer, Apple Computer



COOKING WITH
HYPERTALK 2.0
WINKLER
AND
KNASTER

DISK
INCLUDED

0-553-

Cooking with HyperTalk 2.0

Cooking with HyperTalk 2.0

by Dan Winkler and Scott Knaster

Foreword by Bill Atkinson

Illustrations by Laurie Vertelney

Additional stacks by
Lynn Bekkala, David Levine,
Doris Mitsch, Todd Rundgren,
and Robin Shank

Figures drawn by Barbara Knaster



BANTAM BOOKS

NEW YORK • TORONTO • LONDON • SYDNEY • AUCKLAND

COOKING WITH HYPERTALK 2.0

A Bantam Book/July 1990

Bantam Books gratefully acknowledges Apple Computer, Inc. for use of the stack "Valentine from Doris," by Doris Mitsch.

All rights reserved.

Copyright © 1990 by Dan Winkler and Scott Knaster

Composition by Vytron Graphics, Inc., New York City,
under the supervision of Haideh Ettehadieh.

No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without permission in writing from the publisher.

For information address: Bantam Books.

ISBN 0-553-34738-1

Bantam Books are published by Bantam Books, a division of Bantam Doubleday Dell Publishing Group, Inc. Its trademark, consisting of the words "Bantam Books" and the portrayal of a rooster, is Registered in U.S. Patent and Trademark Office and in other countries. Marca Registrada, Bantam Books, 666 Fifth Avenue, New York, New York 10103

PRINTED IN THE UNITED STATES OF AMERICA

0987654321

To my supportive and loving family—Mom, Dad, Anne, and Debbie.

-Dan Winkler

This book is for Barbara.

-Scott Knaster

FOREWORD

by Bill Atkinson

I have fond memories of the year that Dan Winkler spent six days a week, working very long hours over at my house to create the original HyperTalk language. In my mind I picture him hunched over and concentrating on the screen of a prototype Mac II, then known as a Reno, as our cat Sarah sat on his lap and drooled every time he pet her. Sometimes I picture him sitting in the big sofa chair in the corner, playing the guitar while he pondered a tough design problem, and sometimes I picture him discussing the syntax for a new command and how to present it in the simplest and clearest manner. I think of Dan with my daughter Laura and appreciate his warmth and playfulness. It was a fun and exciting time working together with Dan, and a privilege to share his company. I want to acknowledge Dan as a person with great sensitivity, passion, and determination.

I also want to acknowledge Dan for what he has accomplished with his work. Hundreds of thousands of people around the world who never used any other computer language before have found they can create useful scripts with HyperTalk, and customize their computers to do what they want them to. What Dan has created with HyperTalk has liberated programming from the priesthood of nerds and made it accessible to the lay person.

Cooking with HyperTalk 2.0 provides a large library of HyperTalk scripts that are useful anytime you're creating or modifying HyperCard stacks. The library includes professionally written versions of the scripts that people commonly write for themselves to enhance HyperCard.

This book provides building blocks and authoring tools that are useful when you create your own stacks. You can copy many of these scripts and use them as components to achieve sophisticated results quickly. Some of the scripts are authoring tools that help with tasks that commonly arise in the course of building stacks.

This book teaches you good scripting style and advanced scripting techniques that are helpful when you want to improve your own scripting ability. Dan exploits many subtle facets of HyperTalk that have not been widely known until

now. He shows you how to use HyperTalk to the full extent of its design. The scripts are explained in great detail in the book, so you can learn exactly how they do their magic.

Scott's writing is funny. He makes it enjoyable to learn elegant and powerful scripting techniques. Be careful to read this book only in a place where it's all right to laugh out loud.

This book can be used in gradually more advanced ways as your expertise increases. It's valuable to all users from beginner to expert because it provides thorough explanations of advanced techniques. This is a book that you can learn from, build on, and have fun with.

This book comes out of Dan's passion for and commitment to making tools that empower people to accomplish what they're up to, and let them have fun in the process. He has succeeded in a big way by putting a piece of himself into every HyperCard stack and into this book. I hope you enjoy it.

Bill Atkinson
Menlo Park, California
April 28, 1990

ACKNOWLEDGMENTS

We got some help.

Terry Nasta, Managing Editor at Bantam, kept us sane and on the right track throughout the creation of this book. We might have been able to do it without her help, but we wouldn't have wanted to try.

Scot Kamins kept Dan so busy that he didn't have time to make any mistakes.

Chris Crawford helped get us together to write this book in the first place.

Steve Axelrod of the Empire State Building helped us figure out where we were going and what we were doing.

Allen Gibson arranged for a loaner Macintosh SE during a Denver "vacation" so that the book didn't fall behind schedule.

CONTENTS

Foreword	vii
Acknowledgments	ix
Fast Instructions for Fast People	xv
About This Book	xvii
Introduction: It's a Bird, It's a Plane, It's a Floor Wax	xxi

Chapter 1: Getting Ready 1

A Message to You	1
Cooking Up a Command Library	7
Here We Go	14
Summary	15

Chapter 2: Basic Utensils 17

A Custom Scripting World	17
General Utilities	18
Miscellaneous Fun Gimmicks	31
Typing in Shorthand	45
Working with Handlers in Scripts	58
Summary	67

Chapter 3: Working with Groups 69

Group Selection	69
Group Action	87
Summary	116

Chapter 4: Broadcasters 119

Do Something to Each Card in a Stack	119
Do Something to Each Background in a Stack	121

Do Something to Each Field in a Card or Background	122
Do Something to Each Button in a Card or Background	124
Do Something to Each Card in a Background	127
Summary	129

Chapter 5: Getting Around 131

Card Tricks	131
Script Notes	138
Summary	166

Chapter 6: Checking the Ingredients 169

Show All Buttons	169
Show All Fields	171
Show All Hidden Things	173
Create Statements to Show Hidden Buttons	174
Create Statements to Show Hidden Fields	177
Create Statements to Show Hidden Fields and Buttons	178
Show Each Button with Its Number	179
Display Mouse Coordinates	182
Quick Information about Current Stack and Card	185
Summary	186
Lunch Break!	187

Chapter 7: Processing Text 189

Indent Selection As if It Were a Script	189
Put Numbers at Beginning of Lines	190
Make Arrows Move Insertion Point if There's a Selection	195
Recover Text from Any File	196
Shift Selection Left	199
Shift Selection Right	200
Take Returns Out of Selection	202
Put Returns into Selection	203
Compare the Text in Two Fields	205
Summary	207

Chapter 8: Getting Organized 209

- Post-process a Find Message 209
- Search for Names and Nicknames 213
 - Replacers 215
- Find the Selection 223
- Summary 224

Chapter 9: Ready-Made Solutions 227

- Converting Between Coordinate Systems 227
- Drawing Charts and Graphs 248
- Miscellaneous Solutions 263
 - Summary 265
- Dinner Break! 266

About the Stacks on the Disk 267

- The Command Library 267
 - Robin's Paint Box 269
 - AutoPainter 269
 - Black Dog 269
 - HyperGlyphics 269
 - HyperParrot 270
- Valentine from Doris 270
- Flowfazer™ Demo 270
- Disk-claimer 270

Appendix A: Command Library Notes 271

- Command Library Architecture 271
 - Automatic Features 272
 - Different Strokes 273
 - The Trigger Script 274
- Miscellaneous Command Library System Notes 275

Appendix B: Source Code for Externals 277

Source Code for printPS XCMD	277
Source Code for wordWrap XFCN	283
Source Code for Encrypt XFCN	287
Source Code for Decrypt XFCN	289

Index 293

FAST INSTRUCTIONS FOR FAST PEOPLE

If you want to start using the disk in a big hurry, just read this page:

- Insert the disk and copy the stacks to your hard disk. If you don't have a hard disk, you can use the stacks directly from the *Cooking with HyperTalk 2.0* disk. Also, save up some money and buy a hard disk.
- Open the Cookbook Reference stack in the Command Library folder, click the sandwich, then click the Install Scripts button.
- Click the To the Scripts button. You can browse the commands and see which ones you want to try.
- If you're puzzled and lost at this point, read Chapter 1.
- If the command library doesn't work, read Chapter 1.
- You'll find complete installation instructions in the section called "About the Stacks on the Disk" near the end of the book.
- The other stacks don't need any complex instructions. Just play with them.
- Have fun, and don't forget to read some of the rest of the book!

FAST INSTRUCTIONS FOR FAST PEOPLE

If you want to start using FAST as a big bang, just read this page.

• Insert the disk and copy the stacks to your hard disk. If you don't have a hard disk, you can use the stacks directly from the Compact disk (Type 100 & 101). Also, save up some money and buy a hard disk.

• To use the Compact Disk, start in the Command Library folder click the stack icon. To click the (small) stack icon.

• Click the "E" the Script button. You can browse the commands and see which ones you want to use.

• If you're puzzled and lost in the point, read Chapter 1.

• If the command library doesn't work, read Chapter 1.

• You'll find complete installation instructions in the section called "About the Stack on the Disk" near the end of the book.

• The other stacks don't need any complex instructions. Just play with them.

• Have fun and don't forget to read some of the rest of the book!

ABOUT THIS BOOK

This book is an exploration of all you can accomplish with HyperTalk in HyperCard 2.0, and a narrative of just exactly how to go about doing it. Why did we call it a cookbook? Like a cookbook, it has recipes (or in this case, scripts) for making things. It discusses when you might want to use the scripts, how they were created, and how you might want to change them to make something slightly different. And, perhaps most important, it allowed us to have a really good time coming up with funny ideas for the graphic design of the cover and the rest of the book.

Who can use this book? Anyone who's interested in getting a lot done with HyperCard and in learning how it works along the way. You might be looking for ideas for stacks to help you organize your calendar or keep track of phone numbers. Or you might want to become proficient at programming in HyperTalk. This is a big book of scripts, and they fall into these very general categories:

- Command scripts and utility scripts to help you work faster and make better stacks. These scripts are completely explained so that you can learn from and customize them.
- Complete stacks that are fun, useful, or both.

All of the scripts listed in the book are on the disk. This means that you don't have to worry about typing in any of the scripts that you'll read about, because they're already on the disk.

Not the Same Old

There are already 12,623,442 different books on HyperCard. We didn't want to do yet another one unless we could make it provide something really, really different. Here's what's special about ours:

- Dan is the creator of HyperTalk, and he knows HyperTalk the way most of us know how to breathe. His insights and knowledge about the language are present in every script, and we tried to bring out as much of his expertise as possible.
- We believe that the ideas in this book represent a whole new way of thinking in the HyperCard world. We hope these ideas will be widely used in the future.
- The scripts were created with HyperCard 2.0 in mind. For example, our command library system takes advantage of HyperTalk's "start using" command, which was added in HyperCard 2.0. You'll also find the debugging features in HyperCard 2.0 very handy in learning about our scripts.
- As we present the scripts, we'll also throw in other useful and fun information, including notes about particularly interesting and obscure techniques and features in HyperCard.

In this book, all the HyperTalk scripts have numbered lines. Of course, they don't look that way in the real world. We've done it here so that we can easily refer to any line. Sometimes, a single script line doesn't quite fit and wraps back around to the left margin on a second line. You can tell when this happens, because the wrapped part of the line won't have a new line number. Since all the scripts are on disk anyway, you don't have to worry about how to type them in.

As with most software products, the printed stuff has to be finished before the disk stuff. For the last word on features, changes, and other information, see the *Cooking with HyperTalk 2.0* disk that comes with this book. In particular, you'll find handlers in the command library marked "bonus." These are fresh handlers that are described only on the disk, due to late tinkering to make them as great as possible.

Notes

Paragraphs that look like this are used to mark information that's not necessarily in the mainstream of the current discussion, but might be interesting to you. Some folks call these things sidebars.

Script Tips

You'll also find some Script Tips, which are marked like this. These are general ideas and useful techniques that you can use when you're writing your own scripts.

Who's Who

When a book is written by more than one person, there's confusion and suspicion about who to blame for which parts. To spare you this uncomfortable feeling, we'd like to explain how this book was put together.

To avoid any doubt about who is responsible for something in the book, remember this simple rule: Dan did almost all the scripting, and Scott did almost all the writing. So, when you're reading the prose, like right now, it's usually Scott talking to you (hi there). When you're studying the scripts, the design of the stacks, and the concepts in them, you're usually looking at Dan's handiwork.

We also had some friends who did various other things. Robin Shank helped design the command library, and contributed the beautiful and amazing Robin's Paint Box and AutoPainter stacks, as well as a couple of good recipes at the end of Chapters 6 and 9.

Laurie Vertelney rendered the wonderful pictures that start each chapter.

Barbara Knaster (no nepotism here) MacDrew the figures, as always.

Lynn Bekkala created The Black Dog (you'll just have to see him).

Doris Mitsch did the Valentine, a different animated HyperDoggie.

Marge Boots illustrated the HyperGlyphics stack.

Todd Rundgren and David Levine supplied Flowfazer™ Demo.

How You Can Use This Book

Like HyperCard itself, you can use this book and its disk at several different levels, depending on your interest. Here are some suggestions:

- Install and use the stacks, and browse the book to learn a little about them.
- Install and use the stacks, learn about them, then customize them to your own liking.

- Install and use the stacks, and learn how they work by reading the book. Use the scripting techniques for your own scripts.
- Use the complete stacks, and use the book as a booster seat for your 3-year old.
- Read the book for fun and information, and use the disk as a cat toy.

You'll find that the scripts themselves are lightly commented. That's because they're extensively described in the text of the book.

Using Scripts from This Book in Your Stacks

If you want to use any of the scripts in this book or on the accompanying disk in stacks that you make, you can do so, provided you include this sentence in your stacks:

Portions of this stack are used with permission from the book *Cooking with HyperTalk 2.0*, copyright 1990 by Dan Winkler and Scott Knaster.

You don't have to send us any money or gifts or anything.

INTRODUCTION:

It's a Bird, It's a Plane, It's a Floor Wax

When historians sit down to write about the important questions of the late twentieth century (it's important to sit down when you write, or you'll hurt yourself), there are several sure to be included: Was negative campaigning a factor in the presidential elections of the century? How were the serious problems of the major cities solved? Was "The Adventures of Buckaroo Banzai Across the Eighth Dimension" the best movie ever made? And what was HyperCard anyway?

We'd like to discuss the last one, the question of defining HyperCard. Ever since Apple introduced HyperCard in August of 1987, Macintosh users everywhere (especially in Berkeley, California) have been trying to figure out what it is. Initially, a lot of folks thought it was a database program, since it seemed to be especially well-suited to keeping and finding lots of information.

There were, however, some problems in trying to stick HyperCard with that label. Although it did seem to be organized like a database, with information kept in fields and cards that acted something like records, there was a lot more to it. HyperCard also let you cook up your very own buttons, which did what you told them to do and it came with a whizzy painting program built right in. The world of HyperCard was very busy, with lots of communication going on; fields would actually let you know when they'd been changed, buttons would inform you when the mouse pointer touched them, newly created objects would say hello when they appeared. And there was a rather interesting programming language called HyperTalk that held it all together.

For the first few months of its existence, people seemed obsessed with trying to categorize HyperCard (this was probably just human nature). In their attempt to tell the world what was going on, Apple and the HyperCard folks were saying two important things: HyperCard is a software erector set, and it's system software. Let's take a look at both of those descriptions.

An erector set is a box full of parts that you put together to make something. HyperCard certainly fits that description. The parts are the things that make up stacks: buttons, fields, cards, paint, backgrounds, scripts, sounds. Some of them

are already built for you, like the ones in the sample stacks that come with HyperCard, and sometimes you have to build them yourself. Often, you can take something that's already been built and customize it for your own use.

System software is software that doesn't do anything useful by itself, but instead provides a foundation for other things that use it—like a car uses an engine. This description fits when you think of stacks as applications, created by users, to solve some problem or perform some task. Although Apple took a lot of heat in referring to HyperCard as system software, it seems to be a legitimate claim.

There are a lot of fun, neat things in HyperCard, but probably the most fascinating, especially to more technical folks, is HyperTalk, the language of HyperCard scripts. At first glance, HyperTalk looks a lot like English. After more study, you'll find elements of Smalltalk, and even a little bit of Pascal. Sometimes, finished HyperTalk scripts look like a kind of digital poetry:

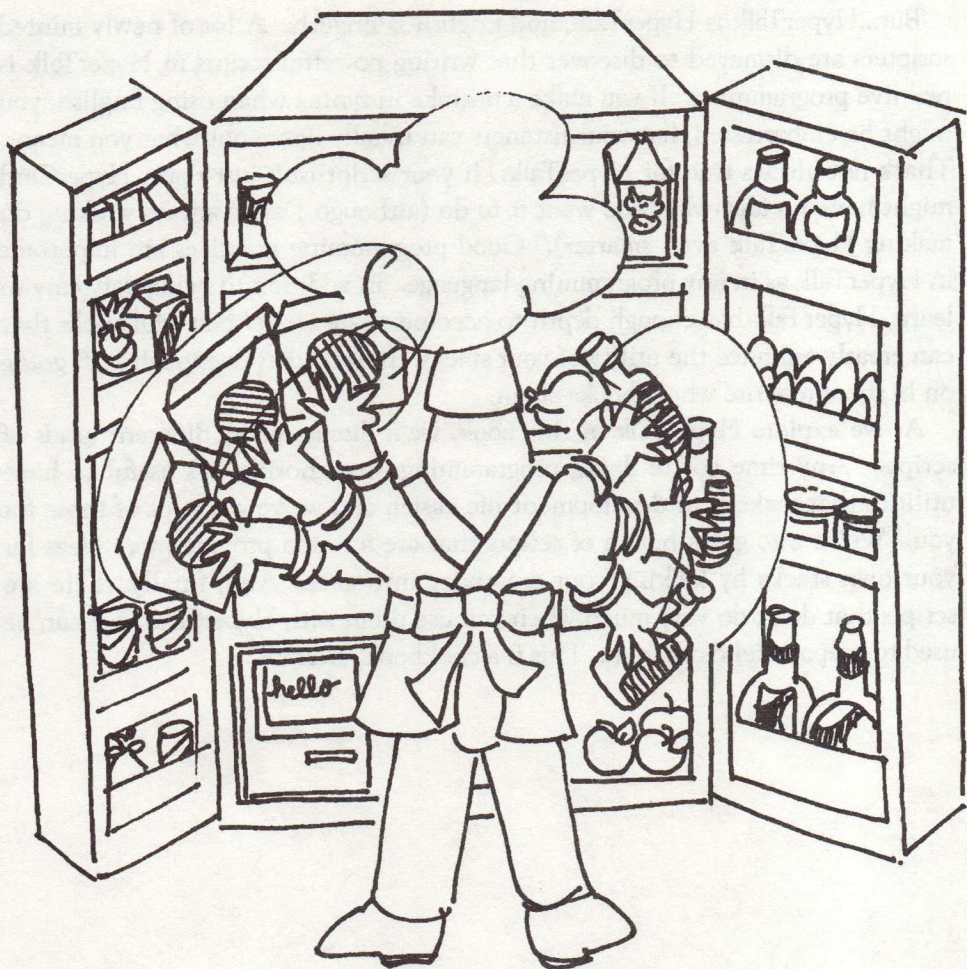
```
on mouseUp
  answer "Do you want to continue?" with "Yes" or "No"
  if it is "No" then exit mouseUp
  repeat 3 times
    play "Happy sound"
  end repeat
  go to next card
end mouseUp
```

HyperTalk is seductive. It looks so much like English that it sucks in new users who say, "I can do that!" And they're right. An interesting truth about HyperTalk is that it actually lives up to its premise of allowing a whole new set of people to take control of their computers (maybe you're one of them). There are lots of things that are incredibly easy to do with HyperTalk, such as creating buttons that link cards together or displaying information in the message box.

Like conversational English, HyperTalk is tolerant. You can say "go to next card," "go next card," or "go next," and they'll all work the same; as of HyperCard 2.0, you can even say "go to *the* next card," which you couldn't get away with in earlier versions (thanks, Dan). You can use "word 3 of name" or "third word of name" and you'll get identical results. Using "wait 5 seconds" is the same as "wait for five seconds". Many of HyperTalk's features give you multiple options to suit your own style.

But...HyperTalk is HyperTalk, and English is English. A lot of newly minted scripters are dismayed to discover that writing powerful scripts in HyperTalk is real live programming. If you make a mistake in syntax when using English, you might be embarrassed, but your listeners can usually figure out what you meant. That's not always true for HyperTalk. If your script isn't just right, HyperCard might have no idea what you want it to do (although Dan is always working on making HyperTalk even smarter). Good programming practices are important in HyperTalk as in any programming language. In addition to being very easy to learn, HyperTalk has enough depth to accommodate lots of powerful tricks that can greatly enhance the utility of your stacks. In fact, there's enough stuff going on in there to write whole books about.

As we explore HyperTalk in this book, we'll discuss many different kinds of scripts. Any time you're doing programming or authoring, it's useful to have utilities that make your development life easier, and we've got a lot of those for you. We've also got a bunch of scripts that are fun and provide more ideas for your own stacks by kicking your creativity into gear. And, finally, there are scripts that don't do very much when you use them with HyperCard, but can be used to prepare delicious food. This is a cookbook, after all.





Chapter 1

GETTING READY

HyperCard objects communicate by sending messages, and HyperTalk lets you invent your own messages by writing handlers that tell what should happen when those messages are sent. In this chapter, we begin with a quick and friendly look at messages in HyperTalk, and quickly escalate into a demonstration of how you can use this capability to put together your own powerful bag of HyperCard tricks.

A Message to You

When HyperCard is running, it's like a busy little city filled with objects that talk to each other a lot. Whether you're zooming through stacks, designing beautiful cards, making new fields and buttons, or just vegging out in front of your computer awhile, the messages between objects are a-flyin' in HyperCard.

You probably know about some of the system messages that HyperCard sends to its objects. For example, when you create a new card, there's a message called `newCard` that's sent to it as a sort of greeting; when you go to a card, an `openCard` message is sent to the card to let it know you're there. When you click in an unlocked field to type, the field gets an `openField` message.

If you've used HyperTalk very much, you probably know you're not limited just to the messages that HyperCard already knows how to send. One of the most powerful features of HyperCard is that it allows you to define and send your own messages. Since messages are caught by handlers that you write, you can also invent your own commands and use them in your scripts.

Terminology corner

Remember that a *handler* is a set of HyperTalk lines that handles a single message. It starts with "on" or "function" and it ends with "end." A *script* belongs to an object and consists of some text which may contain one or more handlers.

Here's an example. Let's say you've recorded a little song called "Sur la Mer," and you've used a digitizer to get the recording into your Macintosh. Now you want to create a stack that will use your little ditty to make it more interesting. You want to have buttons on various cards in your stack that produce these effects:

1. Hide the card window.
2. Play the recorded sound.
3. Show the card window (while the sound is playing).

Since HyperCard keeps working even while sounds are playing, the effect will be real nifty. Your first thought might be to make a bunch of buttons, each with this script, and stick them where they're needed:

```
on mouseUp
  hide card window
  play "Sur la Mer"
  show card window
end mouseUp
```

Of course, the message we want to handle is "mouseUp," just as in virtually every other button script ever invented. Okay, how many of you out there thought this was a good answer? Go ahead and raise your hands. Aha! That was before you started *Cooking with HyperTalk 2.0*! A better solution would be to invent a brand new handler to do the hide/play/show thing, then call that handler from all our buttons. Since we're creating the handler, we can give it any new name we want. Let's call it "trick":

```
on trick
  hide card window
  play "Sur la Mer"
  show card window
end trick
```

This time, we've created a new handler named "trick." When it's called, it'll do the things we wanted. Now, how do we get our buttons to call this handler? Well, just like any other handler, it responds to messages. If we want a button to call this handler, we just use its name like a command in a script:


```
on mouseUp
    trick
end mouseUp
```

Great! We claimed this was a better solution than putting the whole script in each button. Why? For two reasons. First, by putting the guts of the script into a separate handler, then just calling that handler from our buttons, we avoid having to duplicate the hide/play/show commands in every single button. This saves space in the stack by cutting down on the size of each button script. The second good reason is that doing it this way puts the real work in one place, rather than scattering it around. If we decide to change the “trick” script (and we will), only one script has to be changed, rather than changing lots and lots of buttons.

Does this look familiar to you?

If you’ve programmed in traditional languages, you might think of this handler as a subroutine. If you’ve been spared the awe and mystery of programming in other languages, a subroutine is a unit of a program that performs some specific function and can be used repeatedly by other parts of the program. In HyperTalk, all handlers start with “on something” or “function something.” The “something” is a message name, and that message might be one that HyperCard itself usually sends (like closeCard, startup, or mouseDown), or it might be one you’ve invented yourself (like “trick”). There’s no distinction when you write the handler.

There’s another important issue to think about. This script is kind of like that 50-inch projection TV you want to get: it’s neat, but where do you put it? A good way to figure out where to put it is to consider which other handlers will call it. In this case, our “trick” will be called by buttons all around the stack, so the best place to put it is in the stack script. Why? Most HyperTalk messages are sent to the current card, then passed on down to the current background, current stack, home stack, and HyperCard itself if nothing else grabs them along the way. If we put “trick” in a card script, then any button that’s on that card can call “trick” and it will work great, as shown in Figure 1-1.

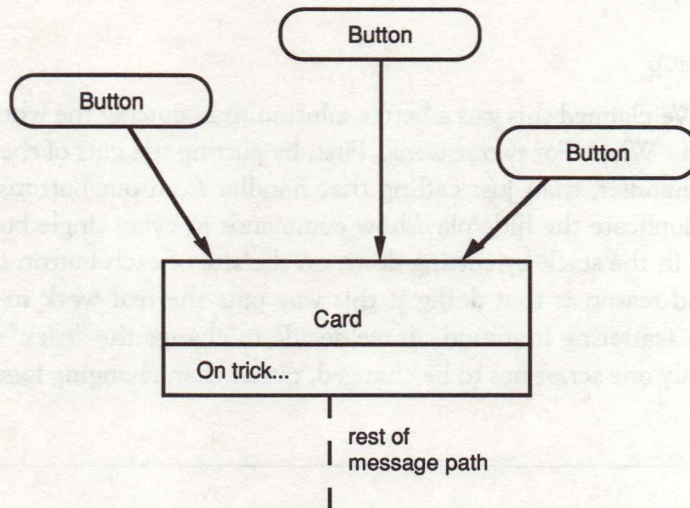


Figure 1-1 Card script catches message

With “trick” in a card script, what happens when a button on another card tries to call it? Whoops, incomplete pass, as you can see in Figure 1-2. The message sent by the button won’t be caught, because the card with the handler is no longer part of HyperCard’s message path.

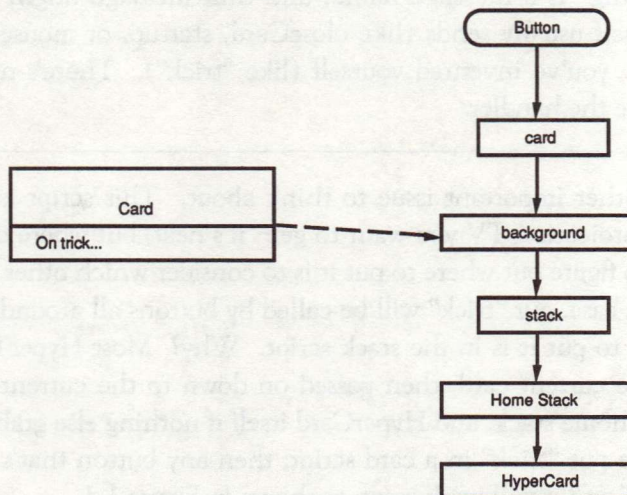


Figure 1-2 Message is not caught

So, if we want “trick” to be usable by buttons on lots of different cards, we have to put it at a lower level. We’ve discovered that putting a handler in a card’s script makes it usable by buttons on that card. If we put “trick” in the background script, any button on any card in that background could call it. If we put it in the stack script, any button on any card in that stack could call it. You get the idea.

If we want “trick” to be available to any button anywhere in the stack, and the stack has only one background, the right place for it is the background script. If the stack has multiple backgrounds, and the handler is needed in more than one background, we put it in the stack script. That’s because the current stack’s script is always part of the inheritance chain. So, our stack and its scripts now look like the arrangement in Figure 1-3. Many different cards have buttons that call “trick,” which is stuffed into the stack script.

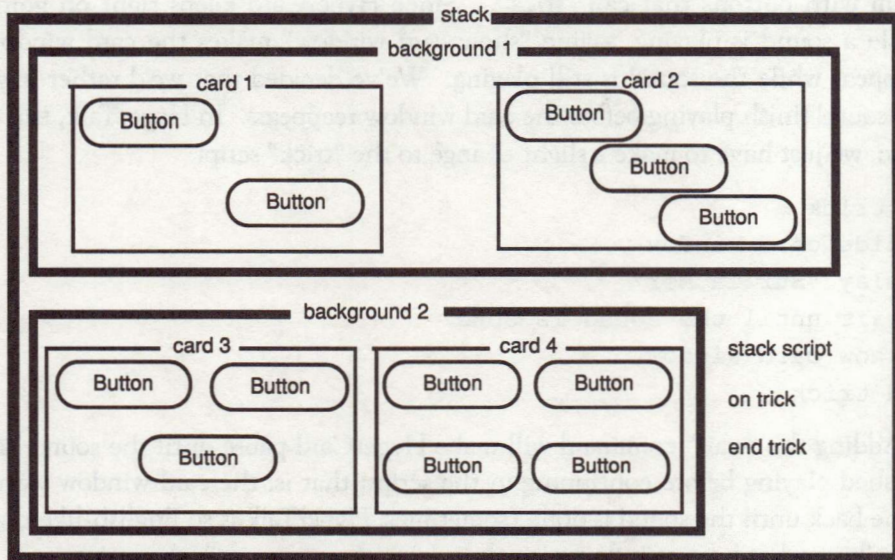


Figure 1-3 Handler in stack script

Dirty tricks

Putting the handler for “trick” in the stack script reveals another handy little feature of HyperCard’s inheritance structure. If we need to write a custom version of “trick” that only affects a particular card, we can do so by putting the handler into the card’s script. Then it will be found first in the hierarchy, and be used instead of the standard one farther down the line. This concept of putting in handlers with the same name at different levels is perfectly legitimate and very powerful, if you know what you’re up to. It can also be the source of unexpected hassles if you do it inadvertently.

We’ll play one more trick with “trick” before moving on. Let’s say we’ve been working on the stack for some time, and we’ve got a few hundred cards, many of them with buttons that call “trick”. Since HyperCard keeps right on going while a sound is playing, saying “show card window” makes the card window reappear while the sound is still playing. We’ve decided that we’d rather have the sound finish playing before the card window reappears. In HyperTalk, this is easy: we just have to make a slight change to the “trick” script:

```
on trick
  hide card window
  play "Sur la Mer"
  wait until the sound is done
  show card window
end trick
```

Adding the “wait” command will make HyperCard pause until the sound has finished playing before continuing in the script; that is, the card window won’t come back until the sound is done (sometimes, HyperTalk is so English-like that describing what a script is doing involves using the same words the script uses).

The interesting thing about putting the “trick” handler into the stack script is that all we have to do when making this change is just add that line. All the buttons call “trick,” which means that they’ll automatically get the new feature. If we’d duplicated the “trick” script in every button, which was our first, faulty idea, we’d have had to change the handler everywhere, in every button. If you

imagine handlers that are even more complex than “trick,” you can really see the advantages to this technique.

The idea of inventing a handler to do something new and useful is a powerful notion, and we’ll explore it in depth in the rest of this chapter.

Polyglot

When you write scripts in HyperTalk, you’re using a language based on English. There are billions of folks who are more comfortable with other languages, though, so HyperTalk can be translated, or *localized* into different languages. This translation is accomplished with a magical (and fairly obscure) HyperCard feature called a translator. It involves creating a special resource that handles the translation for you. So far, this feature of HyperTalk hasn’t been documented, but hang in there.

Cooking Up a Command Library

“Trick” is an example of a handler that’s shared by many buttons in a stack. If we take the idea one step beyond, we can imagine scripts that might be useful to more than one stack—lots of different stacks, in fact. “Trick,” however, is probably too specific an idea to be useful in other stacks. After all, how often will you need to hide the card window, play “Sur la Mer,” and show the card window again? Probably not that often. With just the smallest bit of imagination, though, we can probably think up some handlers that would be useful in many different stacks. Let’s all put on our imagination caps, now....

How about a script that takes the lines in a field and puts them in alphabetical order? Or one that helps us create a calendar? In addition to scripts useful as subroutines for other stacks, we could also collect handlers that are useful when scripting. You might want a nice system for easily aligning a bunch of fields you’ve created, or a way to quickly copy a card from one stack to another, or to automatically change the font for every button on a card, or a fast way to make lots of new friends and buy a bigger house. Almost all of these things are possible in HyperTalk.

Once you’ve written all these scripts, you’re faced with an old question—where to put them? If you put them in scripts of specific stacks, only

those stacks will be able to use them. If you put them in the home stack's script, they'll be accessible to all stacks, but the home stack tends to be a crowded place already. What's the best thing to do?

We said we'd have some scripts useful when called by other scripts, such as a calendar maker, and we call these *subroutine scripts*. Other scripts will most often be called by us directly when we're creating stacks, like the field-liner-upper, and we refer to these as *command scripts*. The easiest way to remember the difference is that we call command scripts when we're making stacks; subroutine scripts are called by our scripts when they're running.

It might be interesting to have a special command library stack somewhere that will hold all our tools in a single script. The only trick would be getting HyperCard to look in that stack for the tools' handlers. You can see this problem in Figure 1-4.

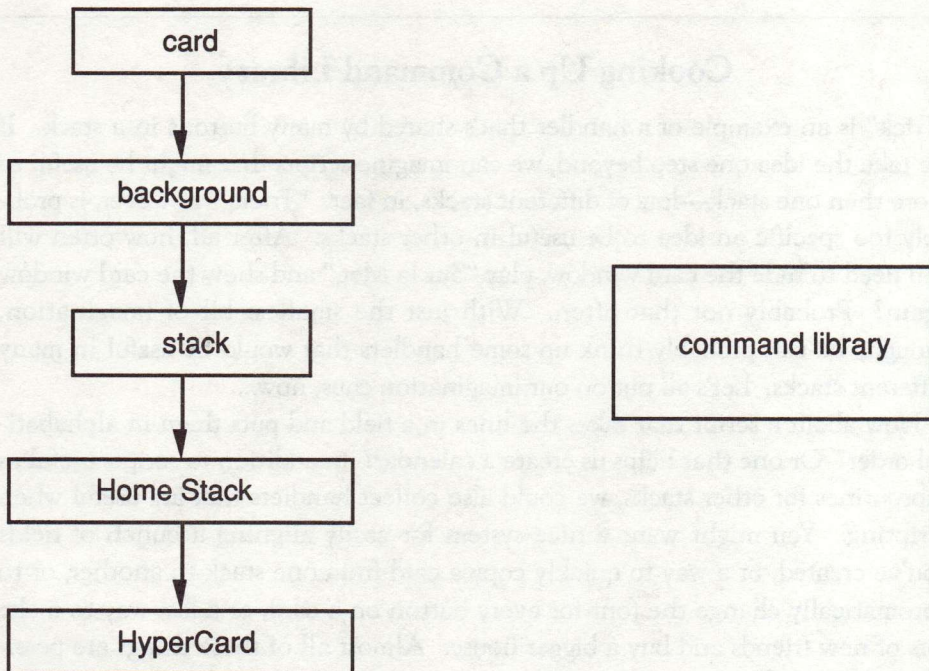


Figure 1-4 Lonely command library stack

HyperCard 2.0 provides a solution to this problem. There's a magical command called "start using stack" that allows you to mess around with the inheritance chain. The "start using stack" command adds other stacks to the inheritance chain and allows them to handle messages. By using a command that looks something like this

```
start using stack "Cookbook Scripts"
```

we can tell HyperCard to give the "Cookbook Scripts" stack a shot at handling any messages that come by. This command puts "Cookbook Scripts" into the inheritance chain just before the Home stack. Now, any messages that aren't handled by the current stack (or any object that's higher) will be sent to "Cookbook Scripts," where our command library can have a chance.

The easiest way to enter a command that will be handled by our script library is simply to type the command into the message box. We still have a problem to solve, though. We'd like to include some commands that operate on a text selection, such as shifting a block of text to the left or right. However, as soon as we start typing in the message box, we lose any selection of text that we may have had (try this yourself in HyperCard if you like – if you select text in a field, then click in the message box, you'll see that the field is no longer selected).

Is there a way to type a command and not lose the selection? Sure, there is! If we can get HyperCard to put up an "ask" dialog box, then type a command into that dialog. The text selection won't be disturbed, and the command can operate on it. The script would look something like this:

1. on command
2. ask "What can I do for you?"
3. do it
4. end command

Line 2 will put up a dialog box and let the user type something in. Whatever is typed will be put into the ever-popular variable named "it." Line 3, which says "do it", is simply HyperTalk's way of saying to execute the command that the user typed into the dialog.

We created this script so that we could type a command without losing a text selection, but if we type "command" (or anything else) into the message box, we'll still lose the selection. We need a way to bring up the command dialog

without typing into the message box. We can do that by using one of the special keys that HyperCard likes.

What goes up?

An interesting question comes up when you're considering HyperCard's inheritance hierarchy. Some users think about messages traveling *up* the hierarchy; that is, they think of buttons and fields as the lowest levels of the chain, followed by cards, backgrounds, and so on, with the big boss, HyperCard itself, at the top of the chain. However, other folks, including the designers of HyperCard, visualize it the other way: buttons and fields are at the top, then you go down to reach cards, backgrounds, and so on. In this view, messages fall *down* through the hierarchy. This way of looking at things has the advantage of matching the visual structure of HyperCard. When you look at a card, you see the buttons and fields frontmost, layered on top of the card, which is on top of the background. The stack and the rest of the hierarchy lurk farther behind, with HyperCard as the ultimate backstop. In this book, we've chosen the second view—we'll talk about the messages traveling *down* the hierarchy.

About those line numbers

No, we're not using a new version of HyperTalk requiring scripts to have line numbers. The numbers that you see at the start of each line of this script are just for our convenience in talking about the script. In the event of an actual script, of course, don't put in the line numbers. By the way, the numbers were put there by a handler that you can read about in Chapter 7.

Isn't that special?

What choices do we have for special keystrokes? HyperCard doesn't provide a way to take control when most keys are pressed, such as the alphanumeric and punctuation keys, but there are a few keystrokes that HyperCard will let you know about: the enter key, the tab key, the return key, the arrow keys, the con-

trol key (when it's pressed with another key) and the function keys that are on the optional Apple Extended Keyboard. Also, you can ask HyperCard if certain keys are down; these are the option, shift, and command keys.

So, we need to choose a keystroke that will bring up the command dialog. A good choice is something we don't have to type during normal use of HyperCard. It's probably not a good idea to use return, enter, or tab, since those keys are frequently used. (Of course, if you don't often use all of these keys, you can make one of them trigger your command dialog.) For now, we're going to choose an unlikely combination that's easy to type, easy to remember, and can be typed on all Macintosh keyboards—shift-tab. With this change, the command-sender script now looks like this:

```
1.  -- script: Command library access
2.
3.  on tabKey
4.      if the shiftKey is not down then pass tabKey
5.      ask "What can I do for you?"
6.      if it is not empty
7.          then do it
8.  end tabKey
```

There are three changes here. First, the message that the handler catches is now `tabKey`, a system message that's sent whenever the user presses the tab key (which is where the clever name "`tabKey`" came from). The second change is line 4...It calls the HyperTalk function `shiftKey` to see if the shift key was also pressed; if not, the handler passes the message on and doesn't do its thing. The final change is the addition of comments in lines 1 and 2. We'll use comments like these whenever we show a script that's actually used on the disk that comes with this book.

This looks like a pretty good system. Usually, we can use a command script just by typing its name into the message box. If we need to operate on a text selection, though, we can just type shift-tab. This will bring up the dialog box asking for a command, which we'll call the *command box*. We'll type the command's name, press return (or press enter or click the OK button, whatever turns you on) to confirm the dialog, and the command will be passed to the command library!

What happens if you depend on shift-tab for some other important function in your stacks, like automatically showing you a list of your next five dental hygiene appointments or something? That's easily fixed. You can change the "command trigger" from shift-tab to just about anything else you want.

For example, if you want to, you can call on the command library whenever you press the tab key, even without holding shift down. All you have to do to make this work is take out line 4 from the script above. Of course, if you do this, you won't be able to use the tab key for its normal HyperCard things anymore, such as moving from field to field. If you never use tab to go from field to field, you won't notice any difference.

Variety pack

Because HyperTalk is so flexible, there are some other variations you might want to use when setting up your command library trigger. To avoid cluttering up this initial chapter, these variations have been herded off together into Appendix A. You'll also find the complete script of the command trigger in Appendix A, including a few more tricks and improvements.

The first command, the angels did say

Now that we've got a wonderful scheme for storing and using our command library, all that's left to do is actually write and discuss the command scripts, a trivial task which will occupy our leisure time for the next few hundred pages or so (don't worry, we promise it'll be fun).

We'll present most of the scripts in the next eight chapters, but before we move ahead, let's write the very first command script for the command library. Then, we'll discuss the arrangement of the following chapters and the scripts you'll find in them.

As we're building and learning about our command library, we'll spend a lot of time looking at the scripts, which are going to be stashed one after another in the command library's stack script, just like so many cars stuck on the freeway at rush hour. For our first script, we'll do one that gets a handler from the command library, displays it, and lets us edit it.


```

1.  on editCommand whichCommand
2.    -- script 1-1: Edit the command library.
3.
4.
5.    -- This handler grabs a handler for you
6.    -- to edit.
7.
8.    set lockMessages to true
9.    go to stack "Cookbook Reference"
10.   if whichCommand is empty then exit editCommand
11.   find whichCommand in field "Handler Name"
12.   if the result is empty
13.   then send mouseUp to bkgnd button "Show Script"
14.   end editCommand

```

The general idea here is to pull in a script from the command library, put it into a field, and display it. After that's done, you can look at it or copy it into your own scripts.

Here's a blow-by-blow description of what happens in this handler. Line 8 turns off messages to prevent closeStack and openStack from being sent as we wander off to the command library. Line 9 actually goes to the Cookbook Reference stack, which is the official way to get into the command library. Line 10 makes sure that we specified a command to find.

Once we're sure that there's something to find, line 11 searches for it in the right place. Line 12 checks to see if it was found by making sure that the result (where HyperTalk reports error messages) is empty. If everything is OK, line 13 pushes the "Show Script" button that's on every card in the Cookbook Reference stack, which takes care of the hard part of grabbing the script for editing.

So, editCommand becomes the very first command script in our library. Once we've installed the command library, we can make handlers like editCommand work by typing into the message box while we're using any stack at all.

Here We Go

OK, now we've put our first command into our command library, so the whole deal is really up and running. The next eight chapters present scripts for the command library and related ideas. Here's the breakdown of where you'll find the scripts:

Chapter 2, *Basic Utensils*, contains scripts that you'll find useful while actually creating stacks and their scripts. You'll be able to use the handlers in this chapter to create a sophisticated, chic, yet friendly and happy environment in which you can create great stacks.

Chapter 3, called *Working with Groups*, provides scripts to help you handle groups of objects and includes group selection of parts and other goodies.

Chapter 4, *Broadcasters*, presents handlers that let you send a single message to a whole bunch of objects at once.

Chapter 5, named *Getting Around*, is all about managing the arrangement of cards in a stack, and various tricks for helping to make it more pleasant to move around stacks.

Chapter 6, *Checking the Ingredients*, provides handlers to help you figure out just how stacks are doing things so that you can learn new techniques for your own creations.

Chapter 7, entitled *Processing Text*, deals with text in your stacks and in the outside world. This includes reading text from other Macintosh files, formatting text, miscellaneous processing, dealing with the arrow keys in an urbane fashion, and so on.

Chapter 8 is called *Getting Organized*, and its handlers give you several techniques for building on HyperTalk's text searching capability.

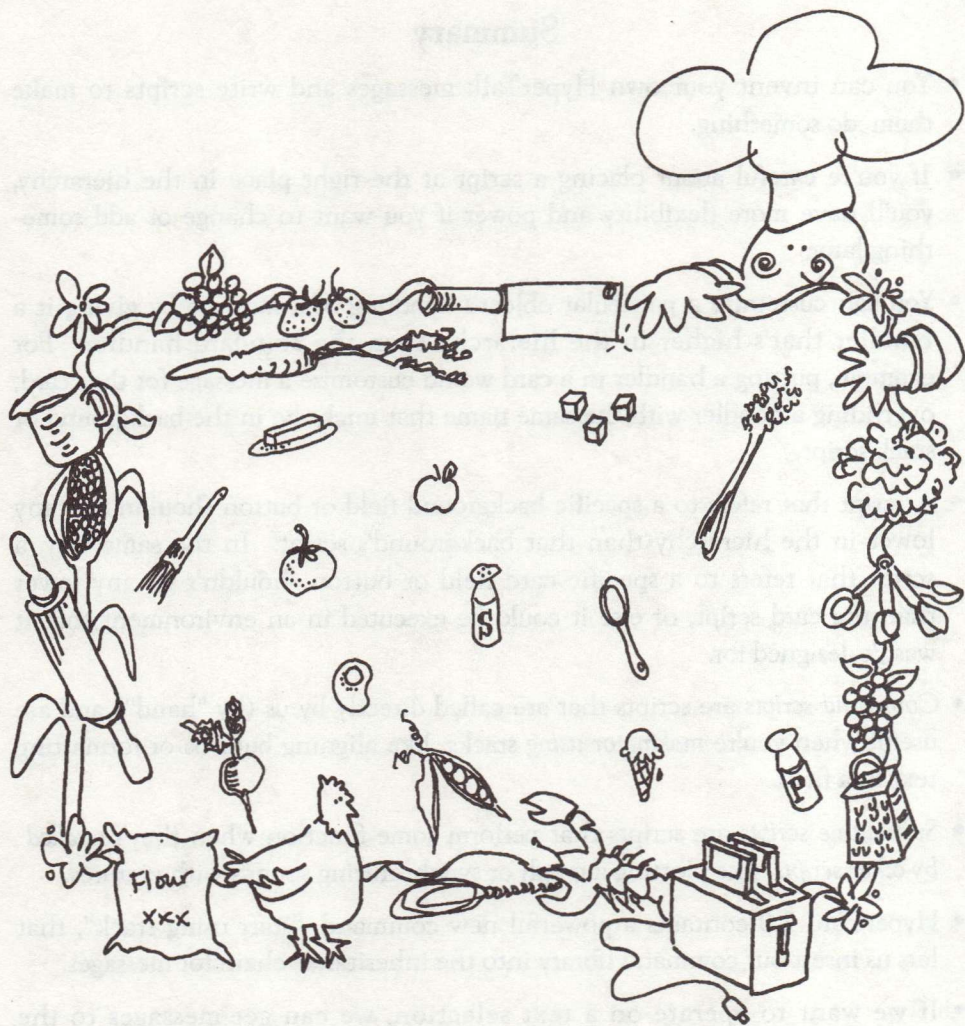
Chapter 9 contains the *Ready-Made Solutions* scripts. This chapter presents subroutine-style scripts that you might find useful in adding functionality to your own stacks.

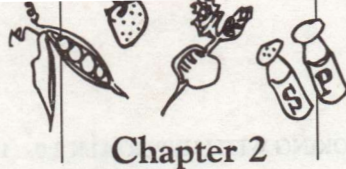
As we proceed through the scripts, we'll present each one and discuss how it works. Because HyperTalk is so rich, there are often various ways of performing the same function. As we describe each script, we'll explain why things were done as they were, discuss possible changes or enhancements, and generally try to give you all the information you'll need in order to really take advantage of the scripts.

Have fun!

Summary

- You can invent your own HyperTalk messages and write scripts to make them do something.
- If you're careful about placing a script at the right place in the hierarchy, you'll have more flexibility and power if you want to change or add something later.
- You can customize a particular object's handling of a message by giving it a handler that's higher in the hierarchy than the standard handler. For example, putting a handler in a card would customize a message for that card, overriding a handler with the same name that might be in the background or stack script.
- A script that refers to a specific background field or button shouldn't go any lower in the hierarchy than that background's script. In the same way, a script that refers to a specific card field or button shouldn't go any lower than the card script, or else it could be executed in an environment that it wasn't designed for.
- *Command scripts* are scripts that are called directly by us (by "hand") and are useful when you're *making or using* stacks, like aligning buttons or formatting text in a field.
- *Subroutine scripts* are scripts that perform some function when they're *called by other scripts*, like plotting a graph or synchronizing sounds with pictures.
- HyperCard 2.0 contains a powerful new command, "start using stack", that lets us insert our command library into the inheritance chain for messages.
- If we want to operate on a text selection, we can get messages to the command library by designating a little-used special keystroke (we're using shift-tab) to bring up a dialog box, then executing a typed-in command.
- There are other possibilities for the command library trigger; they involve the return, tab, enter, function, arrow, option, shift, command, and control keys.
- We wrote the first command script, editCommand. It lets us quickly edit the command library scripts.





Chapter 2

BASIC UTENSILS

This chapter presents the basic set of command scripts to use when making your own stacks or fooling around with those created by other people. They all can be accessed through the command library technique presented in Chapter 1.

A Custom Scripting World

Suppose you're making a great new stack that depicts a map of the world with buttons linking the two-hundred largest cities to more information about those cities. After you've got the whole thing rolling, the leader of the project tells you, yeah, it's great, but it would really be nice if you could use the Palatino font instead of Chicago on those buttons. You visualize how the leader of the project will look as you push a creme pie into his face.

Or, a card you're working on has a really elaborate arrangement of buttons and fields. You need to make the left sides of all the fields line up, and they all have to be the same distance from each other; the buttons should also appear in a nicely staggered stair-step down the screen. You finally go crazy from all this micro-positioning and you have to take a three-hour lunch.

How many times has this happened to you? (Even if your answer is "none," just follow along here.) In the past, when you had to make major repetitive changes like this in most applications, your solution was simple: just redo everything. But HyperCard gives you another choice. Because HyperCard has HyperTalk, you can often write a script to perform these degrading, tedious manual chores, like changing a property for a bunch of buttons.

The problem is that writing a script isn't always the easy way out. Depending on how skillful a scripster you are, you may find it takes as long to write a script to change something around as it does to do it by hand (although writing the script is always a heck of a lot more fun).

One of the bright ideas behind the command library is to try to anticipate these situations, generalize them as much as possible, and write scripts that will

help you out of jams like this. Many of the scripts we'll present in the command library are intended for that kind of use, while others help your HyperCarding in different ways.

General Utilities

There are a few scripts in our collection that provide fundamental functions used in other parts of the command library. In this section, we'll describe these general utility scripts and discuss how they can be used to help implement other neat and dirty tricks.

Automatic scripting

OK, how many of you out there think "automatic scripting" means we've figured out a way for HyperCard to write your scripts if you just think about what you want to do? Well, we might be able to do that with HyperCard version 17.0 or so (scheduled to be released in September of 2018), but for now, we humans have to write the scripts. Automatic scripting refers to a process not quite so cosmic, but still pretty neat and useful, and we'll describe it here.

Here's the idea. You sometimes want to mess around with another object's script programmatically; that is, you want a script to actually modify another script. HyperTalk provides you with the basic magic to do this job. Scripts are properties of objects you can fool around with, and you probably know you can get and set properties like name, textFont, and location. Amazingly enough, you can also get and set an object's script.

Having the power to change a script from within another script is like getting a backstage pass to Disneyland. There's no end to the fun you can have or the damage you can cause. With the ability to modify scripts, you can install new functionality into objects (as we'll do here, since we're responsible people), or you can really mess up the way things are supposed to work.

Joe sent me

You can't always get into stacks and scramble their scripts. If a stack has been assigned a password (with the Protect Stack menu item) and the Private Access option is checked, you won't be able to get your grubby little hands on it, even from within a script, unless you know the password. You might also be able to find it if you're real handy with a disk editor like FEdit.

There are times when it's handy to install a script into another object. One way to do this is to have a complete copy of the script we want to install stashed in the script of another object, like a button. Since a script is just text, we could do something like this:

```
get script of button 1
set script of card 23 to it
```

This would work OK, but it's kind of weird, since it requires us to carry around a button with an executable script not belonging to that button—the button is just a holding place for the script until it gets installed. Another solution would be to put the script into a field as its contents, but this requires us to carry around text in a field, rather than just dealing with scripts. The main hassle with these techniques is that the script and the “installer” (in this case, the two lines of HyperTalk shown above) are separate pieces, and we have to keep track of both of them.

Since the destination script we want to wind up with is just text, one possibility for a slick solution is to develop a script that installs the script by manipulating text. It would consist of several put commands that build up the desired script into a variable, then stick that variable into an object's script. For example, imagine a script to manufacture the script fragment listed above. It would look like this:

```
put "get script of field 1" & return into scriptVar
put "set script of card 23 to it" & return after scriptVar
```

We've now created a script that writes a script! This new script is doing automatic scripting. By using this technique, we can assemble an entire script into a variable, then set the script of the object we want to that variable, and it's all contained in one package—a major advantage. The installer contains the script that's being installed. We have to keep track of only one thing, so there's less chance of losing it.

For most material, translating scripts into this “puttable” format is pretty straightforward. However, there is one case where it gets a little tricky. Think about translating this line into puttable form:

```
go to card "chuck"
```

When you try to convert this into a put command, you have to be slightly

clever (and since you're reading this book, being clever should come naturally to you). You need the cleverness to deal with the quotation marks, because, of course, you're already putting the entire HyperTalk statement into quotes anyway. If you're not paying attention and you try something like this,

```
put "go to card "chuck" " into scriptVar
```

HyperTalk will reward you by telling you that it doesn't know what the heck you're talking about. Instead, you have to use a special tactic for the quotes:

```
put "go to card " & quote & "chuck" & quote into scriptVar
```

This produces the effect we're after. The regular, unquoted parts are put directly, while the quoted part has the quotes explicitly added before and after the characters themselves. It makes writing the installer script a little tougher, but it also ensures that it will work with quoted strings.

You may already have noticed a potential drawback to this technique. Imagine taking a long script and trying to cast it into this form. You could go nuts. The chore of converting a script into puttable form is repetitive and tedious, exactly the kind of work people aren't very good at, but that computers have a real knack for. What we need is a script that will generate these scripts for us.

Automatically generate installer scripts

Of course, if we've bothered to say that we need such a script, we've probably already gone ahead and written it (after all, that's why you bought this book). The handler, which is named `writeScript`, looks like this:

```
1. function writeScript sourceScript,varName
2.    -- script 2-1: Automatic scriptwriting
3.
4.
5.    if varName is empty then put "scriptVar" into
        varName
6.    put quoteLines(sourceScript) into resultScript
7.    repeat with i = 1 to the number of lines in
        resultScript
8.        put "put " before line i of resultScript
```



```

9.      put " after " & varName after line i of
        resultScript
10.     end repeat
11.     put "put empty into " & varName & return before
        resultScript
12.     return resultScript
13. end writeScript

```

Handler name:

writeScript

What it does:

The writeScript handler translates a script into “puttable” form. Given a script as input, writeScript produces a set of HyperTalk commands that can recreate the input script. The script that writeScript produces is called a *manufactured* script. Since writeScript is defined as a function, the script that’s generated is returned as the function result. For extra credit, think about this for a second: writeScript is a script that writes a script that writes a script.

Parameters:

The first parameter, sourceScript, is the script that’s going to be cast into puttable form. The second parameter, varName, gives the name of the variable that will be used in the installation script to hold the recreated script. In our examples above, we used scriptVar for this purpose, and if you don’t pass anything for the second parameter, writeScript is set up to use scriptVar as the default.

Here’s an example. Assume that you have a button named “Driver 8,” and it has this script, left over from Chapter 1:

```

on mouseUp
    hide card window
    play "Sur la Mer"
    show card window
end mouseUp

```

If you sent the command


```
put writeScript (script of btn "Driver 8") into cd fld 1
```

you'd get the following text in card field 1:

```
put empty into scriptVar
put "on mouseUp" & return into scriptVar
put "  hide card window" & return after scriptVar
put "  play " & quote & "Sur la Mer" & quote & return after
  scriptVar
put "  show card window" & return after scriptVar
put "end mouseUp" & return after scriptVar
```

How it works:

Line 5 checks to see if you've passed a parameter for the name of the variable that's supposed to hold the auto-generated script. If not, it supplies scriptVar as the default name. Line 6 does much of the work. It converts the entire script into a HyperTalk string, using the quoteLines function (which we'll describe next). The quoteLines function properly handles quoted strings, just as we showed above, and it uses the HyperTalk constant "return" to indicate the end of each line. When the script comes back from quoteLines, it has quotes around each line, and the whole thing is stored in the variable resultScript.

Now that the text of the script has been turned into a big sequence of puttable strings by quoteLines, the rest of writeScript adds the final touches. We have to add the word "put" to the start of each line, and we need to tack on "after" and the name of the script holder variable to the end of each line.

The repeat loop in lines 7 through 10 executes once for each line in the script. Line 8 inserts the word "put" at the front of each line of the script. Line 9 adds the words that we need at the end of each line, which consist of "after" and the variable name for each line that we manufacture. In line 11, we make sure the variable that's going to hold our script starts out empty. Line 12 designates resultScript as the value the function will return.

The real heart of writeScript is the quoteLines utility, which translates the source script into puttable form. Here's what quoteLines looks like:

1. function quoteLines string
2. -- script 2-2: Convert text to a quoted string
- 3.
- 4.


```

5.  put empty into quotedString
6.  repeat with i = 1 to the number of lines in string
7.    get line i of string
8.    if it is empty then put "return" & return after
        quotedString
9.    else if quote is not in it
10.   then put quote & it & quote & " & return" & return
        after quotedString
11.   else
12.     put quote into expandString
13.     put " & quote & " & quote into calcQuote
14.     put number of chars in it into charCount
15.     repeat with j = 1 to charCount
16.       put char j of it into thisChar
17.       put thisChar after expandString
18.       if thisChar is quote then put calcQuote after
           expandString
19.     end repeat
20.     put expandString & quote & " & return" & return
        after quotedString
21.     get " & " & quote & quote
22.     repeat
23.       put offset(it,quotedString) into emptyCat
24.       if emptyCat is 0 then exit repeat
25.       delete char emptyCat to emptyCat+length(it)-1
           of quotedString
26.     end repeat
27.   end if
28. end repeat
29. return quotedString
30. end quoteLines

```


Handler name:

quoteLines

What it does:

This function takes a script as input and puts it into a form that will work with put commands to reconstruct the script. This involves transferring most of the text between quotation marks, preserving the quotation marks in strings that are already quoted, and handling the end of each line.

Parameters:

QuoteLines takes a single parameter, which is the string that's going to be processed. Its function result is the quoted string.

To see how much work quoteLines does, we'll use the same example as we did for writeScript. Typing

```
put quoteLines (script of btn "Driver 8") into fld 1
```

puts the following text into field 1:

```
"on mouseUp" & return
"  hide card window" & return
"  play " & quote & "Sur la Mer" & quote & return
"  show card window" & return
"end mouseUp" & return
```

By comparing this text to the result we got from calling writeScript with the same parameter in the preceding section, you can see just what part of the work is done by each function. For extra fun, you can use the HyperTalk debugger to watch quoteLines building up its strings. Try putting a checkpoint on line 20 and watching the value of quotedString.

How it works:

The variable used to collect the converted string is called quotedString, and line 5 makes sure that it starts out empty. Most of the function (lines 6 through 28) constitutes a giant repeat loop that processes the lines in the input string one at a time. Every time it seizes a line, quoteLines classifies it as one of three kinds of lines:

1. Empty (easy).
2. Not empty, contains no quotation marks; that is, no "quote" strings (still pretty easy).
3. Contains quotation marks (a pain).

After figuring out which of these three categories a line fits into, the script processes the line and adds something to the output string. Line 7 grabs the current line of the input string and moves it into the local variable called "it" (HyperTalk's get command always places whatever it gets into this famous variable).

Line 8 checks for the special (piece of cake) case of a blank line in the input string. If the line is blank, a corresponding blank line has to go into the output. Exactly how the end of a line is handled is a little bit interesting. Remember that the output of quoteLines is a quoted version of the input string, so the end of a line has to be represented by "& return." If we just kept stringing lines together with HyperTalk's ampersand (&), our output would become a massive collection of little strings connected together with ampersands, like this:

```
"on mouseUp" & return & "  hide card window" & return &
" play " & quote & "Sur la Mer" & quote & return & "  show
card window" & return & "end mouseUp" & return & "ad
infinitem" & return
```

This might be OK to look at in a field (it automatically wraps text to a new line when it gets too long), or to anyone who has a monitor that's twelve feet wide. It might even be interesting for people who enjoy being confused, but our goal here is to produce text that's going to become part of an installer script, a whole sequence of HyperTalk put commands. So it would be nice to break up the text with a new line every so often so that a poor human reader can see what's going on.

With that in mind, there is a reasonable, natural place to insert these new lines. We can create a new line of output every time we move to a new line in the input string. So, when we reach the end of a line in the input string, we have to do two things: first, we have to tack on the characters "& return" to our output string, and second, we have to insert an actual return into the output

string. In `quoteLines`, you can add these elements with a code fragment that looks like this:

```
put something & "& return" & return after quotedString
```

The first `return`, the one that's between the quotes, will produce a new line when the final script is generated, while the second `return` simply creates a new line in `quotedString`, the output of this function.

Getting back to our journey through the `quotedLines` script, we were looking at line 8, which checks to see if the current line in the input string is blank and takes care of things if it is. If it's not blank, the *else* in line 9 takes over. Note that this *else* clause extends all the way to the end of the repeat loop. In other words, if the input line is blank, `quoteLines` will execute line 8, then jump directly past the *else* to the end of the *if* statement in line 27.

What's up front

Taking care of the short "ifs" up front and leaving the complicated ones for the end makes scripts more readable for us humans. With this style, you can get the easy things out of the way first, then concentrate on the hard stuff later.

If the input line isn't empty, line 9 checks to see if it contains a quotation mark (if quote is not in it), using the built-in constant named `quote`. If there are no quotation marks in the line, we've still got a pretty simple case that's handled by the code in line 10, which just copies the text to `quotedString`, surrounded by quotes (put `quote` & it & `quote`), then adds the line break to the quoted string and the new line for the output string itself (& " & `return`" & `return` after `quotedString`). Whew!

As if in anticipated dread, line 11 simply reads "else," because here comes the hard part of this handler—dealing with lines that have quoted strings in them (insert scary music here). Actually, it's not all that bad. Here's the basic strategy: we're going to transfer characters, one at a time, from the source (the script in the variable "string") to the destination (temporarily kept in `expandString`). Every time we transfer a character, we'll check to see if it's a quote. If so, we have to add something to `expandString`. Here's what we add:

```
& quote & "
```


This effectively changes all the quoted strings in the source into properly quoted puttable strings. After the entire line of the source script has been processed, the closing quotation mark, “& return,” plus the actual return are added to the manufactured script.

It don't come easy

Sometimes, when reading about intricate ideas like the one we're talking about now, you might feel lost in the details. If that happens to you, don't worry too much about it. You can still get full use of the tools in the command library without having to understand how they work. We explain the techniques behind them so that you can learn new ideas for scripting, modify the tools if you want to customize the way they work, get full value from this book, and just because we're nice guys. Also, if you're really interested in learning more about how the scripts work, you can use the incredible HyperTalk debugger, which lets you look at the execution of handlers at a more leisurely, human rate of speed, watching as variables change all the while.

Here's the key to what happens where. We said that the puttable line will be constructed in the variable named `expandString`. Line 12 starts out the line with the character that starts every line `quoteLines` constructs, a quotation mark, which we can get to through the HyperTalk constant named `quote`. We also said that we'd be adding

```
& quote & "
```

whenever we encounter a quotation, so for convenience, line 13 sticks that string of characters into a variable named `calcQuote`. Line 14 finds out how many characters are in the line we're about to process, so we don't have to keep going on forever and possibly miss lunch.

Lines 15 through 19 make up the loop that processes each character. Line 16 gets the current character into the `thisChar` variable for easy messing around, and line 17 sends the character directly over to `expandString`. Line 18 checks to see if it was a quotation mark; if so, it adds `calcQuote` (that's the after-quote chunk we defined earlier) to the end of the string it's building.

When we get to line 20, we've processed the entire line, so it's time to add the end of line stuff, a closing quote, the string "& return" to generate a new line in the final script, and a return to start a new line in the manufactured script.

Let's take a look at the lines this technique produces. Feeding quoteLines a source line that looks like this

```
put "The Wrong Child"
```

will produce a puttable line that looks like this

```
"put " & quote & "The Wrong Child" & quote & "" & return
```

Pretending

You may certainly find it challenging to try to understand exactly how quoteLines works—I know I did when Dan showed it to me! One tool that you may find helpful if you're trying to understand it is a time-honored technique called "playing Mr. Computer." To do this, you get out a piece of paper (you know, the old-fashioned analog kind) and you make a heading for each variable as you encounter it. Then, you trace through the script, line by line, writing down the value of each variable as it changes (see Figure 2-1). By doing this, you can see exactly how things are happening in exactly the same way as HyperTalk does (except that you're running a few thousand times slower). Playing Mr. Computer can help you unlock the secrets of the most mysterious scripts.

If you're not the Mr. Computer type, you can use HyperCard's own built-in version of Mr. Computer, the HyperTalk debugger. This nifty feature allows you to step through scripts at your leisure, and it's invaluable in helping you find bugs or watch what's happening in your scripts.

<u>myVar</u>	<u>sting</u>	<u>today</u>	<u>color</u>	<u>animal</u>
1	7	here	red	dog
2	18	gone	green	eat
3	11		blue	turtle
4	9			horse
	4			

Figure 2-1 Playing Mr. Computer

You can check this line with Mr. Computer (described in the box labeled “Pretending,” or you can use the HyperTalk debugger to make sure that’s what you get, if you’re the type who always checks your order at the fast-food restaurant to make sure you got the right thing. An interesting side effect of our technique shows up whenever the source line ends with a quoted string. If you look at the puttable line that quoteLines produced, you’ll see that it includes this near the end:

```
& ""
```

When this chunk is expanded into the final script, what does it produce? Nothing, nada, zilch, bupkus. It’s added on by the quote processor that we’ve built into quoteLines, even though, at the end of the line, it doesn’t do anything except take up space. The manufactured line will work exactly the same if this expression is removed. So, let’s remove it!

That’s just what lines 21 through 26 are designed to do. Line 21 starts by sticking space-ampersand-space-quote-quote into “it.” Our mission, and we’ve decided to accept it, is to seek out and destroy any strings like this that we find, because they’re useless.

Lines 22 through 26 are repeated until we don’t find any more of these strings in the newly produced line. Line 23 uses HyperTalk’s offset function to see if the manufactured line has the desired string in it. If it does, offset will put the character position where the string lies into the emptyCat variable. This variable’s name comes from the fact that neither of the authors like cats, and...no, just kidding (don’t send us any nasty letters, please). The variable’s name really means that this string adds (or concatenates, in more programmerly terms) the empty string. If the empty-adding string isn’t there, offset will return zero, and we’ll quit the loop.

Line 25 handles the case where we found an emptyCat string. It deletes characters starting from the location of the string (which is in emptyCat), and continues for enough characters to remove the entire string (“length (it) -1”). We have to add the “-1” part because the first character is deleted by specifying “char emptyCat.” If we then delete “length (it)” more characters, we’ll get rid of one character too many.

Now we’re really, truly at the end of the script. Line 27 ends the big giant if statement that started way up in line 8 (the one that figures out whether the line is empty, has no quotes, or has quotes). Line 28 ends the even bigger, even

giant repeat loop that started on line 6 and allowed us to grab every line of the input string. At last, in line 29, we return `quotedString` as the function result, since `quoteLines` is a function, and then we get to rest.

Note that `quoteLines` preserves the indentation spaces that come at the beginning of lines, even though they don't change the way the manufactured script will work (see Figure 2-2 for examples of what we're talking about here). This is so you can still see the rough format of the script even when it's in puttable form, and it uses up very little additional disk space. And you thought it was a bug or an oversight! (Anyway, if you don't like the spaces, a fun exercise for you is to change `quoteLines` so that it removes them from the script's puttable form.)

```
put "on mouseDown" & return into scriptVar
put "  global selectedParts" & return after scriptVar
put "  if " & quote & "button" & quote after scriptVar
put "  then put " & quote & "No part" & quote & return
    after scriptVar
put "  else" & return after scriptVar
put "    get the target" & return after scriptVar
put "    if " & quote & "id" & quote & " not in " & return
    after scriptVar
put "    then put " & quote & "id " & quote & " & id of the
    target
```

Figure 2-2 Indentation preserved

Hard times

We could also use “delete char emptyCat to emptyCat+4,” putting the value of “length (it) - 1” directly in the script. This is slightly bad form, though. It's called “hard-coding,” and it's bad for two reasons. First, it relies on us scripters (mere humans) to correctly count the number of characters in “it,” when HyperTalk is much more suited to doing that kind of thing, and it will never make a mistake doing so. Second, if we ever find that we need to change the definition of an emptyCat string, we'd have to have a wonderful memory to remember to change the 4 to whatever the new value is. Once again, rather than depend on our unreliable human memory, we're better off letting HyperTalk handle it for us automatically by using the reference to the length of “it.” This also makes the script easier for us humans to read.

Miscellaneous Fun Gimmicks

In this section you'll find interesting tricks and ideas to provide various capabilities for your HyperCard world. (Scripts found in this section didn't really fit in anywhere else, either, even with our rather liberal labeling system.)

Ask before deleting any button with a script

Have you ever deleted something and wished you hadn't? If not, put this book aside for a few minutes and go delete something that you'll later wish you hadn't. This handler helps in cases where you're about to delete a button, but you've forgotten that the button actually contains some marvelously clever script that you'd really hate to lose. This feature can be installed with the Cookbook Reference stack. By default, it's not installed when you install the command library.

```

1. on protectButton whichButton
2.   -- script 2-3: Button script protection.
3.
4.
5.   -- This handler makes the user confirm before
6.   -- deleting any button that has a script.
7.
8.   if script of whichButton is empty
9.     then exit protectButton
10.  answer "Delete " & whichButton & "?" with "OK" or
    "Cancel"
11.  if it is not "OK" then select empty
12. end protectButton

```

Handler name:

protectButton

What it does:

Whenever a button is deleted, the system sends the deleteButton message to the button just before it buys the farm. Of course, most buttons don't have handlers to catch this message, so it continues down the inheritance chain, usually being scarfed up by HyperCard at the end of the road and just ignored. Not any more, though. Our deleteButton handler hangs out in the command library, which is

installed in the inheritance chain with the “start using stack” command. Since `deleteButton` is rarely caught anywhere else, our handler almost always gets control when a button is deleted.

When it does get control, our handler checks to see if the button that’s about to go away has a script, which is a kind of litmus test to see if a button is very important. If it has a script, our handler asks the user to confirm that the button really should be deleted.

Note that we don’t ever call this handler directly. Instead, it’s called automatically by the system whenever a button is deleted and HyperCard sends a `deleteButton` message.

Emptiness inside

If you create a new button, then edit its script, you’ll see that the script editor automatically fills in the “on mouseUp” and “end mouseUp” statements for you. This seems to suggest that every button has a script, and that our delete-catcher will always be activated. But that’s not really the case. Even though every button seems to be born with these two lines, they’re really only presented by the script editor. You can tell because the script editor kindly places the insertion point on the blank line between the phantom statements. If there’s a real script, the script editor will put the insertion point at the beginning of the script. Until you add to a button’s script, the “on mouseUp” and “end mouseUp” lines aren’t in the script. They’re only shown by the script editor. To prove this, choose “New Button” from the Objects menu, then type

```
script of last button is empty
```

into the message box. (Saying “last button” is guaranteed to refer to the most recently created button, assuming we haven’t moved any of them closer or farther.) This statement should return “true,” meaning that the button’s script is really empty. Whenever you really want to know the truth, just ask HyperTalk.

Parameters:

This handler takes one parameter, which is the name of the button that's about to get killed, passed on by the `deleteButton` handler in the Command Reference stack.

How it works:

The first thing our handler does (line 8) is check for the condition we're interested in, which is whether the button has a script or not. If it doesn't, we don't care about it, and in line 9 we just exit, thus sending the poor part to oblivion, since there's probably nobody else to catch the handler before mean old HyperCard takes it and deletes the button. If the button has a script, it gets a second chance in line 10 as we ask the user whether the button should really be deleted.

Won't come back from Dead Man's Curve

We don't have to use an "else" clause to decide whether to execute the rest of the handler following the "then" statement in line 9 because of the "exit" message in line 9. "Exit" ends execution of the handler right there, so there's no need to follow it with a further test like "else."

If the user decides it's a bad idea after all to try to delete the button, we perform the real magic of this handler at the end of line 11. When a button is about to be deleted, it's always selected, and it happens that HyperCard deletes the button that's selected after sending the `deleteButton` message. So, we can derail the impending deletion by releasing the button from selection. The easiest way to do that is to empty the selection with the "select empty" message. Then, when the handler ends, the button isn't selected any more, and it won't be deleted.

This handler works because of a little flaw in HyperCard's execution of the delete command, and it's probably something you'd only figure out after looking at HyperCard from the inside out. When you ask to delete something, it really should delete it—it shouldn't matter if something else is selected before the deletion actually happens. But we believe in serendipity, so we've managed to turn this bug into a nice, useful feature. This is also a basic principle of market-

ing, like changing slow-pouring ketchup from a liability into an asset. When you're given a lemon, make lemonade!

Automatic stack compaction you can shut off

HyperCard's compaction feature helps keep stacks small. For maximum flexibility, stacks are never compacted automatically, but as they change, free space in the stacks ("air," my brother Ken calls it) accumulates and you get more reason to compact. The handler in this section provides a solution that's easy and convenient, and gets out of the way when you want it to. If you want to install automatic stack compaction, use the automatic stack compaction card in the Cookbook Reference stack.

```

1. on autoCompact
2.   -- script 2-4: Automatic stack compaction that can
3.   -- stop asking
4.
5.   -- Whenever you leave a stack that has too much
6.   -- free space, this handler will automatically ask
7.   -- if you want to compact it. You can tell it to
8.   -- shut up, too.
9.   global autoCompactEnabled,maxFreeSizeOfStack
10.  if autoCompactEnabled is not true then exit
      autoCompact
11.  if cantModify of this stack then exit autoCompact
12.  get the freeSize of this stack
13.  if it > maxFreeSizeOfStack then
14.    put "Compact this stack? (" & it div 1024 & "K
      free)" into prompt
15.    answer prompt with "Yes" or "No" or "Stop Asking"
16.    if it is "Stop Asking" then put false
      into autoCompactEnabled
17.    if it is not "Yes" then exit autoCompact
18.    doMenu "Compact Stack"
19.  end if
20. end closeStack

```


Handler name:`closeStack`**What it does:**

As your stacks accumulate free space, it's nice to have HyperCard compact them for you automatically. On the other hand, maybe you'd rather not bother squeezing the air out of stacks when there's just a little bit of free space. On the other hand, sometimes you're just interested in getting done with your work as quickly as possible, and having a script asking you if you want to compact every time you switch stacks can be very annoying.

This handler is an attempt to compromise all of these conditions. Whenever you leave a stack, it checks to see how much free space that stack has. If it's more than a minimum threshold that you've selected, it will put up a dialog telling you what's going on and asking if you want to compact. This dialog gives you three choices: compact the stack, don't bother, or stop asking me to compact, already. The last choice causes the handler to leave you alone until you quit and restart HyperCard.

Parameters:

None. You don't call this handler directly. It's called whenever a stack is closed; that is, whenever you go to a new stack, it's called just before you leave the old stack.

How it works:

This `closeStack` handler uses two globals. The first one, `autoCompactEnabled`, is completely managed by this handler. It's set to false if the user doesn't want to be bothered with being asked to compact any stacks at all. The second global is named `maxFreeSizeOfStack`. This one should be declared as a global and set in the Cookbook Reference stack to the minimum amount of free space you want to know about. A reasonable value for this variable is 20000, which means you'll be notified whenever there's about 20K of air in your stacks, but you can set it to be as frugal as you want.

Lines 10 and 11 check for a couple of conditions that will prevent us from compacting the stack. Line 10 tests to see if `autoCompactEnabled` has been turned off. If so, we give up immediately by passing the `closeStack` message in case someone else farther down the message chain wants it. Otherwise, the

handler continues and checks if the stack is set to `cantModify`, which means that an attempt to compact it would fail anyway.

Once we're past these initial checks, we need to see how much free space there is in the stack (line 12) and check it against the minimum set in `maxFreeSizeOfStack` (line 13). If it's greater than the minimum, we're in business. Line 14 sets up a variable containing the question we're going to ask, plus the approximate amount of free space in the stack. This value is computed with HyperTalk's `div` operator, which performs a special kind of division—it divides, then drops any remainder. In line 14, it divides by 1024, which is the number of bytes in 1K byte, of course.

After we've assembled the prompt in line 14, we put up a dialog in line 15 that gives the user three choices. If the harried user is in a hurry and chooses "Stop Asking," we put out the "Do Not Disturb" sign and set `autoCompactEnabled` to false, which will prevent this handler from even asking the question from now on (until HyperCard quits and is restarted, anyway). If the user didn't click the Yes button, the `closeStack` message is passed. Otherwise, at last, line 18 actually compacts the stack, the handler ends, the air is squeezed out, and all is well.

No action

This handler, like all others that handle common messages like `closeStack`, can be short-circuited if any object farther up the chain has a corresponding handler and doesn't pass the message on. For example, if you have a stack with a `closeStack` handler that puts up a goodbye message, the automatic compaction won't work unless you include a "pass `closeStack`" statement.

Make a note field

Card fields are one of the niftiest features in HyperCard. In more general database terms, a card field lets you have a field that appears only on a particular record (card), a pretty unusual capability that can come in handy. One good use for card fields is to make some notes or comments that belong just to a specific card. The `newNote` handler creates a card field on the current card, then injects some handlers into it to make it especially useful for note taking.


```

1. on newNote
2.   -- script 2-5:  Make a note field.
3.
4.
5.   -- This command creates a new card field that's
6.   -- useful for making notes about the current card.
7.
8.   lock screen
9.   set editBkgnd to false
10.  doMenu "New Field"
11.  set style of last card field to scrolling
12.  set textSize of last card field to 10
13.  choose browse tool
14.  unlock screen
15.  select text of last card field
16.  -- user now thinks work is done
17.
18.  put "on mouseDown" & return into noteFieldScript
19.  put "  if the commandKey is not down then exit
      mouseDown" & return after noteFieldScript
20.  put "  " & return after noteFieldScript
21.  put "  if width of me > 15 then" & return after
      noteFieldScript
22.  put "  if lockText of me is false then exit
      mouseDown" & return after noteFieldScript
23.  put "    set locktext of me to false" & return after
      noteFieldScript
24.  put "    click at the clickLoc" & return after
      noteFieldScript
25.  put "    exit to HyperCard" & return after
      noteFieldScript
26.  put "  end if" & return after noteFieldScript
27.  put "  " & return after noteFieldScript
28.  put "  -- drag me to the mouseLoc" & return after
      noteFieldScript

```



```
29. put " repeat while the mouse is down" & return after
    noteFieldScript
30. put " set loc of me to the mouseLoc" & return after
    noteFieldScript
31. put " end repeat" & return after noteFieldScript
32. put " exit to HyperCard" & return after
    noteFieldScript
33. put "end mouseDown" & return after noteFieldScript
34. put return after noteFieldScript
35. put "on mouseUp" & return after noteFieldScript
36. put " if the width of me > 15 then makeMeSmaller" &
    return after noteFieldScript
37. put " else makeMeLarger" & return after
    noteFieldScript
38. put "end mouseUp" & return after noteFieldScript
39. put return after noteFieldScript
40. put "on closefield" & return after noteFieldScript
41. put " makeMeSmaller" & return after noteFieldScript
42. put "end closefield" & return after noteFieldScript
43. put return after noteFieldScript
44. put "on enterinfield" & return after noteFieldScript
45. put " makeMeSmaller" & return after noteFieldScript
46. put " " & return after noteFieldScript
47. put " -- delete me if i'm empty" & return after
    noteFieldScript
48. put " if me is return then" & return after
    noteFieldScript
49. put " lock screen" & return after noteFieldScript
50. put " select me" & return after noteFieldScript
51. put " doMenu " & quote & "Clear Field" & quote &
    return after noteFieldScript
52. put " choose browse tool" & return after
    noteFieldScript
53. put " unlock screen with dissolve" & return after
    noteFieldScript
54. put " end if" & return after noteFieldScript
```



```
55. put "end enterinfield" & return after noteFieldScript
56. put return after noteFieldScript
57. put "on makeMeLarger" & return after noteFieldScript
58. put " lock screen" & return after noteFieldScript
59. put " delete line 1 of me" & return after
    noteFieldScript
60. put " put the topLeft of me into saveLoc" & return
    after noteFieldScript
61. put " set height of me to 100" & return after
    noteFieldScript
62. put " set width of me to 180" & return after
    noteFieldScript
63. put " set the topLeft of me to saveLoc" & return
    after noteFieldScript
64. put " set style of me to scrolling" & return after
    noteFieldScript
65. put " unlock screen" & return after noteFieldScript
66. put "end MakeMeLarger" & return after noteFieldScript
67. put return after noteFieldScript
68. put "on makeMeSmaller" & return after noteFieldScript
69. put " lock screen" & return after noteFieldScript
70. put " put return before me" & return after
    noteFieldScript
71. put " put the topLeft of me into saveLoc" & return
    after noteFieldScript
72. put " set height of me to 12" & return after
    noteFieldScript
73. put " set width of me to 12" & return after
    noteFieldScript
74. put " set the topLeft of me to saveLoc" & return
    after noteFieldScript
75. put " set locktext of me to true" & return after
    noteFieldScript
76. put " set style of me to shadow" & return after
    noteFieldScript
77. put " unlock screen" & return after noteFieldScript
```



```

78.  put "end makeMeSmaller" & return after
      noteFieldScript
79.  set script of last card field to noteFieldScript
80.  end newNote

```

Handler name:

newNote

What it does:

This handler, originally written by Robin Shank and modified by Dan, creates a new card field on the current card and gives it a script containing a few handlers. The handlers in this script, which are listed and explained below, customize the field for writing down notes or comments about that card.

The note field appears in two forms: shrunk to a small box, and fully expanded, revealing its text. When the field first appears, it's in the big form, unlocked and ready for typing. After you're done typing, if you close the field (such as by clicking outside the field or pressing enter), it shrinks to the small size.

When the field is small, you can reposition it on the card by holding down the command key and dragging it around, or you can expand it to full size again by just clicking on it. When it's big, its text is locked. Command-clicking on it will unlock the text. Clicking without the command key or closing it will reduce it again. Finally, if you type enter when the field is empty, it vanishes.

That was pretty complex, so here's a summary of what it does:

State	Action	Effect
new (big)	click	selects text
new (big)	close field	makes it small
new (big)	command-click	makes it small
small	click	makes it big
small	command-drag	moves it around
big	click	makes it small
big	close field	makes it small
big	command-click	unlocks text
big, empty	close field	field vanishes

To leave a field displayed in the big state, zoom it open, then click outside the field without changing its contents. This will prevent a `closeField` message from being sent.

After you've got a note field created, try stepping through its script with the HyperTalk debugger to watch what goes on.

Parameters:

None.

How it works:

This handler really has two jobs. First, it creates the card field that's going to be used for note taking, and then it manufactures the appropriate handlers and puts them into the field's script. Before it does any of that, it locks the screen in line 8 to hide all the messy visuals from the user until everything is ready.

We need to make a card field here, so we have to be sure that the user isn't currently working on the background, and line 9 takes care of that. Line 10 actually creates the new field that's going to be the center of our attention. Lines 11 and 12 make sure the field can scroll and set the font size to 10 points. If you want a different size, or even if you'd like to use a different font, you can put something different into the script right here.

After the field is created, line 13 gets the user back to the browse tool, and line 14 unlocks the screen, allowing the user to see the beautiful new field that we've created. At this point, it looks to the user as though all the work is done. Actually, the script keeps running for three more pages!

Next, in line 15, we want to move the insertion point to the new field so the user can type into it. There are a couple of interesting things going on in line 15, even though it looks pretty innocent. What will the "last card field" be? Can we be sure that this will refer to the field we just created?

The answer is yes. When you create a new field or button, HyperCard guarantees that you can refer to it by using the "last" ordinal, at least until you create another object of the same type. This makes it very convenient to do something with a field or button you've just made.

Script tips: The last one

If you look through HyperCard's menus, you'll find five different "New" commands. After you use any of these, either from a script or via the message box, there are easy, guaranteed ways to refer to the object you just created. To refer to a newly-created field or button, use "last field" or "last button," inserting "card" or "background" if necessary in front of "field" or "button." When you create a new card, background, or stack, that card, background or stack becomes current, so you can refer to it with "this," as in "this card."

The other interesting thing in line 15 is that it says to select the text of a field just created—a field that must be empty, because it's brand new. Why do this? Because it will place the insertion point in the new field so that the user can immediately begin typing there.

Pop quiz

What would happen if line 15 said "select last card field?" See if you can guess, and then use HyperCard to find out what really happens.

Now we're all done making the field, but we haven't put the script into it yet. The rest of this handler takes care of building the field's script and installing it. If we're not done yet, why did we go ahead and unlock the screen? Why not wait until all the construction work is really done before having the grand opening?

Because although the field isn't quite ready yet, it *looks* like it is. Its appearance is complete, so we can go ahead and show it to the user. Usually, folks won't start typing right away when the field appears. Its appearance stuns users into inaction for a moment while they figure out what's going on, so HyperCard gets the extra fraction of a second it needs to finish building the field and install its script. Showing some visual progress can make the user feel like things are running along pretty fast.

Even if the user is a blazing supertypist and starts pounding away at the key-

board before the field is open for business, the Macintosh keeps track of keystrokes (“buffers” them, if you’re into fun jargon) until HyperCard is ready to accept them, and no typing is lost. By unlocking the screen as soon as it *appears* to be ready, we’re gaining a precious extra fraction of a second in returning control to the user, and that’s always a nice thing to do, especially since it makes the user feel like things are happening faster.

The rest of the handler, lines 18 and on, installs the handlers for the field’s script using the “write a script” trick we talked about in Chapter 2. We’ll discuss the scripts to be manufactured by these lines. So, for example, when we talk about line 19, we’ll mean the line of script that will be manufactured by line 19. Got that?

The first manufactured handler is `mouseDown`. We only want to do something special with `mouseDown` if the command key is down, so line 19 checks for that. Line 21 determines if the field is big or small by checking its width. If it’s big, then line 22 checks if the field is locked. If not, it exits the handler. If it is locked, line 23 unlocks it and line 24 places the insertion point. This lets you unlock a previously created note field by command-clicking. Line 25 avoids any possible `mouseUp` handlers by exiting directly to HyperCard.

We’ve handled the case of the big field. What if we get a mouse click when the field is small? Lines 29 through 31 deal with that. They simply set up a repeat loop that keeps going as long as the mouse button is down. While it’s down, line 30 cleverly keeps moving the field to the mouse’s location. This branch of the handler also ends with an “exit to HyperCard” in line 32, once again bypassing any `mouseUp` handlers.

The next handler is the `mouseUp` that the `mouseDown` handler was so careful to avoid. In general, we want `mouseUp` to switch the field between small and big. The simple “if” statement in lines 36 and 37 does that by calling handlers named `makeMeSmaller` and `makeMeLarger`, which we’ll get to soon.

The `closeField` handler in lines 40 through 42 is very simple, too. It just calls `makeMeSmaller` when there’s a `closeField`. This ensures that the field will shrink after you’re done editing its text.

Line 44 starts the `enterInField` handler. The `enterInField` message is sent when the user types enter while the field is open (you probably could have guessed that). When this happens, we want to shrink the field, and make it vanish if it’s empty. Line 45 starts by making the field small.

Now, we want to get rid of the field if there’s nothing in it. To do that, we

need to reveal a little secret. When note fields are made small, we insert a return at the beginning of the field. This is done so you won't see a piece of the first character in the field when the field is small. Without that return, you get a tiny unsightly chunk of the first character, and it looks terrible. So, we add a return when we shrink a field, and take it off when we enlarge a field. You'll see how this works in `makeMeSmaller` and `makeMeLarger`.

Given that inside information, we know that an "empty" field really contains a return when it's small, so line 48 is really testing for an empty field. If it's empty, we have to kill it. We lock the screen (line 49), select the field (line 50), get rid of the field (line 51), return to the browse tool (line 52), then add a little flourish by dissolving to the field-less screen (line 53).

After our fancy field disappearing act, there's nothing more for this handler to do. Line 54 closes the "if" statement that checked if the field was empty, and that's all.

The last two handlers in the field script are `makeMeLarger` and `makeMeSmaller`. `MakeMeLarger` starts in line 57. It begins by locking the screen and removing that secret leading return we just talked about (line 59). Then it saves its `topLeft` property, enlarges its height and width, and restores its `topLeft` (lines 60 through 63). In line 64, it turns on the scroll bar for the field. Line 65 finishes the work by unlocking the screen, and line 66 formally ends the handler.

`MakeMeSmaller`'s work is very similar, but in the opposite direction. It locks the screen (line 69) and inserts the telltale return (line 70). Then, it shrinks the field to a mere 12 by 12, preserving the `topLeft` (lines 71 through 74). Line 75 ensures that the field is locked, since the field might just have been created, in which case its text would be unlocked. Line 76 effectively removes the scroll bar and also gives the tiny field a neat-looking drop shadow. Line 77 unlocks the screen, and that's the end of the whole script, at last.

If you have any other special tricks you'd like your note fields to contain, just add handlers to the `newNote` script. For example, you might want to time-stamp each note field, putting the time it was created into its first line. To do that, you can add statements to `newNote` to take care of this operation when the field is created. Adding a `newField` handler to the note field won't help, because the field has already been created by the time the handlers get stuck into it. You have to change `newNote` to get any effect.

Typing in Shorthand

You probably have a set of favorite stacks that you use a lot. To get to those stacks fast, you may have even set up special handlers to take you there quickly. You might also have other commands you use a lot, and for which you've written handlers, probably with short names, to make getting to those commands easier.

This section describes our system of setting up shorthand words, which are alternate names for HyperTalk statements. This is a neat way to make your HyperTalking go faster, or even to create multiple ways to refer to the same command. The shorthand system involves several handlers, plus a card in the Cookbook Reference stack. We'll tell you about all the pieces in this section.

Shorthands

This first handler in the shorthand system lets you define a new shorthand. You get to specify the shorthand you want, and the text you want it to represent. Then, when you type the shorthand into the message or command box, it'll expand into the complete form.

```

1. on shortHand shortWay, longWay
2. -- script 2-6: Shorthands: add a new shorthand.
3.
4.
5. -- This handler is part of a set of pieces that
6. -- implement a shorthand system.
7.
8. global shortHands
9. put shortWay & "," & longWay & return after shortHands
10. saveShortHands
11. end shortHand

```

Handler name:

shorthand

What it does:

This handler assigns a shorthand to a string of characters, usually a HyperTalk command. For example, if you have a stack called "Traveling Wilburys" that

you use a lot, you might set up a shorthand, like this:

```
shorthand tw, "go to stack" & quote & "Traveling Wilburys" &
quote
```

After you do this command, you'll be able to just type "tw" whenever you want to go to the Traveling Wilburys stack. The shorthand system will automatically expand the shorter name into the full, longer command. It's great!

Parameters:

The shorthand command takes two parameters. The first is the new shorthand, called "source" in the script. This is the shortcut. The second parameter, named "expansion," is the phrase created when you type the shorthand. You should make sure that you don't use any real commands as shorthands, or you'll have trouble getting access to them.

How it works:

This part of our shorthand system is the simple handler that gets installed in the command library, and whose script is listed above. Basically, all it does is maintain a global variable called "shorthands" to hold the shorthands in their short and long forms.

Line 8 declares the global variable we're going to use. In line 9, we add the newly declared shorthand and its long form (with a comma separating them) to the global. We stick it onto the end with a return after it, leaving one shorthand per line in our global variable. In line 10, we call "saveShorthands," a handler we've installed to make our shorthands live across launches of HyperCard. To see how it works, just keep on reading!

Expand a shorthand

Creating shorthands is just part of the battle. Compressing data is easy; it's the decompression that makes it interesting. In this handler, we take care of converting the shorthand text back into its full form.

1. function expandShortHand inString
2. -- script 2-7: Shorthands: translate a shorthand.
- 3.
- 4.


```
5.  -- This handler, part of the shorthand system, expands
6.  -- a shorthand when it's typed into the message box.
7.
8.  global shortHands
9.  put 0 into expandCount
10. repeat
11.   put true into plainTextFlag
12.   put empty into outString
13.   repeat with wordNum = 1 to the number of words in
       inString
14.     get word wordNum of inString
15.     if it is in shortHands then
16.       repeat with shortHandNum = 1 to the number of lines
           in shortHands
17.         if it is item 1 of line shortHandNum of shortHands
             then
18.           get item 2 of line shortHandNum of shortHands
19.           put false into plainTextFlag -- we did an
               expansion
20.           add 1 to expandCount
21.           exit repeat
22.         end if
23.       end repeat
24.     end if
25.     put it & " " after outString
26.   end repeat
27.   if plainTextFlag then return outString -- no
       expansions so all done
28.   put outString into inString -- something expanded so
       keep going
29.   if expandCount > 50 then
30.     answer "Recursive shorthand detected"
31.     return outString
32.   end if
33. end repeat
34. end expandShortHand
```


Handler name:`expandShorthand`**What it does:**

This handler converts abbreviated, easy to type shorthands into real HyperTalk commands to get something done. It's called when you type something into the message box or the command box, and before your very eyes, you see the shorthand expanded into the full message. This handler is a function that returns the expanded version of the string. Because the handler uses HyperTalk's item syntax to separate the shorthand from its full form, your shorthands can't contain commas.

One of the features of this shorthand system is that it will expand shorthands recursively—that is, it'll handle shorthands that contain other shorthands. This allows you to have multiple levels of abbreviation.

Parameters:

You'll never call this handler directly, since it's called for you when you type something into the message box or the command box. Anyway, it takes one parameter, which is the shorthand to be expanded.

How it works:

Most of the scripts in the shorthand system need access to the "shorthands" global, which holds all the shorthands and their long forms, so it's declared in line 8. Line 9 sets up a variable called `expandCount` to help us guard against shorthands that expand infinitely; for example, if "do" turns into "do do," `expandShorthands` will keep expanding every "do" into "do do," until we wind up with something like "do do do do do do do do"—too much do-do. Lines 10 through 33, or most of the script, comprise a big repeat loop that keeps looking for shorthands. A shorthand only has one word, since the whole idea here is to reduce typing, but the string you typed can be composed of any number of shorthands, and each one will be handled with care and expanded to its full form.

The idea of the shorthand translation is to keep passing through the string until there are no more shorthands to be expanded. There's a variable called `plainTextFlag`, initialized in line 11, used to keep track of whether we're done

expanding shorthands. Line 12 prepares a variable that holds the expanded string we're building up.

Line 13 starts another repeat loop that continues through line 26. This is the heart of the script. It steps through the string to be expanded, word by word, checking to see if each word is a shorthand, expanding it if so, leaving it alone if not. In line 14, we get ready to test a word from the input string, and the adventure begins.

Lines 15 through 24 compose an "if" statement that checks to make sure what you've typed is really a shorthand. It makes this check by using HyperTalk's elegant "is in" operator that quickly determines whether the tested word is in "shorthands." If not, the "if" statement fails and exits through line 24, moving on to other things that we'll talk about shortly.

If the string really is a valid shorthand, the neat part of the script executes. Line 16 starts a repeat loop to find the shorthand and figure out what it's supposed to mean. The loop is set up to cycle once for each line in "shorthands," the global variable holding all the shorthands we've created. This will make sure we look through all the shorthands we've got, if we have to.

In the very guts of the repeat loop, line 17, we check to see if we've found the shorthand we're looking for. If so, line 18 gets the string we want to expand so we can mess with it later. Since we've found a shorthand, we know we're going to have to go through the shorthand-looking process again, so we set our flag in line 19. Line 19 is also a good place to put a checkpoint so that you can watch the action with the HyperTalk debugger. We're also keeping a count of how many times we expand, to help prevent out-of-control endless loop expansion, and we increment our counter in line 20. Once we find the shorthand, there's no need to keep going through this repeat loop, so line 21 takes a quick exit and avoids checking the rest of the shorthands.

After we're done with the loop that finds the shorthand and replaces it with the expanded form, we get to climb out of the "if" and "repeat" hierarchy we've gotten into. This takes care of lines 22 through 24. At this point, the variable "it" holds a word that's either been passed through untouched, or was a shorthand and has been converted into one or more new words.

No matter where "it" came from, line 25 sticks it onto the end of outString, which is the output string we're building up, and line 26 ends the repeat loop

that scans completely through the string looking for shorthands. By the time the script gets to this point, we know the entire input string has been scanned for shorthands, and if any were found, they've been expanded. What next?

Just in case

Be aware that there's an interesting double-check going on here. The first if statement (line 15) only checks to see if the word is somewhere, anywhere in "shorthands." It's possible that the word we're checking for is actually in an expanded name, rather than in a shorthand, or maybe it's just part of a shorthand. Consider this: let's say you've created the shorthand "SO" as a shortcut for "Sinead O'Connor." Then, later, you type "sin" into the message box. When the handler checks to see if "word wordNum of inString is in shorthands," it will be true, since "sin" is in "Sinead O'Connor," even though it's not a shorthand. However, nothing bad will happen, since the loop that starts in line 16 checks specifically for the shorthand to be in item 1 of a line. If it's not there, which is what will happen to "sin," it simply gets passed through untouched, and everything is hunky dory.

It's not yet time to pack it in. Remember, we allow recursive shorthands, as discussed earlier. In line 27, we check to see if we found any shorthands on our pass through the input string. If we didn't, then we're all finished, and line 27 will end the handler by returning the value in outString as the function result. If plainTextFlag is false, the last pass found some shorthands, so we might still have some shorthands in the newly-expanded string. This means we have to scan through it all again. Line 28 gets ready for that by putting our ever-growing output string into inString, which will be processed for more shorthands.

One more thing to do. In line 29, we check to see if recursive shorthands have gone loony. In this case, loony is defined as being 50 levels of expansion; you may want to adjust this number to suit yourself. Anyway, if this hyperactive recursion is discovered, the handler puts up an alert (line 30) and stops fooling around immediately (line 31). Otherwise, the handler ends in a much kinder and gentler fashion, closing out the all-encompassing if and repeat statements started way back when.

Now we've got more of the shorthand puzzle put together. We've built a handler that will add a new shorthand, and one that will expand a shorthand into its full glorious form. Next we're going to put in stuff to make it actually work when you type into the message box.

Evaluate message box for shorthands

The dinky handlers in this section provide the crucial glue for the shorthand system: they call on `expandShorthands` to translate the shorthands after the user types return or enter into the message box.

```
1. on expandMessageBox
2. -- script 2-8: Evaluate the message box for
3. -- shorthands.
4.
5. -- This handler makes sure that anything you type
6. -- into the message box gets evaluated for
7. -- shorthands.
8. put expandShortHand(the message box)
9. end expandMessageBox
```

Handler name:

`expandMessageBox`

What it does:

This handler takes control after you type something into the message box, then press return or enter to execute it. It makes sure any shorthands you've typed gets properly expanded before the whole message box is shipped off to do its thing.

If you install the shorthand system, this handler will be called by both `enterKey` and `returnKey` handlers in the Command Reference stack's script.

Parameters:

None.

How it works:

This script does just one thing. It calls `expandShorthand` to process the text in the

message box and replace it with the decoded version by expanding any shorthands that might be there. This statement uses a powerful HyperTalk shortcut. If you use the put statement without specifying where you want to “put,” HyperTalk will assume that you want to put something into the message box, which turns out to be the right assumption in this handler. This is one example of how HyperCard is always trying to do little things to make your life easier. Isn’t that nice?

Making shorthands permanent

Shorthands are a lot more useful if you can count on the same ones being around every time you start HyperCard. You should be able to build a set of shorthands that you can always use when you’re working. This section describes a few scripts and techniques for making sure your shorthands don’t bite the dust every time you quit HyperCard.

In HyperCard, the basic technique for keeping a value even when you quit and restart is to put the value into a field. Lots of stacks like to use hidden fields to do this so the information doesn’t get in the way of the user.

Sometimes, a hidden field isn’t the right thing to do. It may be more appropriate not only to make the field visible, but to draw attention to it and allow your users to edit it directly. That’s what we’re going to do here with shorthands. The Command Reference card that installs the shorthand system has a field that holds our list of shorthands.

We’ve already discussed the shorthand script that creates new shorthands. By putting the shorthands out in the open like this, we have another, more direct way for users to make new shorthands, and an easy way to change those already defined. Of course, to make this work, we have to be sure the shorthands card and the global variable named shorthands are talking to each other, which is the purpose of the scripts in this section.

1. on initShortHands
2. -- script 2-9: Shorthands: make shorthands permanent.
- 3.
- 4.
5. -- Here's more of the shorthand system. These work
6. -- together to make sure that shorthands live across
7. -- launches of HyperCard.
8. global shortHands


```

9.  lock screen
10. push card
11. go to card "ShortHands" of stack "Cookbook Reference"
12. if the result is empty -- card exists
13. then put card field "ShortHands" into shortHands
14. pop card
15. unlock screen
16. end initShortHands
17.
18. on saveShortHands
19. global shortHands
20. lock screen
21. set lockMessages to true
22. push card
23. go to card "ShortHands" of stack "Cookbook Reference"
24. if the result is empty -- card exists
25. then put shortHands into card field "ShortHands"
26. pop card
27. unlock screen
28. end saveShortHands

```

Handler name:

initShorthands

What it does:

This handler initializes the global variable "shorthands" from the shorthands card. InitShorthands should be called when you start HyperCard. This will be set up to happen automatically after you've installed the command library.

Parameters:

None. You don't call this handler directly, since it's called when HyperCard starts up.

How it works:

InitShorthands is responsible for creating the global variable shorthands, declared in line 8. Next, this handler will go and get any shorthands that have been saved. The easiest way to do that would be with a statement like

```
put card field "Shorthands" of card "Shorthands" into
shorthands
```

But what if there's some problem and there's no "Shorthands" card? The script would stop with the ignoble "No such card" error alert. Our initShorthands handler uses a little trick to perform some error handling. We can use the go statement in line 11 as a preflight check. If that command fails, there's no alert box—HyperTalk simply continues along and puts "No such card" into the "result" function.

So, in line 12, we can check to see if the "go" statement was successful. If it was, the "result" function will be empty, and we can then assume the card is there. Then line 13 loads the shorthands variable from the card field. If the "go" statement didn't go, HyperTalk's result function will hold some sad error message, and the shorthands variable will remain empty, but the script won't stop with an error message, since we handled the error internally.

Now let's go back and pick up a couple of lines that we skipped. Since we're trying to go to another card in line 11, we locked the screen in line 9 to spare the user from having to watch the cards flash by. It gets unlocked in line 15. Also, in line 10 we push the current card, making it easier to return there in line 14 with a pop card statement. That's it for setting up shorthands when HyperCard starts. Next, we'll talk about how to save them when they change.

Handler name:

```
saveShorthands
```

What it does:

Even though there's a card with a list of shorthands, the user can add new shorthands while HyperCard is running by calling the shorthand handler (described earlier). When this happens, saveShorthands makes sure that the shorthands card has the latest information about shorthands.

Script tips: Locked up tight

In most of our handlers that lock the screen, we don't bother to unlock it before the handler ends, because HyperCard automatically unlocks the screen for us at idle time, after it finishes with any pending handlers. You might have noticed that `initShorthands` is an exception. It explicitly calls `unlock screen` in line 15. How come?

Well, HyperCard's screen-locking and unlocking isn't really a toggle; it's a counter. In other words, if you call `lock screen` three times, you have to call `unlock screen` three times, too, or it won't unlock (until it reaches idle time). So, our rule for calling `unlock screen` explicitly in handlers is this: if we expect our handler to be called by another handler that might also lock the screen, we should explicitly unlock the screen before leaving. `InitShorthands` is called (indirectly) by `getHomeInfo`, which just happens to lock the screen (these coincidences are truly amazing, aren't they?). If your handler is not likely to be called by another, you can probably get away with not unlocking the screen.

Parameters:

None. You don't call this handler directly. It's called by the shorthand handler after a new shorthand is created (see script 2-6).

How it works:

`SaveShorthands` is basically the inverse of `initShorthands`. The `initShorthands` handler runs when HyperCard starts, and it grabs the shorthand information from the shorthand card and plugs it into the `shorthands` variable, where it will be easily accessible to the scripts that need it. `SaveShorthands` runs when a new abbreviation is created by the shorthand handler. It takes the information in the `shorthands` variable (which, of course, will be lost when HyperCard goes

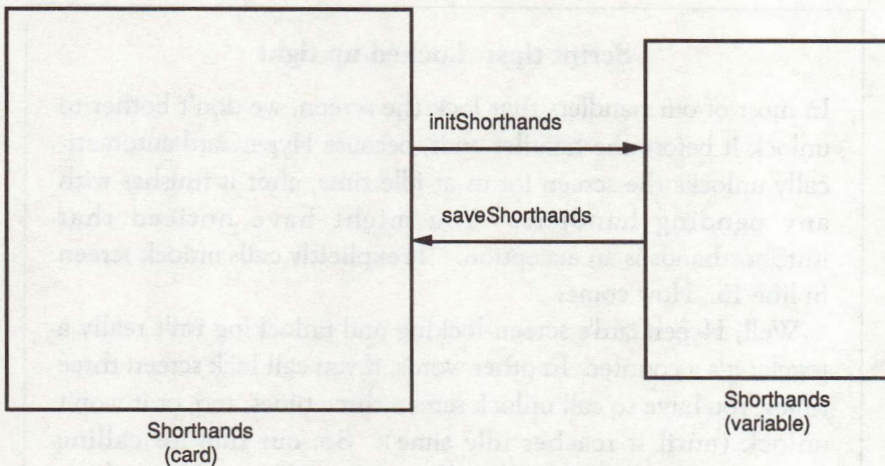


Figure 2-4 Shorthand information flow

away), and puts it into the shorthands card, which will keep this precious list until the next time we summon HyperCard. This two-way flow of information, pictured in Figure 2-4 ensures that the shorthand information is consistent and up-to-date.

If you compare `initShorthands` and `saveShorthands` carefully, or even carelessly, you'll see that they're identical except for two things. First, `saveShorthands` adds a call to lock HyperTalk messages, preventing any stray messages. Second, line 13 in `initShorthands` puts the information *from* the field *into* the variable; the corresponding line in `saveShorthands`, line 25, puts the information *from* the variable *into* the field. These two differences tell the story of their respective functions.

We've defined the handler called to make sure the shorthands field is kept up-to-date. Now we need a `closeField` handler to complete the system.

```

1. on closeField
2.  -- script 2-10: Shorthands field script.
3.
4.
5.  -- This handler saves shorthands field changes into
6.  -- the corresponding global.
7.

```



```

8.  global shortHands
9.  if last char of me is not return then put return
    after me
10. put me into shortHands
11. end closeField

```

Handler name:

closeField

What it does:

This closeField handler lives in the script of the “shorthands” field on the “short-hands” card. It gets called, like any other closeField handler, whenever the text in a field is changed and the selection is removed from the field. According to that coffee table favorite, *HyperTalk 2.0: The Book*, by Dan Winkler and Scot Kamins, “actions that remove the selection include (but are not limited to) opening another field, pressing the enter key, tabbing, clicking outside the field, toggling the message box, switching into the background, and changing cards.” The job of this closeField handler is to make sure the shorthands variable is updated whenever the user changes the shorthands field directly.

Parameters:

None. Of course, this is another handler you don’t call directly. It gets called when HyperTalk generates the system message closeField.

How it works:

As you can see (lines 8 through 10 above), there’s not much to this one. It simply declares the shorthands global, makes sure that the field’s contents end with a return, then puts its own value into the global variable. This ensures that the variable reflects any changes the user makes directly to the field.

We’ll do one more trick with shorthands. If we tie the shorthand conversion to the shift-tab handler that takes us to the command library, we can use shorthands in the command box. This is a pretty simple addition to the shift-tab script, and it’s explained in Appendix A along with the rest of the shift-tab stuff.

That takes care of the entire shorthand system. Remember, treat your shorthands right, and they’ll give you years of faithful service.

Working with Handlers in Scripts

In HyperTalk, a handler is a set of statements that executes in response to a message. A handler always begins with "on" or "function," and it ends with "end," which sounds very reasonable. A single object's script can contain lots and lots of handlers. It's not unusual for a single object's script, especially a stack script, to contain dozens of separate handlers. This section presents some handlers used for dealing with individual handlers in a script.

Extract a handler from a script

This handler knows how to pull out a single handler from a script, just like Bullwinkle shows Rocky how to pull a rabbit out of his hat. Unlike that trick, though, this one works all the time.

```

1. function getHandler handlerName,scriptText,
   returnOffsets
2.  -- script 2-11: Extract a handler from a script.
3.
4.
5.  -- Given the name of a handler and a script, this
6.  -- handler gets the text of the specified handler.
7.  -- Optionally, it will return starting and ending
8.  -- character expressions of the handler. Check for
9.  -- function first because on is a substring of function.
10. get offset("function " & handlerName,scriptText)
11. if it is zero then get offset("on " & handlerName,
   scriptText)
12. put it into startChar
13. put return & "end " & handlerName into footer
14. put offset/footer,scriptText into endChar
15. if endChar is not zero then
16.   add length/footer) to endChar
17.   repeat until char endChar of scriptText is return -
18.   or char endChar of scriptText is empty -- skip
   comment
19.   add 1 to endChar
20. end repeat

```



```

21.   repeat while char endChar + 1 of scriptText is return
22.     add 1 to endChar -- take trailing returns
23.   end repeat
24. end if
25. if returnOffsets is true
26.   then return startChar & "," & endChar
27. else return char startChar to endChar of scriptText
28. end getHandler

```

Handler name:

getHandler

What it does:

Given a script and the name of a handler, this function returns the text of that handler's script. Optionally, you can ask for just the starting and ending character positions of the handler in the script instead of the complete text of the handler.

Parameters:

There are three parameters. The first one, cleverly named `handlerName`, is the name of the handler you're looking for. This name can refer to either a standard message handler or a function handler, and it shouldn't include the "on" or "function." This script will tack those on. The second parameter, `scriptText`, is the text of the script to search through. Usually, this will be an expression that leads to a script, such as "script of button ID 5," but if you happen to have a script stored in a field or a variable, you can use it there.

The third parameter is optional. `GetHandler` normally returns the entire handler you request. You should pass "true" in the third parameter, named `returnOffsets`, if you'd rather receive just the starting and ending character positions and not the entire handler.

How it works:

Since `getHandler` knows that every handler starts with either "on" or "function," it adds those words automatically. In line 10, it looks for the word "function" plus the handler's name, as passed in the `handlerName` parameter in the `scriptText` parameter. This line works by using the built-in function "offset,"

which will return zero if the function isn't found. If that happens, line 11 looks for the handler again, this time with "on" in front of it.

After we've found the start of the handler and stashed it away into the `startChar` variable (line 12), it's time to find the end of the handler. First, we make a variable called `footer` that holds the word "end" and the name of the handler. In line 14, we find the end of the handler, then add the length of the footer, putting the position of the last character in the handler's "end" line into the `endChar` variable.

The naming of cats

Even though message handlers are more common than function handlers, `getHandler` very carefully makes sure to search for a function first. As you may have guessed, this is because "on" is the last part of "function," and any search for "on mungoJerrie" would also find "function mungoJerrie," which would confuse `getHandler`'s idea of where the handler begins.

Lines 17 through 24 are used to grab any extra carriage returns or comment lines in the script following the end statement. This works by peeking ahead at the next character in the script to see if it's a return. If it is, we bump the value of `endChar` by one, and keep looking until we don't see any more returns.

At this point (after line 24), we have the desired handler's starting and ending locations stored in `startChar` and `endChar`. The only thing left to do is to return the function's value. In line 25 we test to see if the caller asked for offsets to be returned. If so, we simply return those values, with a comma in between (line 26). If the caller wanted the whole text of the handler, line 27 gets it out of `scriptText` and returns it, and the handler ends. We'll use this automatic handler extraction power in our next script.

Place a handler into a script

This script lets you insert a handler into a script without disturbing any other handlers in that script. Before inserting the new script, it checks to see if the target script already has a handler with the same name as the one you're inserting. If it does, that old version is replaced with the shiny new one.


```

1. function putHandler newHandler,scriptText
2. -- script 2-12: Place a handler into a script.
3.
4.
5. -- Given the name of a handler and a script, this
6. -- handler puts the text of the specified handler
7. -- into the specified script, replacing the previous
8. -- version of the handler in that script.
9.   put word 2 of newHandler into handlerName
10.  put getHandler(handlerName,scriptText,true) into
    offsets
11.  put item 1 of offsets into startChar
12.  put item 2 of offsets into endChar
13.  put newHandler & return into char startChar to
    endChar of scriptText
14.  return scriptText
15. end putHandler

```

Handler name:

putHandler

What it does:

PutHandler is a function that inserts a handler into a script. The new handler and its destination are passed in as parameters. If the destination script already has a handler with the same name, it's ruthlessly deleted in favor of the new handler.

Parameters:

There are two parameters to putHandler. The first, newHandler, contains the text of the handler to be inserted into the script. The second parameter, named scriptText, holds the text of the script to have the new handler stuck into it. The function returns the entire script with the new handler in it.

How it works:

Even though putHandler seems like the inverse of getHandler, it's really not. GetHandler finds a handler within a script and returns it, or reports its location.

PutHandler inserts a handler into a script. If there's already a handler with that name, putHandler must replace it. Ironically, this means that putHandler has to call getHandler to find the handler that's already there.

In line 9, we put the second word of the new handler into the handlerName variable. This will always be the name of the handler, since every handler starts with "on someThing" or "function someThing." In line 10, we call getHandler to get the character positions of a handler with the same name. If there is no handler with that name, the function returns "0,0."

Lines 11 and 12 put the offsets into startChar and endChar. In line 13, HyperTalk elegantly replaces the old handler with the new one in one fell swoop, as they say (fell, *adj.*, Cruel, fierce, unsparing). This is accomplished by specifying both the characters to put, which are "newHandler & return," as well as the positions to put them into, which are "char startChar to endChar of scriptText."

Remember that putHandler is a function, and it's supposed to return the text of the script with the new handler installed. Line 14 takes care of this obligation, and the handler finishes up in line 15.

More helpful help

The HyperCard help system is a good way to get answers to lots of questions arising while you're building your stacks. It includes a basic description of HyperTalk, including the names of built-in commands and functions and their syntax. This simple handler lets you get to the help topic a little faster, especially if you're interested in learning more about a HyperTalk topic.

```

1. on help topic
2. -- script 2-13: More helpful help (help with a topic).
3.
4.
5. -- This handler lets you specify a topic when you
6. -- ask for help with the standard Help stacks.
7.
8.   if topic is empty then
9.     ask "What topic would you like help on?" with
       "Anything"
10.    put it into topic

```



```
11. end if
12.
13. go to card 1 of bg "Content" of stack "HyperTalk
    Reference"
14. if topic is "Anything" then pass help
15. find topic in field "Title"
16. if the result is not empty then — not found
17.   lock screen
18.   go to card 1 of bg "Content" of stack "HyperCard Help"
19.   find topic in field "Title"
20.   if the result is not empty — not found, go back
21.   then go back
22. end if
23. end help
```

Handler name:

Help

What it does:

There are several different ways you can ask HyperCard for help: you can type command-?, you can choose “Help” from the “Go” menu, or you can send the “help” message. Actually, all of these activities wind up sending a “help” message that’s usually handled by HyperCard. This means you can intercept all cries for help by defining your own help handler, which is what we’ve done here. When our handler gets control, it simply looks for the desired thing after going to the help stacks.

Parameters:

This handler takes one parameter, and that’s where all the fun comes in. The parameter is the topic you’re interested in finding out about, and the handler simply attempts to find it in the help stack. It searches for the desired topic in background field “Title,” which contains the title of the card in HyperCard 2.0’s help stacks.

How it works:

Like lots of our other scripts, we want to allow enough flexibility to accept a parameter if one is sent, or ask the user if a topic wasn't sent. Line 8 checks to see if there's already a topic. If not, lines 9 and 10 ask the user for one and put the answer into "topic." If the user doesn't fill in a topic, it keeps the default "Anything," as shown in line 9.

After making sure there's a topic, line 13 takes us to the main part of the HyperTalk reference stack. We're assuming here that most "help" questions will be about HyperTalk vocabulary, rather than general HyperCard stuff. Line 14 checks to see if there's a topic to look for. If not, it passes the help message on, probably to be handled by HyperCard itself. Otherwise, the work continues in line 15.

We actually look for the topic in line 15, using the HyperTalk Reference stack's Title field. If we find something, the right card will be displayed, and if not, we'll try again in the other help stack. Line 17 locks the screen before going to the new stack. Line 18 gets us ready to search in the main part of the HyperCard Help stack, and line 19 does the looking. Once again, if it finds anything, the right card will be displayed, and if not, we give up in line 21 and return from whence we came.

Script tips: What is getHomeInfo, anyway?

If you're like most good stackheads and you've done your time rooting around through all the stuff you get with HyperCard, you've probably discovered a long and curious handler named `getHomeInfo` in the Home stack. Because `GetHomeInfo` isn't documented anywhere, and because it seems to be so important and so mystical, HyperCard folks are often curious about it. Here's the story about `getHomeInfo`, as described by Dan.

`GetHomeInfo` is called on `startUp` and `resume` to restore several global variables and properties. `GetHomeInfo` might be better named `restoreUserPreferences`, but it was very late at night and nobody wanted to type that much. The values of these variables and properties are stored in buttons and fields on several cards in the home stack. The cards are named "User Level," "stacks," "applications," and "documents."

`GetHomeInfo` sets four global properties of HyperCard, according to buttons on the "User Level" card. They are:

- `userLevel`, which comes from card field "User Level," based on the five User Level buttons
- `powerKeys`, from button "Power Keys"
- `blindTyping`, from button "Blind Typing" (you're probably starting to figure out how this works)
- `textArrows`, from button "Text Arrows."

The card field "User Level" on the "User Level" card is hidden. Its value is maintained by a handler named `setUserLevel` in the card script. The buttons "Power Keys," "Blind Typing," and "Text

Arrows” use their hilite property to keep track of whether these properties are true or false.

GetHomeInfo also sets four global variables, which are:

- userName, from card field “user name;”
- stacks, from field “paths” of card “stacks;”
- applications, from field “paths” of card “applications;”
- documents, from field “paths” of card “documents.”

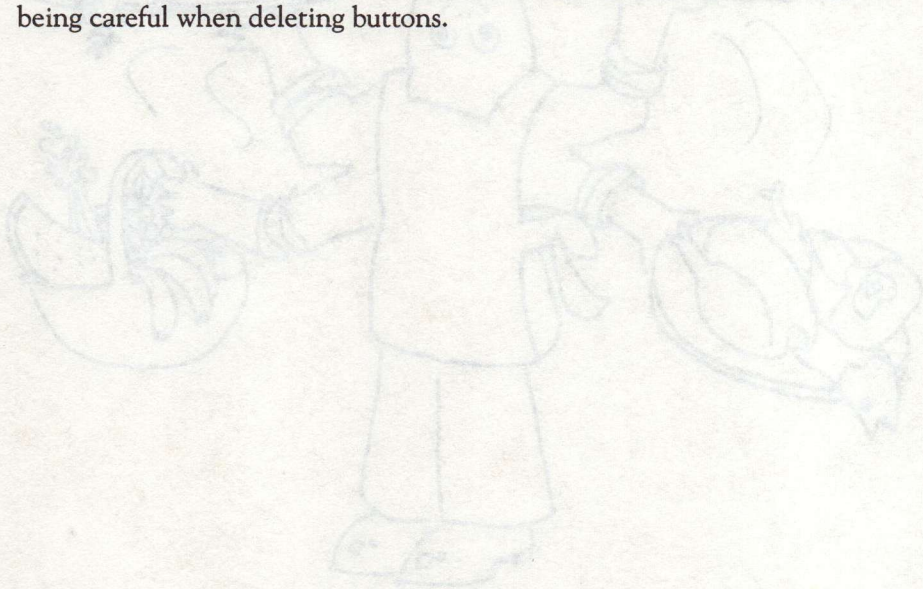
The global variable “userName” is never used by HyperCard. It’s there entirely for the convenience of scripts that may want to use it, and there’s no way to know if it’s even been set up properly by the user. The other three globals are read and changed by HyperCard. There’s no good reason why these are global variables and not global properties—that’s just the implementation. In hindsight, they probably should be global properties.

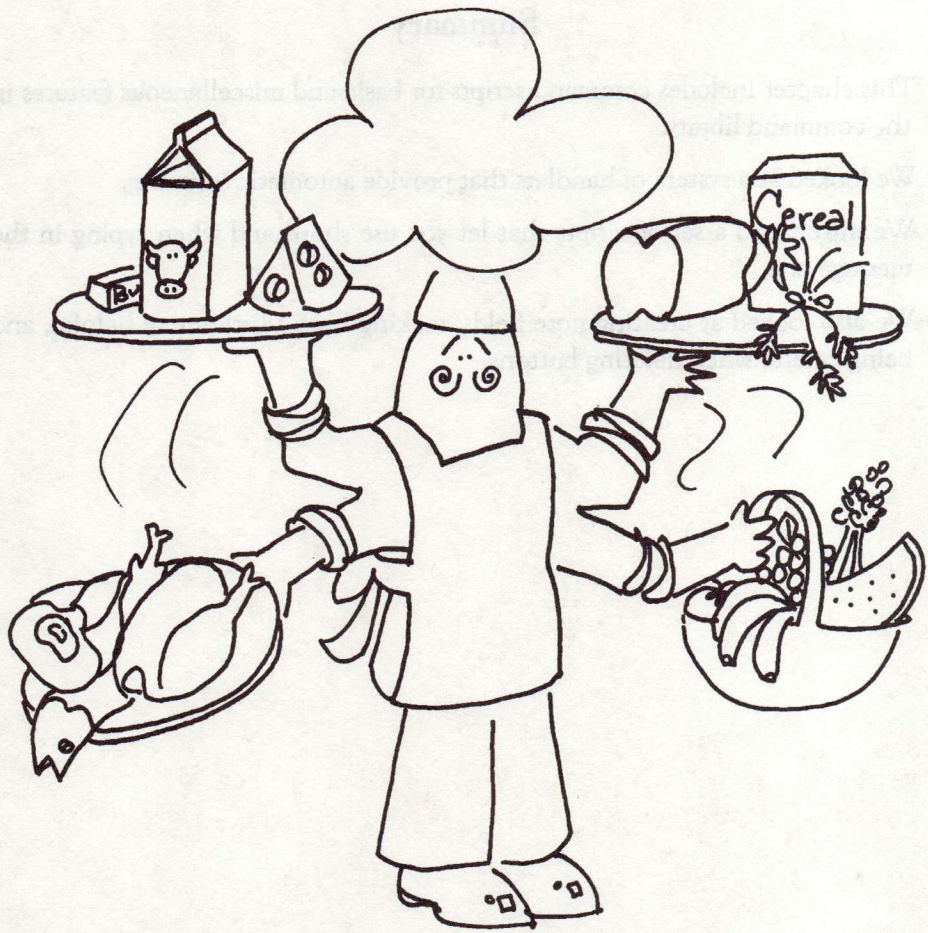
HyperCard looks at the “stacks,” “applications,” and “documents” variables when it’s looking for a stack, an application, or a document to open. It uses these variables as a list of folders (that is, search paths) that might contain the desired item. If the item can’t be found, HyperCard puts up a dialog asking the user where the item is. If the user finds it, HyperCard adds the new search path to the appropriate global variable and into the corresponding field on the card in the home stack. There’s no script that performs this searching and automatic updating mechanism. HyperCard itself has the lookup and updating hard-wired into it.

GetHomeInfo also has become the standard place to put handlers that have to be called when HyperCard gets started. If you look in the Home stack, you’ll see that getHomeInfo is called by handlers for both “startUp” and “resume.” HyperCard sends “startUp” when it’s first started, and it sends “resume” if the user has previously started an application from HyperCard without MultiFinder turned on, then quits that application, which restarts HyperCard.

Summary

- This chapter includes command scripts for basic and miscellaneous features in the command library.
- We looked at a system of handlers that provide automatic scripting.
- We introduced a set of scripts that let you use shorthand when typing in the message box.
- We also looked at creating note fields, making help a little more helpful, and being careful when deleting buttons.







Chapter 3

WORKING WITH GROUPS

Like the basic food groups, you'll often find yourself working with groups of objects in HyperCard (I know, that's really stretching the metaphor). In this chapter, we'll see handlers designed to help you work with groups of objects all at once. And they're nutritious, too!

Group Selection

The first set of scripts can be used to answer a question HyperCard users have been asking ever since there was HyperCard: how to select and manipulate groups of buttons and fields all at once in a simple and elegant manner.

The ability to tell HyperCard you want to work with several things at once has several nice benefits. First, if you want to do the same thing to a bunch of objects, like delete them, it's easier and more convenient (for you, that is, not necessarily for HyperCard) to select everything at once, then tell HyperCard to delete everything you selected. There are other neat tricks you can do with group selections of objects, as you'll see in this section. Here are the scripts.

Click on parts

In HyperCard, a *part* is a button or a field. This chapter's group selection system lets you work with any collection of parts, including any combination of card and background parts. This first script, which implements the `selectParts` handler, lets you group-select a bunch of parts by clicking on them, one at a time. When you're done selecting parts, you press the enter key. The message box provides feedback as you select things. The handler takes care of such miscreant cases as trying to select the same part twice and clicking where there's no part.

1. on selectParts
2. -- script 3-1: Group selection: click on parts
- 3.
- 4.


```

5.  -- Set card script to catch mouseDown and eat mouseUp.
6.
7.  global saveCardScript,selectedParts,selectedPartsCard
8.  put empty into selectedParts
9.  put long name of this card into selectedPartsCard
10. put script of this card into saveCardScript
11. choose browse tool
12. put "Click on the parts you wish to select. Press
    enter to stop."
13.
14. -- write a script
15. put "on mouseDown" & return into scriptVar
16. put "    global selectedParts" & return after scriptVar
17. put "    put the shiftKey is down into deselectFlag" &
    return after scriptVar
18. put "    if " & quote & "button" & quote & " is not in
    the target and " & quote & "field" & quote & " is
    not in the target" & return after scriptVar
19. put "    then put " & quote & "There is no part there."
    & quote & return after scriptVar
20. put "    else" & return after scriptVar
21. put "    get the target" & return after scriptVar
22. put "    if " & quote & "id" & quote & " is not word 3
    of it" & return after scriptVar
23. put "    then put " & quote & "id " & quote & " & id
    of the target into last word of it" & return after
    scriptVar
24. put "    if deselectFlag then" & return after
    scriptVar
25. put "    if it is not in selectedParts" & return
    after scriptVar
26. put "    then put the target & " & quote & " is not
    selected." & quote & return after scriptVar
27. put "    else" & return after scriptVar
28. put "    put offset(it & return,selectedParts)
    into startChar" & return after scriptVar

```



```
29. put "          delete char startChar to startChar +  
    length(it) of selectedParts" & return after  
    scriptVar  
30. put "          put the target & " & quote & " is now  
    deselected." & quote & return after scriptVar  
31. put "          end if" & return after scriptVar  
32. put "    else" & return after scriptVar  
33. put "  if it is in selectedParts" & return after  
    scriptVar  
34. put "          then put the target & " & quote & " is  
    already selected." & quote & return after scriptVar  
35. put "          else" & return after scriptVar  
36. put "          put it & return after selectedParts" &  
    return after scriptVar  
37. put "          put the target & " & quote & " is now  
    selected." & quote & return after scriptVar  
38. put "          end if" & return after scriptVar  
39. put "    end if" & return after scriptVar  
40. put "  end if" & return after scriptVar  
41. put "  exit to HyperCard" & return after scriptVar  
42. put "end mouseDown" & return after scriptVar  
43. put return after scriptVar  
44. put "on doneSelecting" & return after scriptVar  
45. put "  global saveCardScript,selectedParts" & return  
    after scriptVar  
46. put "  set script of this card to saveCardScript" &  
    return after scriptVar  
47. put "  put the number of lines in selectedParts & " &  
    quote & " parts selected" & quote & return after  
    scriptVar  
48. put "end doneSelecting" & return after scriptVar  
49. put return after scriptVar  
50. put "on closeCard" & return after scriptVar  
51. put "  doneSelecting" & return after scriptVar  
52. put "end closeCard" & return after scriptVar  
53. put return after scriptVar
```



```

54. put "on enterKey" & return after scriptVar
55. put " doneSelecting" & return after scriptVar
56. put "end enterKey" & return after scriptVar
57.
58. set script of this card to scriptVar
59. end selectParts

```

Handler name:

selectParts

What it does:

When you send the selectParts message to your command library, you select parts by clicking on them. To select an unlocked field, hold down the command key while clicking (known in some parts as command-click). You can also remove a part from the group selection by shift-clicking on it. Press the enter key when you're finished selecting.

Parameters:

None.

How it works:

SelectParts uses the automatic scripting trick discussed in Chapter 2. The script selectParts writes gets shoved temporarily into the current card's script. Since most folks don't take too kindly to having their card scripts destroyed by passing commands, selectParts uses the saveCardScript global to hold onto the original card script before bashing it.

This handler uses two other globals. One, selectedParts, is used to hold the list of parts that you click on, and two, selectedPartsCard (as you might guess from that wonderfully descriptive variable name—congratulations, Dan), holds the name of the card with the selected parts on it. We'll use it later in other scripts implementing the selected parts idea.

Lines 8 through 10 initialize the globals to appropriate values—selectedParts is emptied, selectedPartsCard gets the name of the card, and saveCardScript gets a copy of the card script. Line 11 conveniently makes sure the browse tool is active, so you can use selectParts even if you're working with one of the other tools. Line 12 just gives you some feedback in the message box to help you

remember you're in part-selection "mode."

The rest of the handler is automatic scripting magic. Lines 15 through 56 create four handlers that will be inserted into the current card's script in order to implement group selecting. If you've been paying attention, you might guess these lines were created automatically with the `writeScript` handler, and you'd be right. Let's use lines 15 through 56 to see what the manufactured scripts will look like.

The first handler written is for `mouseDown`. This is a tricky way of figuring out when a part is clicked on. Most buttons have handlers for `mouseUp`, but handling a `mouseDown` is pretty rare. So, our group selection system relies on getting `mouseDowns` in the card script to know what's going on. If a part happens to catch a `mouseDown`, you won't be able to add it to the group selection (unless it also happens to include a "pass `mouseDown`" statement).

First, line 17 checks to see if the user was holding down the shift key when the `mouseDown` happened. If so, it means the user wants to remove a part from the selection, not add one, so we set a variable called `deselectFlag`. Note that "the `shiftKey` is down" is a logical expression: it returns either true or false, and the result is simply stuck into `deselectFlag`.

In lines 18 and 19, the script checks to see if the click happened on a part, or just out in space somewhere on the card. The script uses the function "the target," to identify the object that was the initial recipient of the message the current handler is working on. In this case, "the target" will tell us which field or button was clicked on, if any. If the user did manage to hit a field or button (remember, a part is a field or button), the else path (line 20) is taken, which lasts for the rest of the `mouseDown` handler.

At this point in the script, we know that the user has clicked on a part, and we can get the part's name by calling HyperTalk's "the target" function. We're going to make a list of the parts that the user clicks on, and we'd like them all to be in the same format. The answer we get from the target can vary a little. If the part in question has a name, the target will return something like this:

```
card button "Harry Callahan"
```

This is the format you'll get for parts with names. If the part has no name, the target will return information like this:

```
card button id 2
```


Rather than keep both kinds of references in our list of group-selected parts, we'd like to put them all into the same format. Since all parts, whether they have names or not, can be referred to in the second form, that's what we'll use. So, line 21 of the script gets the target. Line 22 cleverly checks to see if the target returned a result for a named part or an unnamed one; if the target did have a name, the if statement in line 22 will be true, and line 23 will convert it into the "ID" form. If the part had no name, word 3 of the target will already be "ID," the format we want.

Now we've got the reference to the selected part just the way we want it. The next thing we have to do is see if the user is interested in adding it to, or removing it from, the selection. This check happens in line 24, where the `deselectFlag` is examined. If the user wants to deselect this part, the shift key is down, which sets `deselectFlag` to true (line 17), and "if `deselectFlag`" is true. This causes line 25 to be executed, a test to see if the part is already part of the selection. If not, it doesn't make any sense to deselect it, and line 26 tells the user this.

Script tips: Time waits for no one

In this script, we check the status of the shift key in line 17 by calling HyperTalk's `shiftKey` function and sticking its answer into a variable, `deselectFlag`. We use `deselectFlag` later when we want to know whether the user pressed the shift key. Wouldn't it be simpler to call the `shiftKey` function in line 24, and just punt `deselectFlag`? Yes, but it wouldn't be exactly what we want. We want to know whether the shift key was down when the part was clicked on. By the time line 24 is executed, a significant amount of time may have already passed since `mouseUp`—maybe just a tenth of a second, but long enough for a quick-fingered user to release the shift key. It's always a good idea to test the modifier keys (shift, option, command) as close as possible to where they'll be used, or unexpected things can happen to fast-typing users.

If the script falls through to the else in line 27, it means the part shift-clicked on is already part of the group selection, and we have to remove it. In line 28, we find out exactly where in the group selection the unwanted part is located. Remember that “it” contains the name of the part that was clicked on (it got there in line 21), so “offset (it & return,selectedParts),” which winds up in startChar in line 28, tells where to find the part that’s going to be deselected. Then, in line 29, the name of the part is chopped from the list of selectedParts. The “to startChar + length(it)” part of the delete command makes sure the entire line, including the return at the end of the line, is removed. Line 30 is used to report the news.

After line 31, we’ve concluded the “else” part of the “if deselectFlag” statement. Now we have to implement the part of the handler that adds buttons and fields to the group selection, which will happen in the rest of this script.

In line 33, we check to see if the part is already in the list of parts we’ve selected. If so, line 34 gets executed and announces to the user that the part is already in the group selection. Finally, if it hasn’t already been selected, line 36 adds it to the selection by putting its name into a new line of the global selectedParts. This global is used to compose a return-separated list of the selected parts. Line 37 provides feedback by putting the target (the part we just added to the selection) into the message box.

All that’s left in lines 38 through 42 is some housekeeping. Line 38 ends the “if it is in selectedParts” statement, line 39 ends the “if deselectFlag” statement, and line 40 closes the giant if statement which began in line 18. Line 41 is a little magic. Remember that all this stuff is happening as a result of a mouseDown message. By issuing this “exit to HyperCard” command, HyperCard gets control while the mouse is still down. When the user lets go of the mouse button, HyperCard normally sends a mouseUp message. But the “exit to HyperCard” causes HyperCard to forget it’s looking at a mouse click, so the mouseUp evaporates and is never seen. This implementation detail handily avoids making the parts think they’ve been clicked.

When you’re in selectParts, remember this mouseDown handler gets installed in the card script, along with three little handlers we’ll describe next. The first one, doneSelecting, is manufactured by lines 44 through 48, and is called when the user is all through selecting parts. It does just two things. Initially, in line 46, it kindly restores the script of the current card to what it was before we came tromping through (remember we saved it back at the beginning of selectParts, in

line 10). Then, line 47 makes the message box say how many parts have been selected (this works because there's one line in `selectedParts` for every selected part).

There are two ways to stop selecting, and their handlers are defined next. They're implemented by calling `doneSelecting`. One way to stop is by going to another card, so a `closeCard` handler created by lines 50 through 52 calls `doneSelecting`. The other way is to press the enter key, defined by the `enterKey` handler built by lines 54 through 56. If you want to implement other ways to end the selection process, just define handlers for them to call `doneSelecting`, convert them into puttable form using `writeScript`, and add them to `selectParts`.

Insanely great

The idea of “borrowing” an object's script to overcome the way it handles messages is a powerful, devious technique that can have wide applications. This is the kind of thing HyperCard allows, with its ability for scripts to get scripts, write scripts, write scripts that write scripts, and so on. We hope this technique and others like it will encourage you to come up with your own strange and wonderful new ideas.

Group selection by specifying a rectangle

Another convenient way to select a group of parts is by drawing a rectangle that encloses all the parts you want to mess with. Obviously, this can be quicker and more convenient than having to click on every part individually. This script defines a handler called `rectSelectParts` to let you select multiple parts.

```

1. on rectSelectParts
2.  -- script 3-2: Group selection by specifying a
3.  -- rectangle.
4.
5.  -- Select a group of parts by clicking at topLeft and
6.  -- bottomRight of enclosing rectangle.
7.
8.  global selectedParts,selectedPartsCard

```



```
9.  put empty into selectedParts
10. put long name of this card into selectedPartsCard
11. put "Click at top left..."
12. wait until the mouseClicked
13. put the clickLoc into topLeft
14. put "Click at bottom right..."
15. wait until the mouseClicked
16. put the clickLoc into botRight
17. put topLeft & "," & botRight into selectionRect
18. repeat with i=1 to the number of buttons
19.   if loc of button i is within selectionRect
20.   then put " & id of button i & return after
       selected Parts
21. end repeat
22. repeat with i = 1 to the number of bkgnd buttons
23.   if loc of bkgnd button i is within selectionRect
24.   then put "bkgnd button id " & id of bkgnd button i &
       return after selectedParts
25. end repeat
26. repeat with i = 1 to the number of fields
27.   if loc of field i is within selectionRect
28.   then put "bkgnd field id " & id of field i & return
       after selectedParts
29. end repeat
30. repeat with i = 1 to the number of card fields
31.   if loc of card field i is within selectionRect
32.   then put "card field id " & id of card field i &
       return after selectedParts
33. end repeat
34. put the number of lines in selectedParts & " parts
    selected."
35. end rectSelectParts
```


Handler name:`rectSelectParts`**What it does:**

The `rectSelectParts` handler defines an alternative way to select multiple parts. If the goal of `selectParts` was labor-saving, then `rectSelectParts` is downright lazy-making. To make it work, you type `rectSelectParts` into the command box. The message box then asks you to click first at the upper left, then the lower right corners of a rectangle surrounding the things you want to select. After you've obliged, the message box lets you know how many parts you've captured.

Parameters:

None.

How it works:

You might expect that the structure of this script would be very similar to `selectParts`, and of course, you'd be absolutely wrong. The environment in which you select things with this script is much more tightly controlled, so the script can be much simpler. With `selectParts`, we had no idea how many parts were going to be selected, so we had to take steps to control the select "mode." This time, we know that there are going to be exactly two clicks, which makes things a lot easier.

The most important simplification in this script is that we don't have to use the automatic scriptwriting technique, since the selecting environment is so much more controlled. Another difference is that we don't have to click on parts this time, just around them, so we won't have to use "the target" to figure out what to select.

The basic strategy for our script is quite simple. First, we'll ask the user to define a rectangle by clicking at its corners. Then, we'll examine all the buttons and fields, both in the current card and the current background, to see which of them fall within the rectangle. Just as in `selectedParts`, we'll use a couple of globals, `selectedPartsCard` to hold the name of the card, and `selectedParts` to keep the list of the selected parts themselves. Now let's look at the script.

Lines 9 and 10 initialize our globals, just as in the previous script. Line 11 tells the user to click at the upper left corner of the rectangle, and line 12, in

plain English and perfect HyperTalk, waits for that to happen. Line 13 stashes the location of the click in a variable called `topLeft`. Lines 14 through 16 repeat the process, this time for the lower right corner of the rectangle, which is kept in a variable named `botRight`.

After both corners have been specified, line 17 puts them together, with a comma between them, into the `selectionRect` variable. This sets up `selectionRect` to work with HyperTalk's "is within" operator later in the script.

Most of the rest of the script consists of four repeat loops that check to see which parts, if any, are within the specified rectangle. There's one loop each for card buttons, background buttons, card fields, and background fields. Lines 18 through 21 are the loop for card buttons. Line 18 sets up the loop to execute once for each card button (by not specifying "card" or "field" in front of "button," HyperTalk always assumes we mean card buttons, since that's the more common case).

Best guess

When HyperTalk was being created, Dan assumed that most buttons would be card buttons, and most fields would be background fields, which is why those became the defaults. And what really happened? Well, that depends. In stacks that present hypertext-style information, there are lots of card buttons, often used to activate links to other cards. In database-style stacks, however, there are usually a lot of background buttons, since there's a common function across many cards. Of course, for compatibility, the defaults can never ever change. What if Dan had it to do all over again? Maybe both cards and fields would be "background" by default. Truly, hindsight is 20-20.

Lines 19 and 20 do the work in the loop. They use HyperTalk's "is within" operator to see if a button falls within the rectangle the user specified earlier. If so, an expression that contains the names of the card button is tacked on to the end of the `selectedParts` variable. This is the only processing that happens for each card button in the loop.

The process of checking to see if the "loc of button i is within `selectionRect`" takes place for background buttons in the next loop. Other than checking for background buttons instead of card buttons, this loop is identical to the first

one. After all the buttons are checked, there are two more loops to check for card and background fields potentially within the selection rectangle. Again, these loops are identical except for the kind of part they're looking for. Also, remember that a field with no "card" or "background" specifier is considered to be a background field, while a button that doesn't say "card" or "background" is always a card button. Why? Because most buttons are card buttons, and most fields are background fields (as we just discussed in the "Best Guess" note).

The last few lines of the handler clean things up. Line 34 simply reports the total number of selected parts (there's one line in `selectedParts` for every part captured in the selection rectangle), and line 35 ends the handler.

Touch-a-touch-a-touch-a-touch me

It's very important to note that this script uses the button and field property "location" (which can be shortened to "loc," as it is in this handler) to figure out if the parts are within the selection rectangle. Location is a point representing the center of a field or button. This means that our script actually tests to see if the *centers* of all the fields and buttons fall within the selection rectangle (see Figure 3-1).

This is slightly different from the way rectangular group selection works in most other programs. For example, in the Finder, when you draw a rectangle to group-select icons, any icon touched by the selection rectangle is selected; in MacDraw, only objects completely enclosed by the selecting rectangle wind up selected. You can modify `rectSelectParts` to use either of these schemes by invoking HyperTalk's `topLeft` and `botRight` parameters to be more picky about whether to include a selected part. Small brain exercise: what happens if you just change "loc" to "topLeft?" Try it and see if you can figure out what's going on.

Group selection of cards

HyperCard 2.0 introduced the notion of marked cards, which is sort of a group selection of cards. Marking cards allows you to select a group of cards in a single

stack. In this section, we'll create a group selection handler that goes beyond marked cards by letting you select cards from different stacks, too.

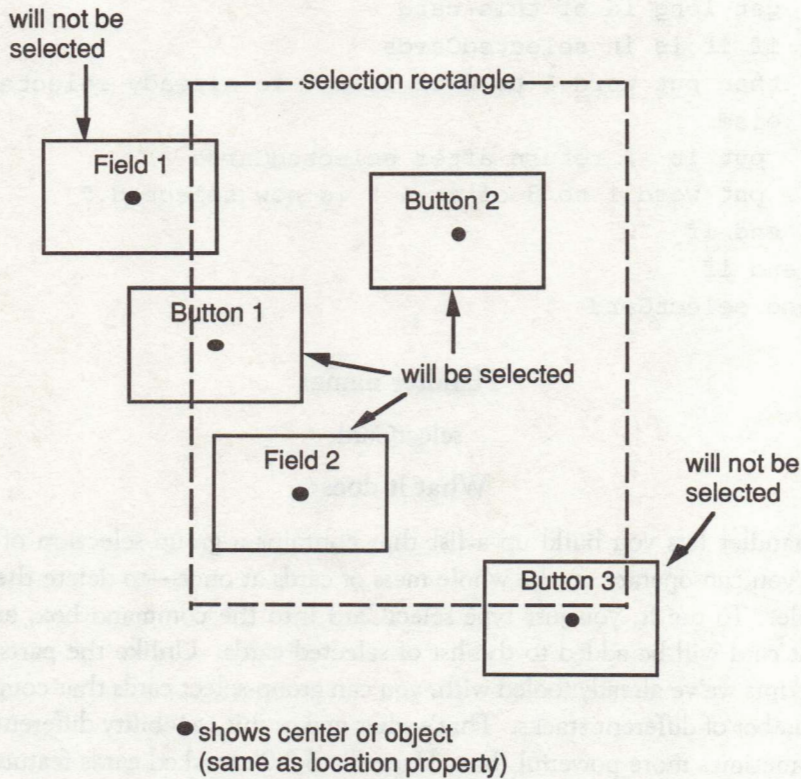


Figure 3-1 Selecting object centers

```

1. on selectCard clearsSelection
2.   -- script 3-3: Group selection of cards.
3.
4.
5.   -- This handler adds the current card to the
6.   -- currently selected cards or clears the list of
7.   -- currently selected cards.
8. global selectedCards
9. if clearSelection is not empty then

```



```

10.  put empty into selectedCards
11.  put "No cards selected."
12.  else
13.  get long id of this card
14.  if it is in selectedCards
15.  then put word 1 to 3 of it & " is already selected."
16.  else
17.  put it & return after selectedCards
18.  put word 1 to 3 of it & " is now selected."
19.  end if
20.  end if
21. end selectCard

```

Handler name:

selectCard

What it does:

This handler lets you build up a list that contains a group selection of cards. Then, you can operate on the whole mess of cards at once—to delete them, for example. To use it, you just type `selectCard` into the command box, and the current card will be added to the list of selected cards. Unlike the parts-selection scripts we’ve already fooled with, you can group-select cards that come from any number of different stacks. That’s what makes this capability different from, and sometimes more powerful than, HyperCard 2.0’s marked cards feature: you can actually work on a selection of cards that come from multiple stacks. If you try to select a card that’s already selected, the handler will tell you so.

SelectCard also lets you clear out the list of selected cards; to do this, just include a parameter when you call `selectCard` (any one at all will do, although “clear” is recommended to remind you of what’s going on). If a parameter is present, the list of selected cards will be cleared.

SelectCard takes advantage of the command box’s ability to remember the last command you typed in (see Appendix A for details). To select a bunch of cards, you’ll probably only have to type `selectCard` once. After that, you can go to the next card you want, type shift-tab, and “selectCard” will already appear in the command box.

Parameters:

There's one optional parameter. If it's there, no matter what it is, the list of selected cards will be cleared. We recommend that you use "clear" as the parameter. This gives you the flexibility to introduce other values for the parameter to do something else.

How it works:

This script is pretty straightforward. Compared to the last few we looked at, it's simple. Line 1 defines the handler. It's declared with a parameter, named `clearSelection`. This doesn't mean you have to send a parameter when you call it. It just gives the script the ability to access any parameter by name that is passed when the handler is called, as we'll do later in this script. Line 8 defines a global called `selectedCards` that, as you can probably guess if you're paying attention at all, is used to hold the list of selected cards in the group selection.

Within reach

A handler can get to all parameters passed to it, even if the handler doesn't declare it has any parameters in its first line. There are three HyperTalk functions to help you out: `params`, `param`, and `paramCount`. The function "params" returns the complete current message and parameters. You can also use "param" with a number to get to any single parameter; for example, `param(4)` returns the fourth parameter, if there is one, or empty if there isn't. Finally, `paramCount` can tell you just how many parameters were actually passed. Parameters undeclared by their handler are second-class citizens in only one sense: there's no way to refer to them by name in the handler, as you can with declared parameters.

Line 9 checks to see if the user passed anything for the `clearSelectionFlag` parameter ("if `clearSelection` is not empty"). If so, the if statement is true, and lines 10 and 11 are executed; these lines deselect any group-selected cards by putting empty into `selectedCards`, and then tell the user what's going on with a little bit of message box feedback. If there is no parameter, the else part of the if statement happens, which includes all the statements in lines 12 through 20.

The handler examines the current card using its long ID (line 13). It checks to see if the card has already been selected (line 14). If so, line 15 informs the user, yes, we are paying attention here. If the card isn't already part of the selection, line 17 puts its long ID into the selection and tells the user about it in the message box.

Why do we use long IDs like this to keep track of the selected cards?

```
card id 90125 of stack "Anderson:All Good People"
```

This kind of specification is designed to be as exact as possible, and is pretty immune to the problem of two cards having the same identifier. The card ID is guaranteed (or your money back) by HyperCard to be unique within the stack, and adding the full stack pathname to the identifier is necessary to help HyperCard find the card you're looking for.

That's it for selecting multiple cards. So far, we've shown ways to select multiple things, but haven't come up with anything to do with them. Hang in there, because we will (after only one more group selection script).

Group selection of backgrounds

To complete our collection of group selection tools, here's one that lets you create a selection consisting of a bunch of backgrounds.

```

1. on selectBkgnd clearSelectionFlag
2.   -- script 3-4: Group selection of backgrounds.
3.
4.
5.   -- This handler adds the current background to the
6.   -- currently selected backgrounds or clears the list
7.   -- of currently selected bkgnds.
8.
9.   -- selectBkgnd clear -- any non-empty parameter clears
10.  -- previous selection
11.  -- selectBkgnd -- adds current bkgnd to the list of
12.  -- selected bkgnds
13.
14.  global selectedBkgnds
15.  if clearSelectionFlag is not empty
16.  then put empty into selectedBkgnds
17.  else -- get around bug in long id of bkgnd
```



```

18.  get "bkgnd id " & id of this bkgnd & " of " & name of
      this stack
19    put it & return after selectedBkgnds
20.  end if
21. end selectBkgnd

```

Script tips: Parameters

HyperTalk's parameter passing system is one of the most interesting and flexible features of the language. Handlers may be defined to have any number of parameters; you define them by listing their formal names in the "on handlerName" line. When the handlers are called, the caller can pass all or some (or none) of the parameters along—HyperTalk won't care. Then, the handler's code can check to see if any of the parameters are empty, and if so, it can supply default values for them. This lets you pull some really nifty tricks with optional parameters in handlers. The only restriction here is if you call a handler and leave out some parameters. You then have to pass empty for the left out ones, like this:

```
msgWithThreeParams one,empty,three
```

This message takes three parameters. In this case, the second parameter is empty. If you want to leave off one or more of the rightmost parameters, you don't have to supply the commas or pass empty at all, so you could do it this way:

```
msgWithThreeParams one
```

This message will be sent without the second and third parameters, as done previously in `selectCard`: its one parameter is optional, and if the user doesn't want to send it, there's no problem. Just for completeness, we'll also point out that HyperTalk is tolerant when you pass too many parameters to a handler. It doesn't do anything with the extra parameters that are sent along, but you can still get to them with HyperTalk's parameter-sniffing functions, as described earlier.

Handler name:`selectBkgnd`**What it does:**

Like the previous script, this handler lets you build up a list of backgrounds you can perform some group operation on. As in group selected cards, the backgrounds you select can come from different stacks. You use it just like the previous card-selection script: type `selectBkgnd`, and the current background will be added to the selection. If you pass a parameter, you'll clear the list of selected backgrounds.

Parameters:

If you want to clear the list of selected backgrounds, pass something as a parameter. Just for fun, we recommend using “clear,” but you're free to use “Götterdämmerung” or “Pepperland” or your mother's maiden name if you want.

How it works:

Line 1 defines the handler, including the optional parameter `clearSelectionFlag`. Line 14 declares the global variable that's gonna hold the list of selected backgrounds. In line 15, the script checks to see if the `clearSelectionFlag` parameter was sent, and if it was, clears the selection, of course. If not, an interesting digression takes place in lines 18 and 19, which we'll talk about now.

What we'd really like to do here is get the long ID of the background and add it to our list. However, there's a problem: asking HyperTalk for the long ID of a background doesn't quite work right. When you ask for the ID of a background, HyperTalk ignores your choice of long, short, or just plain ID. All it ever gives you is the short ID, just a lonely number with no other identifying characteristics, like the words “background ID” or the stack name or anything else.

We won't let this little bug get us down, though. We'll persevere and implement our own version of long ID in line 18. We just tack on the words “bkgnd id” to the front of the ID, and we add “of” and the name of the stack to the end, which gives us a better specification for the current background. This is where HyperTalk's variety of ways to get information have proven to be very

useful, because it lets us work around a bug that might otherwise make us change our strategy.

After our trickery gets us past the bug in HyperTalk, we add the artificially constructed long ID of the background to the list of selected backgrounds kept in the `selectedBkgnds` global. And that's that.

Group Action

Aha! At last! You thought we were going to go on selecting groups of things all day, without ever doing anything with the selected things, didn't you? Well, once again, we've decided to go for the practical. Since we came up with such fun ways of making group selections, we thought it was only right that we also have some useful things to do to the selections. By the way, these scripts make excellent fodder for stepping through with the HyperTalk debugger, so you debugger fans will have an especially good time.

Delete a group of parts

Just to show what fun-loving guys we are, the first group operation we're going to define is the real destructive one—deleting the group of parts you selected. This also happens to be one of the most useful things you can do with a group selection, so it's not just our basically entropic nature that produced this script, but a love for great tools, too.

```

1. on deleteParts
2.   -- script 3-5: Group selection: delete a group of
3.   -- parts.
4.
5.   -- This handler deletes the parts that were
6.   -- previously selected with selectParts or
7.   -- rectSelectParts
8.   global selectedParts,selectedPartsCard
9.   if the long name of this card is not
       selectedPartsCard then
10.    -- only delete parts on current card
11.    put "go to " & selectedPartsCard
12.    exit deleteParts
13.  end if

```



```

14. lock screen
15. repeat with i = 1 to the number of lines in
    selectedParts
16.   get line i of selectedParts
17.   select it
18.   if it contains "button" then doMenu "Clear Button"
19.   else doMenu "Clear Field"
20. end repeat
21. choose browse tool
22. put the number of lines in selectedParts & " parts
    deleted."
23. end deleteParts

```

Handler name:

deleteParts

What it does:

The idea behind this handler is pretty simple. After you've selected a group of parts with `selectParts` or `rectSelectParts`, you can use this `deleteParts` command to delete them all at once. As a nice convenience, `deleteParts` makes sure you're looking at the right card (that is, the one you selected the parts on) before blowing them away. This prevents any fun surprises like deleting a whole bunch of fields you really didn't want to delete—the kind of thing that can spoil your whole day. Of course, if you try to delete background fields, you do get HyperCard's standard friendly warning message about deleting background fields and their text (see Figure 3-2).

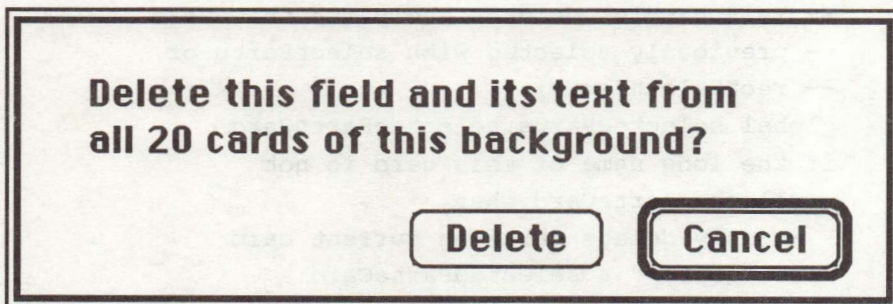


Figure 3-2 Friendly confirmation alert

Parameters:

None.

How it works:

In line 8, `deleteParts` says it's going to use a couple of globals defined earlier in the group selection scripts. `SelectedParts` contains the actual list of parts in the group selection, while `selectedPartsCard` tells which card contains the selected parts. Even though these globals were defined in the selection script, they have to be mentioned again here with the "global" keyword. If we don't say they're global, HyperTalk will assume that they're local variables, despite their names, and nothing will work right and it will be very depressing.

In lines 9 through 13, the handler uses the information kept in `selectedPartsCard` to figure out if we're currently looking at the card with the group selection. If not, no deleting happens. Instead, the handler puts "go to" and the name of the card with the selected parts into the message box, then in line 12 exits the handler and skips the rest of the script.

Why doesn't the script just go to the card automatically, delete the doomed parts, then return? Deleting things you don't want to delete can be a traumatic experience, so this script stops short of whisking you to another card. Instead, it places the command you need into the message box, so that you can execute it just by pressing the return key. If you're not as cautious as this, you might want to modify the script to take care of remembering the current card, going to the other one, deleting the parts, and returning.

If the script reaches line 14, it means that we're already on the fateful card, and it's time to start zapping those parts. Line 14 makes sure the screen won't be updated until all the deletions are done. This means we won't see each part individually selected and deleted, which will make the operation look a lot smoother. Try commenting out this line to see what it looks like without a locked screen.

The repeat loop in lines 15 through 20 does the dirty work in this handler. If you remember our parts-selection scripts, `selectParts` and `rectSelectParts`, the group selection that was stuck into the variable `selectedParts` was created by using one line per part name. So, the repeat loop is set up in line 15 to run once for each line in `selectedParts`. Line 16 grabs the current line of `selectedParts`. The "get" in line 16 puts the current line into "it," and line 17 selects the part

that's named in the current line. In this line, we're using HyperTalk's `select` command, which will automatically choose the right tool for the job, either the button or field tool, before selecting the part.

Lines 18 and 19 take care of deleting the part. There are two kinds of parts: buttons and fields. Line 18 checks to see if "it," which is the name of the current doomed part, contains "button." If so, a button is selected, which means that there'll be a menu item that reads "Clear Button." That's the one we want. If it doesn't contain "button," it has to be a field, and the appropriate menu item is "Clear Field," which is executed in line 19 as a result of the "else" part.

There are a couple of interesting questions here. The first is our method of deciding whether the selected part, the one that's about to be deleted, is a button or field. We're deciding by checking to see if the word "button" is in the current line from `selectedParts` (if it contains "button"). But what, you may ask, if we've got a devious, really unusual situation in which there's a field named something like "Buttonhole"—that is, a field whose name contains the word "button." Wouldn't this screw things up? No! Remember that `selectedParts` was built from the IDs, not the names, of the parts it contains. So, even if a field named "Buttonhole" was selected, its name hasn't been recorded in `selectedParts`, only its ID. This makes sure that the "if" in line 18 will get it right.

Another fun question comes from the curious-looking stuff in lines 18 and 19, the instructions that actually do the deleting. These lines look kind of awkward, especially in HyperTalk. Isn't there a simpler, one-liner way to accomplish the deletion? Well, these lines were written because the only way to delete things is by choosing a menu item—there's no direct "delete yourself" message in HyperTalk. And, whenever you accomplish something in HyperTalk by using `doMenu`, you must get the exact wording that's in the menu item, or you'll get a nice alert like the one you see in Figure 3-3. So, you can't just say

```
doMenu "Clear"
```

because in this case, there will never be an item named "Clear." It will always be "Clear Button" or "Clear Field," so we're stuck with having to decide which one we want.

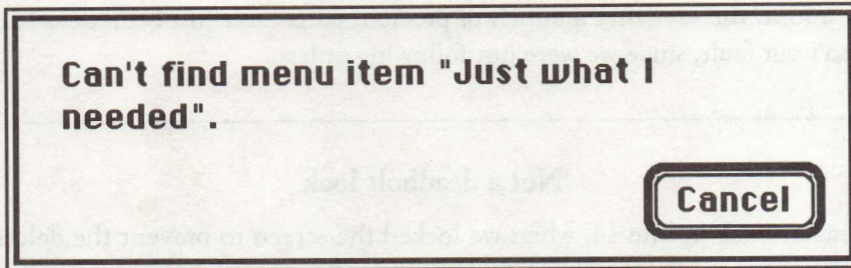


Figure 3-3 Another friendly HyperCard alert

Even though the menu items keep changing slightly, the command-key shortcuts always stay the same. So, for example, in the “Paste” menu item, no matter what you’re pasting, you can always get it done with command-V; in the same way, whatever it is that’s going to be cut, command-X will take care of it. Of course, HyperTalk scripts can type command keys by using the “type” command.

What happens if we replace lines 18 and 19 with the single command

```
type "x" with commandKey
```

which is the same thing as typing command-X? Since command-X is the shortcut for “Cut,” whatever is selected will be cut. Isn’t that just what we want? Well, almost. Our script is written to use “Clear” instead of “Cut.” Both of these commands delete whatever is selected. The difference between them is that “Cut” puts whatever it cuts onto the clipboard, which means the previous contents of the clipboard are lost; “Clear” leaves the clipboard alone. Our deleteParts script probably shouldn’t mess up the clipboard, so “Clear” is the right choice. Unfortunately, “Clear” doesn’t have a command-key shortcut, so there’s no way to have a single, neat “type” statement like we could if we were cutting instead of clearing.

Back now to the blow-by-blow description of our script. When the “Clear” command in line 19 takes place, and we’re clearing a background field, HyperCard will automatically supply the warning alert asking if you’re sure you want to delete the field and the text from all the cards sharing that background, which we showed you in Figure 3-2. Line 20 is the end of the loop that’s been killing off all the selected parts. By the time we get to line 21, all the selected parts will have been deleted, although the locked screen prevents us from seeing that yet. In line 21, we set the current tool back to browse. Finally, we use line

22 to inform the user that a bunch of precious parts have just been deleted, but it wasn't our fault, since we were just following orders.

Not a deadbolt lock

Remember back in line 14, when we locked the screen to prevent the deletions from showing up one at a time? In line 22, we're using the message box to tell how many parts were deleted. Won't the lock screen command still in effect prevent the message box from displaying this information? No. Locking the screen only prevents changes to the card display itself, not the message box, so we can still put up messages even if the screen is locked.

If you were paying close attention, you may remember we locked the screen. We seem to have neglected to unlock it again. What's going on? Well, HyperCard uses its idle time (that happy time when nothing else is going on and no other messages are being sent) to restore some transient properties to their usual values. This is done to prevent the user from being stuck in any uncomfortable modal situations, such as a locked screen, that really make sense only when a script is executing (this irritating situation is known as "surprise a la mode").

One of the properties reset at idle time is `lockScreen`. Since some idle time happens whenever a handler finishes executing, this property will always be reset to false after a handler is done. This means you never have to put "unlock screen" at the end of a handler—this will happen automatically when the handler finishes. Of course, if you need to do something in the handler requiring the `lockScreen` to be changed (if you only want to lock the screen for a certain operation, for instance), you have to fiddle with the property yourself. For more information on this phenomenon, see the script tip entitled "Locked up tight" in Chapter 2.

After using `deleteParts`, the parts are absolutely gone. There's no way to use "Undo" to get them back. We've still got a list of them in `selectedParts`, of course, but all the information we have left about them is their IDs, certainly not enough to reconstruct them. Is there any way we can give the `deleteParts` handler the ability to undo its work, to undelete the parts? Hey, of course there

is—this is HyperCard, folks! There are lots of things you can do: for example, create a handler that catches deleted things and saves them for a rainy day. See what else you can come up with.

Paste a group of parts

In this handler, we'll use group selection of parts as a way to spread a bunch of parts around in more than one place by pasting the whole selection as a group.

```

1. on pasteParts
2.  -- script 3-6: Group selection: paste a group of
3.  -- parts.
4.
5.  -- This handler pastes the parts that were previously
6.  -- selected with selectParts or rectSelectParts.
7.
8.  global selectedParts,selectedPartsCard
9.  set lockMessages to true
10. set lockScreen to true
11. repeat with i = 1 to the number of lines in
    selectedParts
12.   push card
13.   go to selectedPartsCard
14.   select line i of selectedParts
15.   type "c" with commandKey
16.   pop card
17.   set editBkgnd to line i of selectedParts contains
       "bkgnd"
18.   type "v" with commandKey
19. end repeat
20. set editBkgnd to false
21. choose browse tool
22. put the number of lines in selectedParts & " parts
    pasted."
23. end pasteParts

```


Handler name:

pasteParts

What it does:

This handler lets you select some parts with one of the group selection scripts (selectParts or rectSelectparts), then paste them into other cards and backgrounds all at once. This works a lot like the deleteParts scripts that we just dissected, although there are a few interesting differences and some new techniques you might want to pick up on.

Parameters:

None.

How it works:

Like most of the scripts in our group selection system, pasteParts uses selectedParts and selectedPartsCard, so they're declared as global variables in line 8. We set lockMessages to true, and we also lock the screen in line 10 to keep our screen cleaner and fresher smelling.

The repeat loop in lines 11 through 19 does the work in this handler. The strategy this time is a little bit different from deleteParts, for a few reasons. First, we have to switch between two cards, which we'll call the source and the destination (calling them the paster and the pastee would just be too obnoxious), as we add the new parts. Second, this time we're not using "Clear," so we should be able to take advantage of the "type with commandKey" trick we described earlier, but couldn't use in deleteParts. A third difference is that some parts belong to the card, while others belong to the background. Why are we doing things this way? You'll see the thrilling answers to this and other questions as we describe the script.

The first thing we have to do is go over to the card holding the selection (the source) and start grabbing parts. HyperCard, of course, has no built-in way to select more than one part at a time (that's why we invented this elaborate group selection system). This means we'll have to do a lot of bouncing around between the source and destination cards as we grab one part at a time and move it over.

In line 11, we start processing the parts in the usual way, by looping once for each line in selectedParts, which, as you surely know by now, contains one line that describes each part in the group selection. We have to journey over to the source card to get the first part, so line 12 saves our place by remembering the current card

with a “push card” command. Then, in line 13, we actually journey to the source card to start grabbing parts.

Outstanding in our field

One of HyperTalk’s neat features is that you can do some fooling around with a card without having to go to that card. For example, if you need to get the contents of a field on a card that’s not the current card, you can do it without using a “go card,” as long as the card is in the same stack. All you have to do is add a card specifier to the field you want, like this:

```
get field "he made it" of card ID 6530
```

This command will get what’s in the specified field on the other card without having to change the current card. For a small number of fields (less than about ten), this is usually a lot faster than going to the other card, and it doesn’t disrupt the screen for the user. If you need to get more than ten fields, you’ll find it faster to go to the card first. We can’t use this technique in pasteParts, though, because we need to select the parts, not just get its properties, and there’s no way to select a part by “remote control” like this.

After we’ve used the “go to” command in line 13 to get to the source card, we’re ready to select parts. The loop variable “i” holds the number of the current part in the selection we want to work with. Line 14 uses “i” to select the part we’re interested in. Remember, you don’t have to choose the button or field tool when you use HyperTalk’s “select” command. It will automatically choose the right tool when it selects an object.

Now that we’ve selected the part we’re interested in, we need to make a copy of it to take to the destination card. We can do that by using the “Copy” item in the Edit menu. As we discussed earlier, items in the Edit menu change slightly, depending on certain conditions. The “Copy” item changes according to what’s currently selected. If a button is selected, it says “Copy Button,” as you might imagine, and you can probably guess that if a field is selected, it’ll say “Copy Field.”

When we were deleting things with the “Clear” item, we had to use an “if” statement to guarantee that we got the wording of the menu item exactly right (see the discussion of the `deleteParts` handler). We also described a HyperTalk trick that lets you use the “type” command to execute menu items with command-key shortcuts. This time, since command-C is a shortcut for “Copy,” we can use that trick to make the script simpler, which is what you see in line 15. This puts the selected part on the clipboard.

Now we have to go back to the destination card and paste that sucker in. The “push card” command in line 12 remembered the destination card for us, and line 16 takes us back to where we started by popping the card we pushed.

When we originally selected parts to create the group selection, all kinds of parts, both background and card, were eligible. We didn’t have to do anything special to indicate which were which. However, when we paste parts, it’s a little bit trickier. Newly created parts, including those you paste, belong to the card or background based on the `editBkgnd` property (which you usually set with the “Background” item in the Edit menu). When `editBkgnd` is set to true, it’s usually said that HyperCard is in “background mode.”

Naughty, naughty

You may know that “mode” is usually a dirty word in the Macintosh interface. It’s true, modes are artificial restrictions on users and often should be avoided, but they occasionally have their place in software. One of HyperCard’s ways of dealing with background modes is to paint the menu bar like a zebra, just to remind you that you’re there. This distinctive visual clue helps you remember you’re in an unusual mode.

So, if you copy a background field, then paste, the newly pasted field will be a background field only if `editBkgnd` is set to true; otherwise, it’ll become a card field, which is probably not what you wanted. We have to be careful about dealing with background fields and buttons.

Line 17 implements a little cleverness to make sure that background parts in the group selection become background parts when they’re pasted. It works by checking to see if the current part in the group selection is a background part,

and then setting the `editBkgnd` property appropriately. If you're pretty new to scripting or programming in general, you might think of a simple, straightforward way to accomplish this task, and it might look something like this:

```
if line 1 of selectedParts contains "bkgnd"
then set editBkgnd to true
else set editBkgnd to false
```

There is, however, an easier way, and now we'll talk about how it works. First, notice the relationship between the expression

line 1 of selectedParts contains "bkgnd"

and what the `editBkgnd` property is set to. They're the same: if the "line i of selectedParts..." expression is true, then `editBkgnd` becomes true; if the expression is false, `editBkgnd` is false, too.

Next, notice that the "line i" expression is just that: an expression giving a logical result, true or false. The last part of a "set" command is a logical expression, which is usually just the constant true or false, but can also be something more complex, like an expression. (Imagine a light bulb turning on overhead.) So, what if we tried combining the two lines together, as in line 17? You might find line 17 easier to read if we added parentheses, like this:

```
set editBkgnd to (line i of selectedParts contains "bkgnd")
```

The parentheses here don't change the meaning of the expression. They just make it clear where to find the beginning and the end of the logical expression.

Terminology corner

When you're talking to computer types, values that are either true or false are called boolean values—what we've been calling logical values so far. So, a function like `shiftKey`, which returns either true or false, is called a boolean function. The term "Boolean" is named after the famous mathematician George Boole, who invented 0 and 1 but got stuck after that.

Everything after the "to" in this line is a logical expression—that is, it reduces to either true or false. HyperTalk first figures out whether this expression is true

or false, then sets `editBkgnd` to that result. The trick you see in line 17 is a common technique you can use if you're solving a logical question (such as whether a string contains another string) to set a logical property (like `editBkgnd`). See, these things are easy once they're demystified (and if you weren't mystified about it to begin with, then you don't have to worry about it at all).

We've still got some work to do to finish up `pasteParts`—like, we haven't pasted anything yet. We take care of that in line 18, which uses the “type” command to choose “Paste” from the Edit menu. This saves us from having to worry about whether it really says “Paste Button” or “Paste Field.” The end of the repeat loop shows up in line 19, so we'll continue through the loop for the next group-selected part and go back to line 12.

After we've gone through the loop once for each part in the group selection, all the pasting has been done, and all the parts should have been correctly installed in the card or background. To finish up, we make sure that `editBkgnd` is turned off and choose the browse tool in lines 20 and 21. Finally, we use line 22 to display the obligatory message that says how many parts were pasted. Remember, this message will appear even though `lockScreen` is true, because `lockScreen` only affects the card window (which gets unlocked at idle time anyway).

Align properties of a group of parts

If you've ever wanted to change any property of a bunch of parts all at once, such as choosing a different font for a set of fields, lining up some buttons, or unlocking the text of many fields quickly, you'll be able to use `alignParts`. This handler is useful when you've already created a bunch of objects, then you decide you want to change something about all of them at once; the font, for example. With `alignParts`, you make that kind of change easily and quickly, letting the computer do all the tedious work.

1. on `alignParts` prop
2. -- script 3-7: Group selection: align properties of a
3. -- group of parts.
- 4.
5. -- This handler aligns specified properties of the
6. -- parts that were previously selected with


```

7.  -- selectParts or rectSelectParts
8.  global selectedParts,selectedPartsCard
9.  if the long name of this card is not
    selectedPartsCard then
10. -- only align parts on current card
11.  put "go to " & selectedPartsCard
12.  exit alignParts
13. end if
14. if prop is empty then
15.  ask "Align which property?"
16.  if it is empty then exit alignParts
17.  put it into prop
18. end if
19. get value of (" short " & prop & " of " & line 1 of
    selectedParts)
20. put " to " & quote & it & quote into masterProp
21. put "set " & prop & " of " into setProp
22. lock screen
23. repeat with i = 2 to the number of lines in
    selectedParts
24.  do setProp & line i of selectedParts & masterProp
25.  end repeat
26. end alignParts

```

Handler name:

alignParts

What it does:

You can use this handler to make sure the parts in a group all have the same settings for some of their properties. First, you select the parts with selectParts or rectSelectParts. The first part you select is called the *guide*. When you use alignParts, the other parts in the selection have their parts set according to the guide. For example, if you want all the parts in the selection to have the same location for their left edges (which is represented by the left property), make sure

that the first part you click on is already at the desired position. Then, you can use `alignParts` to set the left property of all the parts in the entire selection to match the setting of that property in the guide part (the first one you selected).

Script tips: Maintaining an even keel

Notice in `pasteParts`, as well as in some other handlers, we don't save and restore the current tool or the background mode. Instead, we just "jam" these to the most common values, the browse tool and edit background turned off, at the end of the handler. Isn't this rude? Didn't your mother and Robert Fulghum always teach you to leave things the way you found them? What if a user calls `pasteParts` with the button tool set, and it doesn't respect this setting?

The answer is that it's not always a good idea to preserve every setting. For some major, modal settings, like the tool and the `editBackground`, it's better to leave the user in a default, well-known state when you're done. Leaving most handlers with HyperCard in background mode, or with a tool other than browse selected, can be disconcerting. Even if those were the settings when the command was sent, most users are more comfortable seeing the default settings, so that's the way our handler works. Of course, if you'd rather the script save and restore the tool and the edit background, just go ahead and add that capability. After all, it's your book!

Parameters:

This handler takes one optional parameter, called `prop`. It's used to tell which property you want to set, and it can be the name of any property of a field or button. You can look in Figure 3-4 to see the list of button and field properties. This parameter is optional; if you don't include a parameter when you call `alignParts`, the handler will ask you which property you want to align.

Button and field properties:

bottom	name	textAlign
bottomRight	number	textFont
height	rectangle	textStyle
id	right	top
left	script	topLeft
location	style	visible
		width

Button properties:

autoHilite
hilite
icon
sharedHilite
showName

Field properties:

autoLineHeight	sharedText
autoTab	showLines
dontSearch	textHeight
dontWrap	textSize
lockText	wideMargins
scroll	

Figure 3-4 Button and field properties

Note: if your selection has both buttons and fields, you can only set the properties that work for both buttons and fields. If you try, for example, to set the autoHilite of a field, you'll get an alert telling you what a foolish thing that was to do.

How it works:

Like the other scripts in the group selection of parts system, alignParts uses the globals, selectedParts and selectedPartsCard, which were set up when the group selection of parts was first created. Lines 9 through 13 are involved in finding out if we're on the right card. Line 9 checks to see if we're currently looking at the card with the selected parts. If not, it puts a "go" command into the message box that will take us to the right card (line 11), then exits the handler. This is the same strategy we used in deleteParts if the current card wasn't the card where the parts were selected.

We said this handler takes an optional parameter called `prop` which specifies the property we're going to mess with. Lines 14 through 18 make sure there's a property to align even if no parameter was passed. In line 14, we check to see if the handler was called with any value for `prop`. If not, the `if` statement causes line 15 to be executed, which asks which property the user was interested in. Line 15 uses the `"ask"` command, and puts whatever the user types into `"it."` If the user is some kind of troublemaker who calls the handler with no parameter and then refuses to answer the question, or clicks the `"Cancel"` button, `"it"` will be empty, and `alignParts` gives up in line 16. Otherwise, for cooperative folks who supply an answer, line 17 does just what it says (put it into `prop`) to make sure that `prop` contains the name of the property we're going to work with.

I write the songs

This section of the handler uses an interesting idea to deal with a parameter that may or may not be there. If the parameter wasn't passed, our handler tries very hard to supply a value for it anyway by asking the user for a value and then setting the parameter to that value. This is called writing to your parameters. Your parameters are nice enough to call you; why not write to them now and then?

One look at line 19 and you'll know it's doing something interesting, obscure, or magical, because it obviously requires some study. Remember, the first part selected is special, because it's the guide. So, our first job is to figure out which property we're going to set, then get the value of that property from the guide.

What are some of the properties we might be aligning? One of them is `name`, which is both a button and a field property. If we ask for a part's name, we'll get something that looks like this:

```
card button "Reckoning"
```

Hmm...that's not really what we want. If we use that value to set another button's name, we'll get a button with the name

```
card button "Reckoning"
```


when what we really want is a button named

Reckoning

We want a way to get *just* the part's name, without the "card button" jazz on the front of it. Since this is a pretty common request, HyperTalk provides an easy way to do it: we ask for the short name of the part instead of just the name. This gives us something like this:

Reckoning

which is exactly what we want.

Now let's consider some of the other properties we might choose. A common request is to select a bunch of fields and set them all to use the same font, kept in the `textFont` property. If we ask for a part's `textFont`, that's exactly what we get; there's no additional stuff on the front of it like we got when we asked for a part's name.

But this seems to present a problem. For some properties, like name, we *want* the short form of the property. For others, most of them, in fact, like `textFont`, there is no short form of the property, only a standard form, and we're perfectly happy with it. Unfortunately, there's no easy way to know which properties require us to use "short" from the ones that don't.

What can we do? Luckily, HyperTalk provides a solution. Even though there's no such thing as the "short `textFont`" or the "short rectangle" of a part, HyperTalk tolerates these expressions and simply ignores the "short" and gives you the property. In fact, HyperTalk ignores "short," "abbreviated," or "long" for properties with only one form, and it doesn't complain about them. This means we can use "short" whenever we ask for a property. When it's needed, it'll work, and when it's not needed, it won't hurt anything.

Back now to line 19. Its mission is to get the value of the appropriate property. As we've just discussed, to be sure we get what we want, we have to ask for the "short" version of the property. So, line 19 assembles a string that consists of the word "short," the property in `prop`, the word "of," and the first selected part. This produces an expression like

```
short textFont of card button id 5
```

which is placed in the parentheses in line 19. Yes, it's true that "short `textFont`" looks a little weird, but remember that HyperTalk won't mind it, so why should

you? The rest of the statement, “get value of...,” retrieves the value of the property specified in that expression and puts it into “it.”

When we actually get ready to start setting the properties of our selected parts, we’ll need a statement that says something like

```
set textFont of card button id 5 to Helvetica
```

We’ve already gotten the value of the property (in this example, it’s Helvetica) into “it,” and we’ve also got the “textFont” expression in prop, so we can reuse it. Lines 20 and 21 are used to create this standard “set” command that we’ll use. In line 20, we create a variable called masterProp to hold the part of the command that tells what to set each part’s property to. Remember, “it” holds the value of the property we’re going to jam into each part. To continue with the example we’ve been using, masterProp would contain

```
to "Helvetica"
```

which is what we’re going to set all the textFonts to. In line 21, we construct the front half of the statement we’re going to need and stick it into a variable called setProp. This variable is made by stringing together “set,” the property name, and “of,” which is what line 21 does. Our example will be

```
set textFont of card button id 5
```

and you can check it yourself by looking at line 21.

We now have everything ready for the big property-changing party, which takes place in the repeat loop in lines 23 through 25. Before we start changing ’em, we lock the screen in line 22 so that all the changes will become visible at once. Things will also go faster, since avoiding screen updates speeds up the whole operation. The repeat loop beginning in line 23 will run once for each selected part except the first one.

Remember that the first part is the guide for the rest of the parts in the selection, and is therefore immune to our messing around here. It’s already set the way we want it, so the repeat loop starts with line 2 of selected parts. There’s actually only one statement in the repeat loop (line 24). It strings together the words from setProp, the current line of selectedParts, and masterProp, producing a command in this form:

set the `textFont` of card button id 5 to Helvetica

By using this command with the “do” keyword, as line 24 does, the command is executed, and the property is set to the value we want. This operation is repeated for every part in the group selection. Of course, they’re all set to the same value of the desired property, just like we wanted. Isn’t it nice when things work out?

Offset (distribute) a group of parts

The `alignParts` handler is useful when you need to make a bunch of parts have the same value for a property. `OffsetParts` is for when you don’t want the same value, but you’d like to numerically spread out a property evenly across a group of parts. This lets you do such fun things as space a set of buttons nicely down the screen.

```

1. on offsetParts prop,offset
2.  -- script 3-8: Group selection: offset (distribute) a
3.  -- group of parts.
4.
5.  -- This handler offsets specified properties of the
6.  -- parts that were previously selected with
7.  -- selectParts or rectSelectParts
8.  global selectedParts,selectedPartsCard
9.  if the long name of this card is not selectedPartsCard
10   -- then only offset parts on current card
11.    put "go to " & selectedPartsCard
12.    exit offsetParts
13.  end if
14.  if prop is empty then
15.    ask "Offset which property?"
16.    if it is empty then exit offsetParts
17.    put it into prop
18.  end if
19.  if offset is empty then put 30 into offset
20.  if offset is "spread" then put partPropSpread(prop)
    into offset
21.  get value of (prop & " of " & line 1 of selectedParts)
22.  put " to " & it into masterProp
23.  put "set " & prop & " of " into setProp

```



```

24. lock screen
25. repeat with i = 2 to the number of lines selectedParts
26.   add offset to last word of masterProp
27.   do setProp & line i of selectedParts & masterProp
28. end repeat
29. end offsetParts
30.
31. function partPropSpread prop
32. global selectedParts -- assumes we're already at the
   right card
33. put the number of lines in selectedParts into
   partCount
34. if partCount < 2 then return 0
35. put value(prop & " of " & last line of
   selectedParts) -
36. - value(prop & " of " & first line of selectedParts)
   into spread
37. return round(spread/(partCount - 1))
38. end partPropSpread

```

Handler name:

offsetParts

What it does:

This handler is similar to alignParts, but instead of setting a bunch of properties to the same value, it distributes the values over a range. This is most useful for arranging parts evenly spaced on the card. Of course, if you're clever, you can think of other neat uses for it, too. Any numeric property of a part is fair game for offsetParts. Properties that aren't numeric, like name and script, don't make sense with offsetParts.

Parameters:

The offsetParts handler takes two parameters, and both of them are optional. The first parameter is the property you want to work with. If you don't pass a value for this parameter, which is called prop, the handler will ask you which

one you want, just as in `alignParts`. The second parameter gives the value you want to offset the properties by. For example, if you're spacing buttons 16 pixels apart vertically by offsetting their "top" properties, you should pass 16 for the second parameter, which is named "offset" in the handler. If you don't pass a value for this parameter, the handler plugs in a default value of 30. You can also pass "spread" for the second parameter. This will make `offsetParts` spread the parts out evenly, using the first and last parts as anchor points. Figure 3-5 shows what happens when you use the spread parameter.

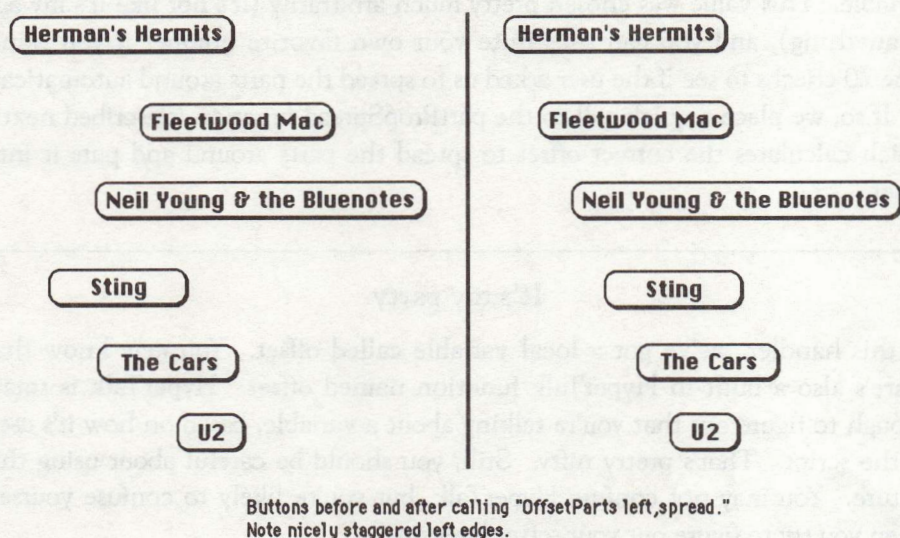


Figure 3-5 Effect of `OffsetParts`

How it works:

Most of this script is very similar to `alignParts`, and the first few lines ought to look as familiar to you as your family by now. Line 8 declares our interest in those ubiquitous `selectedParts` and `selectedPartsCard` globals. Line 9 through 13 are the standard check to see if we're on the card we started with, and if not, they put the standard message into the message box to take you back there if you want to go.

Once we're past that stuff, we get to the good part. Lines 14 through 16 use the same tactic that we employed in `alignParts`. If the user didn't pass anything for the `prop` parameter, there's not much we can do. So, we give the user a sec-

ond chance in line 15 by asking, please, to tell us the name of the property we're going to offset. If the user doesn't want to play and doesn't answer the question ("if it is empty" in line 16) then the handler gets mad and just gives up. Assuming that we've got a cooperative user, line 17 puts the user's answer into the variable `prop` and we can use it just as if it were sent right along with the command in the first place.

When the user doesn't pass anything for the second parameter (which is kept in the offset variable), line 19 finds out, and it puts a default value of 30 into the variable. This value was chosen pretty much arbitrarily (it's not like it's my age or anything), and you can substitute your own favorite number if you want. Line 20 checks to see if the user asked us to spread the parts around automatically. If so, we place a quick call to the `partPropSpread` function (described next), which calculates the correct offset to spread the parts around and puts it into `offset`.

It's my party

In this handler, we've got a local variable called `offset`. You may know that there's also a built-in HyperTalk function named `offset`. HyperTalk is smart enough to figure out that you're talking about a variable, based on how it's used in the script. That's pretty nifty. Still, you should be careful about using this feature. You may not confuse HyperTalk, but you're likely to confuse yourself when you try to figure out your script later on.

Now that we've got legitimate values for both parameters, we can proceed. The strategy we're going to use is very similar to what we did in `alignParts`. We need to build a couple of partial expressions that we can string together with the name of a part to change the desired property. We start in lines 21 and 22 by building the `masterProp` variable, which contains the property value we're going to use (didn't think we'd be talking about real estate in this book, did you?).

Notice that this line is a lot like the corresponding one in `alignParts`, but with an interesting difference. In `alignParts`, we asked for the "short" form of the property, and we launched into a long discussion of just exactly how and why we did that. This time, there's no mention of "short.." Why? In this handler, we're offsetting parts, which means we can only use properties with numbers in them,

and there aren't any numeric properties of parts that use "short." So, we've left off "short" this time.

In line 23, we set up the other variable we'll need by putting the "set" part of the statement into `setProp`. We close the drapes in line 24 so that nobody can watch while we're working. Lines 25 through 28 are the repeat loop that changes the properties. Three of the four lines in this loop are identical to the corresponding loop in `alignParts`. The new stuff is in line 26. Without line 26, we would just be jamming the same value into each part's property, as we did in `alignParts`. In line 26, the value at the end of `masterProp` (the value we're using to set the properties) is changed by adding "offset" to it. Then, line 27 makes the change in the property. Line 28 ends the loop, line 29 ends the handler, and that's the end.

Handler name:

`partPropSpread`

What it does:

This function is used as a utility by `offsetParts`. It's called if you specify that you want to spread the selected parts evenly between their extreme points.

Parameters:

This function takes one parameter, named `prop`, representing the property you're interested in spreading out. Normally, `offsetParts` calls `partPropSpread` for you, so you'll never have to dial it directly.

How it works:

The job of this function is to figure out how to evenly spread the parts out using the property chosen by the caller. Typically, the desired property will be something like top or left, to make a bunch of buttons or fields appear evenly spaced on the screen. To do this, the handler finds the first and last items selected, determines the value of the chosen property for these items, then averages the result among the total number of parts.

Line 32 indicates we'll be using the global `selectedParts`. In order to average the property, we need to know how many parts there are. We can figure this out by seeing how many lines there are in `selectedParts`, and we use that value to set

up a variable called `partCount` (line 33). In line 34, we make sure that there are at least two parts to average. If there aren't, we've got a degenerate case where there's nothing to spread, so we just return with zero.

Assuming that at least two parts are selected, the single statement spread over lines 35 and 36 does the work. It looks hairy, so let's demystify it.. Basically, it's just figuring the difference between the value of the chosen property in the first and last of the selected parts. The statement first builds an expression like this:

```
prop & " of " & last line of selectedParts
```

What does this get us? Well, remember, `prop` is the name of the parameter passed to this function; in other words, it's the property that we're interested in. The expression "last line of selectedParts" gives us the last line in `selectedParts` (not a surprise there). Putting the whole thing together, we have an expression that refers to the property we want in the last selected part. Putting "value ()" around it, as in line 35, simply gets the actual value of that property.

In the same way, the second expression finds out the requested value from the first part in the set of selected parts. When we've finally computed both of these values, the value from the first part is subtracted from the last part's value, and the result is put into a variable called `spread`. To help understand this little transaction, let's play Mr. Computer and assume some real values to simplify the statement in lines 35 and 36. Those of you with an actual Macintosh and an actual copy of HyperCard close at hand may want to step along here with the HyperTalk debugger for even more fun. We'll assume that the desired property is "left," which is the left coordinate of the part's location, and we'll further pretend that the first selected part is card button ID 4, and the last selected part is card button ID 12. In this case, lines 35 and 36 mean this:

```
put value (left of card button ID 12) -  
- value (left of card button ID 4) into spread
```

Each of the calls to HyperTalk's `value` function returns the value of the expression inside. So, in our example, this statement subtracts the left property of the first selected part from the left property of the last selected part, then puts the answer into `spread`. In line 37, we divide `spread` by the number of selected parts minus 1. The effect of this is to spread the difference evenly over the range of selected parts. The result of this division is then sent back as the function result.

Spread too thin

Notice that `partPropSpread` uses the first and last selected parts as the extreme points, the ones to use when figuring how to spread things out. But what happens if the first and last selected parts are not the parts at the extreme locations, as so beautifully illustrated in Figure 3-6? Our script will simply spread the parts out using the first and last selected parts as the anchor points, which may not be what you want. This implies that you need to be careful which parts you select first and last if you're going to spread them out. In particular, if you use `rectSelectParts`, you may be surprised at how things get spread out, because the parts will have been selected in their positional order (see `rectSelectParts` to understand why).

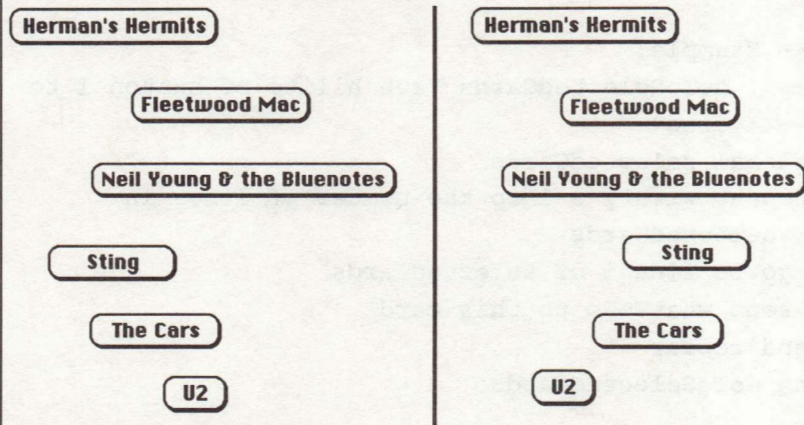


Figure 3-6 OffsetParts gone awry

Actually, this is a feature (stop that snickering). It lets you spread the parts using whichever two parts you choose as anchors, not just the top and bottom ones. One interesting change you could make to `offsetParts` would be to add another possible value for the `offset` parameter that always spreads the parts between the extremes of the chosen property rather than just the first and last selected. In our example above, this would mean sifting through the "left" of each selected part to find the smallest and largest values, then using them to compute the offset.

Do something to every card in the group selection of cards

This handler lets you quickly do something to all the cards you've gathered together in the group selection of cards. For example, you can use it to make copies of cards, print cards, or set a card field on each card to an initial value. Remember that the selected cards can come from many different stacks, so you can really get a lot done with this handler. Of course, before you use this handler, you should select the cards using `selectCard`, which we talked about earlier.

```

1. on doToSelectedCards whatToDo
2.  -- script 3-9: Group selection: act on group-selected
3.  -- cards.
4.
5.  -- This handler repeatedly executes a given statement
6.  -- for every card in the currently selected cards.
7.
8.  -- Example:
9.  -- doToSelectedCards "set hilite of button 1 to
10. -- true"
11. global selectedCards
12. repeat with i = 1 to the number of lines in
    selectedCards
13.   go to line i of selectedCards
14.   send whatToDo to this card
15. end repeat
16. end doToSelectedCards

```

Handler name:

`doToSelectedCards`

What it does:

This simple handler provides you with a way to execute a HyperTalk command for a set of cards all at once. First, you should select the cards by using the `selectCard` command script. Then, use `doToSelectedCards` to accomplish what you want to do to each card. This handler can be very useful if you've created some cards and then discover that you want to add some card part, like a card button, to each card.

Parameters:

This handler takes one parameter, and it's required. The parameter, which is put into the variable named "whatToDo," must be a valid HyperTalk message, just like one you'd put into a script. DoToSelectedCards will send this message to every card in the group selection of cards, one at a time. If you want to refer to the card being operated on, you can use HyperTalk's "target" function.

How it works:

This handler has to have access to the selectedCards global, so it declares that in line 11. The work in this simple handler is done in line 12 through 15. Line 12 sets up a repeat loop to execute once for each line in selectedCards; that is, once for each card in the group selection of cards. Line 13 goes to the current card in the group selection. After we're there, line 14 does the job. It sends the command in "what to do" to the card. In line 15, the loop ends, and line 16 is the end of the handler.

Happy ways

You have to send the HyperTalk message as a parameter when you call doToSelectedCards. This can make it a little tricky to send quoted strings along, but it can be done if you're careful. Say, for example, that you're taking a survey, and you want to know how many card buttons there are on every card in your group selection. You want to list the numbers of buttons in the message box, separated by commas, and you don't care which card each one came from. You're only interested in how many there are. So, you want the message box to look something like Figure 3-7.



Figure 3-7 Message Box

What's the message you want to send to each card? Well, if we were going through in that tedious, human way of ours and getting the information, the command you'd send to each card would be

```
put number of buttons & "," after msg
```

However, remember, you're going to be sending this entire string as a quoted parameter to the command library, which makes it tough to put quotes around the comma. Instead, you can use the built-in HyperTalk constant named "quote" to solve the problem, and your command line will look like this:

```
doToSelectedCards ("put number of buttons & " &  
quote & "," & quote & "after msg")
```

(Obviously, this is two lines here because it won't fit across this page. It must be one line when you type it into the command box.) This works because it first assembles the string that's going to be sent off to doToSelectedCards. It adds together the "put number of buttons & ," the quote, the comma, another quote, and "after msg" strings, then ships it off to the handler. The small size of the command box that you have to type in makes this just a little more fun. For better maneuverability, you can use the arrow keys to scroll sideways through this dialog, or you can drag past the left or right edges to make it autoscroll. You might also want to enhance doToSelectedCards to prompt you for a message and quote it for you.

If typing all that seems like too much of a mess, you can also define a handler to send this message for you. Then, you can type just that handler's name into the command box. This is also the only way to send multiple statements at once through doToSelectedCards. If you do this, you should put the handler in the script where it covers the smallest possible domain. If your message will be traveling through cards of one background only, put it in the background script; otherwise, use the stack script.

Do something to every background in the group selection of backgrounds

This handler is almost identical to `doToSelectedCards`, except that it operates on the group selection of backgrounds. This command is useful if you've created multiple backgrounds, then want to make some uniform change to them, such as adding or modifying a background field. We've included it here for completeness, but hey, you might actually need to use it someday, and when you do, you'll sure be glad it's here!

```

1. on doToSelectedBkgnds whatToDo
2.  -- script 3-10: Group selection: act on group-selected
3.  -- backgrounds.
4.
5.  -- This handler repeatedly executes a given statement
6.  -- for every background in the currently selected
7.  -- background.
8.  -- Example:
9.  -- doToSelectedBkgnds "set cantDelete of this
      bkgnd to true"
10.
11. global selectedBkgnds
12. repeat with i = 1 to the number of lines in
      selectedBkgnds
13.   go to line i of selectedBkgnds
14.   send whatToDo to this bkgnd
15. end repeat
16. end doToSelectedBkgnds

```

Handler name:

`doToSelectedBkgnds`

What it does:

This handler assumes you've used `selectBkgnd` to create a group selection of backgrounds. Once you've got that group selection, you can use `doToSelectedBkgnds` to perform some identical action to each of them. Some

examples of what you might want to do here include setting the backgrounds' `cantDelete` property, clearing out the backgrounds' scripts, or collecting the names of all the backgrounds in a stack into a field somewhere.

Parameters:

There's one parameter, named `whatToDo`, originally designed to contain the predicted final standings for the National Basketball Association's Midwest Division for the 1993-94 season, using a previously undocumented feature of HyperCard. After careful thought, Dan changed it so that it holds the message that you want to send to all the selected backgrounds.

How it works:

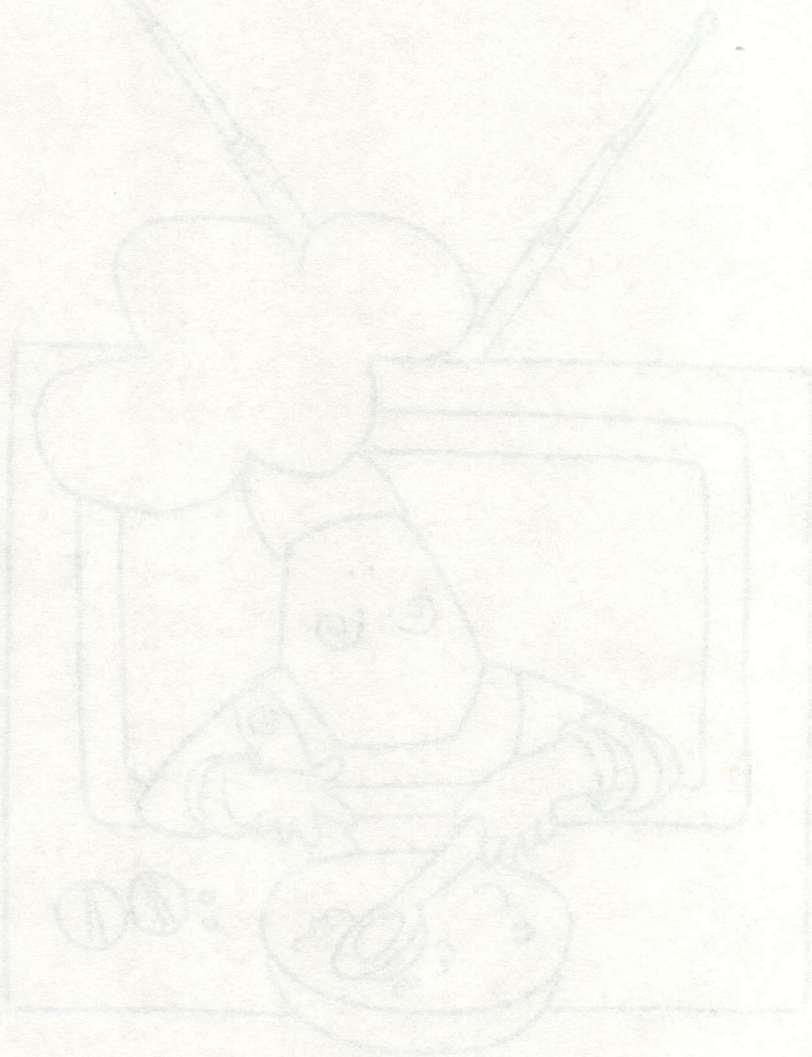
This handler works very much the way that `doToSelectedCards` works. In fact, if you look at it, you'll see that it works exactly the same way as `doToSelectedCards`. If you start with the script for `doToSelectedCards`, then replace every occurrence of "card" with "bkgnd," you'll get `doToSelectedBkgnds`.

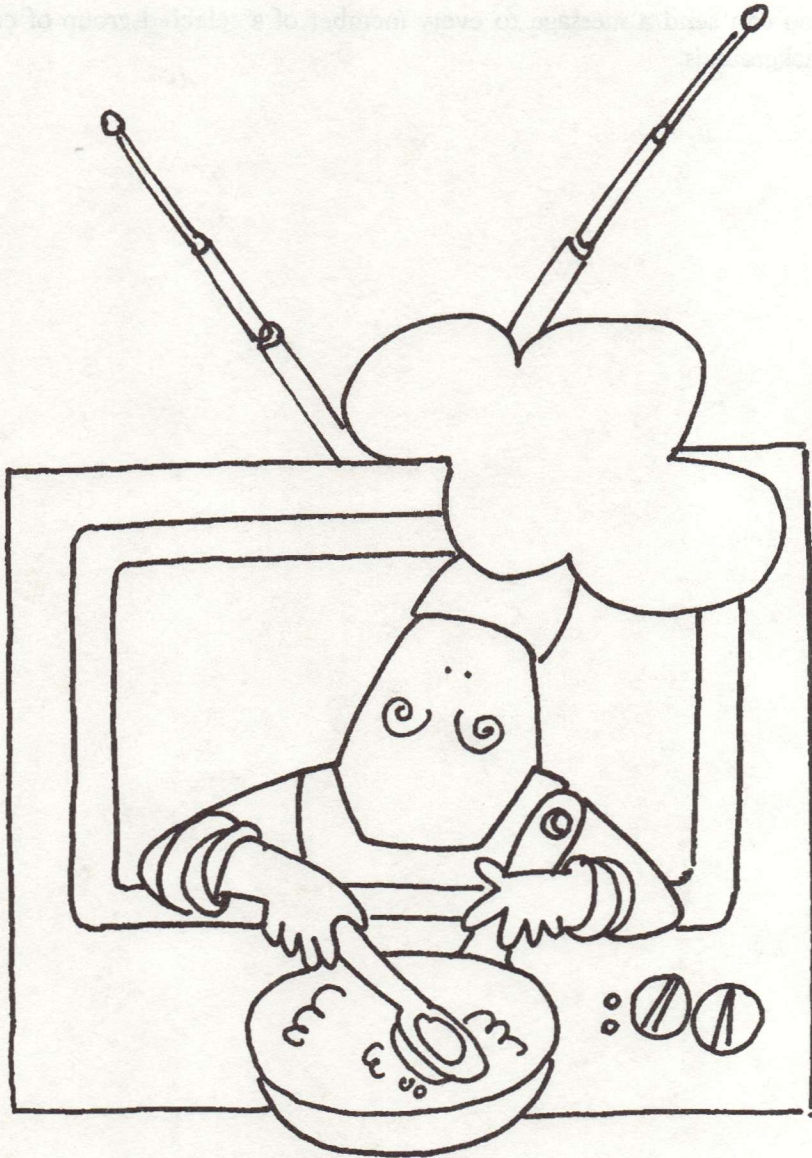
Because they're so similar (exactly the same, even), it would be pretty easy to roll them together into the same script. This would require some work to make it practical, though, for a couple of reasons. First, remember that there are separate group selections of cards and backgrounds, so we'd have to roll the two selection handlers (`selectCard` and `selectBkgnd`) together, too. Second, the unified script would have to be called `doToEachSelectedCardOrBkgnd`, which is really stupid and takes up too much disk space. You're invited to tackle these problems and work on combining the scripts if you're interested in doing so.

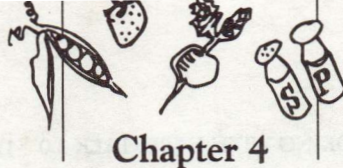
Summary

- This chapter contains handlers that help you work with groups of objects all at once.
- There are two command scripts that let you select groups of parts (fields and buttons) in order to work with them.
- You can also select groups of cards or groups of backgrounds.
- A selected group of parts can be deleted, pasted, aligned, or rearranged in various ways.

- You can send a message to every member of a selected group of cards or backgrounds.







Chapter 4

BROADCASTERS

At the end of Chapter 3, we looked at a couple of handlers (`doToSelectedCards` and `doToSelectedBkgnds`) that enabled you to send a single message to a bunch of objects at once. In this chapter, we generalize that technique and use it to send messages to groups of related objects.

Because these handlers are involved in sending out messages to many receivers, we decided to call them broadcasters. You use these handlers when you need to perform some action on all the cards in a stack, all the fields in a background, and so on. These handlers are different from the “`doToSelected`” handlers in Chapter 3 in that you don’t have to create a group selection first. Instead, they operate on all the objects within a particular range.

Do Something to Each Card in a Stack

We’ve discussed examples of selecting cards, then performing an action on each card in the selection by using `doToSelectedCards`. Sometimes you’ll want to act on each card in a stack. When you do, you can use this handler instead of having to select all the cards first.

```
1. on doToEachCard whatToDo
2.   -- script 4-1: Do something to each card in a stack.
3.
4.
5.   -- This handler repeatedly executes a given
6.   -- statement for each card in the current stack.
7.
8.   repeat with i = 1 to the number of cards
9.     go to card i
10.    send whatToDo to this card
11.  end repeat
12. end doToEachCard
```


Handler name:

doToEachCard

What it does:

This handler goes to each card in the current stack. When it gets there, it sends the card a message passed along by the user. Then, when it's done doing things, it doesn't do anything else.

Parameters:

Like the other “doTo” handlers, this guy takes one parameter, which is the text of the message that you want every card to get. If you don't pass anything for this parameter, every card will do exactly what you passed it—nothing. Although the handler will zip through the stack showing you each card, you can accomplish the same thing more quickly and with a lot less typing with “show all cards.”

Script tips: Domain street

Notice that we don't have to say “number of cards of this stack.” The “of this stack” part is implied. By default, HyperTalk's “number” function returns the number of things in the object's domain; so if you're asking about cards or backgrounds, it's how many there are in a stack. For background parts, the background is the domain, and for card parts, the card is the domain. You can override this default for cards by adding an “in background” expression to the end, as in “number of cards in background 4.”

How it works:

This handler is pretty straightforward. Although it doesn't have a group selection to act on, as the “doTo” scripts in Chapter 3 did, it uses a similar technique. It sets up a repeat loop that takes us to each card, tells each card what it's supposed to do, then moves on.

The repeat loop starts in line 8. It's one of those marvelous HyperTalk statements that reads pretty much like English, and will execute once for each card in the stack.

In line 9, we move to a new card, and in line 10 the desired message is passed along to the card. And as Porky Pig said, th-th-th-th-th-th-that's all folks! (Trivia question: name the two songs that were used as the themes for Warner Brothers classic cartoon series, Loony Tunes and Merry Melodies. The answer is at the end of this chapter.)

Do Something to Each Background in a Stack

Our next little broadcaster allows us to perform an action on every background in a stack. Once again, this handler allows you to do something to each background in a stack without having to group them into a selection first.

```

1.  on doToEachBkgnd whatToDo
2.      -- script 4-2: Do something to each background in a
3.      -- stack.
4.
5.      -- This handler repeatedly executes a given
6.      -- statement for each background in the current
7.      -- stack.
8.      repeat with i = 1 to the number of bkgnds
9.          go to bkgnd i
10.         send whatToDo to this bkgnd
11.     end repeat
12. end doToEachBkgnd

```

Handler name:

doToEachBkgnd

What it does:

Like the preceding handler, this script automatically loops through all the backgrounds in the current stack and sends them a message that you specify. In reality, this handler is probably less practical than doToEachCard; at least, we couldn't think of any real nifty examples to tell you about. Anyway, we'd hate to leave you hanging in case you really needed it someday, so it's presented here.

Parameters:

DoToEachBkgnd takes one parameter—the message sent to each background as the handler tours through your stack. If you don't pass anything for the parameter, you'll get a quick look at a card of each background in the stack, but nothing will happen to them.

How it works:

As a matter of fact, it works exactly like doToEachCard. Where doToEachCard specifies cards, this handler refers to backgrounds. In line 8, the repeat loop is set to execute once for each background. In line 9, we jump to the current background as specified by the loop variable. Line 10 sends the appropriate message. Lines 11 and 12 tell HyperTalk that we really do know what we're doing and make the script editor's indentation line up nicely.

You may not have realized it, but you can use a command like "go to bkgnd i" to go directly to a card belonging to a specified background. You can use this syntax in the go statement, and we do it in this script. HyperCard takes you to the first card it finds in the specified background, because you can't really be sure which card you're getting when you say "go bkgnd i." But knowing it's in the specified background, is good enough to serve our needs in this handler.

Do Something to Each Field in a Card or Background

The next in our series of broadcaster handlers is one that lets you blast a message out to all the fields that belong to the current card or background, or both, if you want to reach a really large audience. It uses a couple of interesting techniques involving parameter handling, but it's mostly very straightforward and similar to the other broadcaster scripts.

```

1.  on doToEachField whatToDo,options
2.      -- script 4-3: Do something to each field in a card
3.      -- or background.
4.
5.      -- This handler repeatedly executes a given
6.      -- statement for each field in the current card or
7.      -- background.
8.      if options is not "cardOnly" then
9.          repeat with i = 1 to the number of fields

```



```

10.         send whatToDo to field i
11.     end repeat
12. end if
13. if options is not "bkgndOnly" then
14.     repeat with i = 1 to the number of card fields
15.         send whatToDo to card field i
16.     end repeat
17. end if
18. end doToEachField

```

Handler name:

doToEachField

What it does:

This handler lets you perform some action on each field in the current card, current background, or both, by sending a message of your choice along to them. This is useful if you want to make sure that all of a background's fields have the same font and style, if you want to make them all the same size, or if you want to quickly put the same value into all of them.

This handler can do a lot of the same things as some of the group action scripts (alignParts, offsetParts), but it's more general and it doesn't require you to select the fields first. You can refer to the field currently receiving the message by using HyperTalk's target function, as in this example:

```
doToEachField "set textStyle of the target to bold"
```

As doToEachField loops through all its fields, the target will always refer to the one being operated on.

Parameters:

DoToEachField takes two parameters. The first one is required and the second one is optional. The first parameter contains the message that's going to be sent to every field. The second parameter is used to tell the handler whether you're interested in reaching all the card fields, the background fields, or just every darn field in sight (both background and card fields, that is).

There are two possible values for the second parameter, named "options" in the script: cardOnly means it will work on card fields, and bkgndOnly will get

the message sent to just background fields. If you don't pass any value at all for the second parameter, it will work on both card and background fields.

How it works:

This handler consists of two repeat loops, one each for card and background fields, with an appropriate if statement to make sure that the right stuff happens. Line 8 begins by checking the options parameter to see if the caller only wanted card fields touched. If the parameter is not "cardOnly," it means the caller wanted background fields, at least, to get it. Line 9 then starts a repeat loop that will send the message to each background field, as you can see in line 10. Note that lines 9 and 10 simply refer to "field," and background fields are assumed, by default.

After the end of the first repeat loop and if statement, we get set to do it all again, this time for card fields. First, line 13 checks to see if user told us not to deal with card fields by specifying "bkgnOnly." If card fields weren't excluded, we crank up another repeat loop that's going to send the message to the card fields. Once again, we use HyperTalk's "number" function to know how many card fields there are for us to loop through.

What happens if we ask this handler to send to all card or background fields, and there aren't any card or background fields? You might think we'd see that dreaded HyperTalk put-down, "Never heard of card field number 1." But no, the repeat statement saves us from this indignity and ridicule. Let's say there aren't any card fields. So, line 14 would evaluate to something like this:

```
repeat with i = 1 to 0
```

since "the number of card fields" is zero. What does HyperTalk do when it's confronted with an impossibility like a repeat loop that's supposed to count up from one to zero? It does the smart thing: it skips the loop entirely. This neatly prevents any error messages when there aren't any fields of one flavor or the other.

Do Something to Each Button in a Card or Background

Now that we've presented the script that takes care of each field in a card or background, there has to be an equivalent one for buttons, and this is it. It uses the same techniques as the field handler, just to show you again the wonderful

Nonsense

If you read through this handler, you may notice something that seems a little backwards. Line 8 checks to see if

```
options is not "cardOnly"
```

That is, it checks for the negative of a condition. The same technique is applied in line 13 when we check to see if the parameter is not "bkgndOnly." Doesn't this seem backwards? Shouldn't we check for the variables being equal to the desired value, then take the appropriate action? Well, let's try it.

When do we want card fields to receive the message? When options is set to "cardOnly" or to empty. When should background fields be included? When options is "bkgndOnly" or empty. So, we could have written the if statements like this:

```
if options is "cardOnly" or options is empty
then -- cd fields
if options is "bkgndOnly" or options is empty
then -- bg fields
```

This style is slightly more wordy than the one we used, but it is a little more obvious in showing what's going on. It has one other important difference: it requires that the specific words "cardOnly" or "bkgndOnly" be used, and it also assigns a special meaning to options being empty. The original scheme just uses "cardOnly" to prevent background fields from getting the message, and vice versa. In the end, it's your choice, and you probably can implement it either way. After all, that's why they call it a personal computer.

orthogonal flexibility of HyperTalk. And isn't that just one of the reasons why we all like playing with it so much?

1. on doToEachButton whatToDo,options
2. -- script 4-4: Do something to each button in a card
3. -- or background.
- 4.


```

5.    -- This handler repeatedly executes a given
6.    -- statement for each button in the current card or
7.    -- background.
8.    if options is not "bkgndOnly" then
9.        repeat with i = 1 to the number of buttons
10.         send whatToDo to button i
11.        end repeat
12.    end if
13.    if options is not "cardOnly" then
14.        repeat with i = 1 to the number of bkgnd buttons
15.         send whatToDo to bkgnd button i
16.        end repeat
17.    end if
18. end doToEachButton

```

Handler name:

doToEachButton

What it does:

This handler sends a message you specify to each button on the current card. You can specify only card buttons, only background buttons, or both. This is useful for clearing out the names of all the buttons on a card, for example, or giving them all the same script. As in doToEachField, you can refer to the current button by using “the target.”

Parameters:

There are two parameters, the same as in doToEachField. The first one is the message to be sent to each button, and the second parameter allows you to specify card buttons only (“cardOnly”) or background buttons only (“bkgndOnly”). Leave the second parameter blank to get both card and background buttons.

How it works:

This handler is identical, line for line, to doToEachField, except that every occurrence of the word “field” is replaced by “button.” Briefly, for those who hate to look back, there are two repeat loops, one each for card and background

button, and an if statement that tests the second parameter, called options, to make sure the right loops are executed.

Do Something to Each Card in a Background

This handler fine-tunes the idea of sending a message to a bunch of cards. It automatically restricts the message to just the cards in the current background—kind of like a private neighborhood party for the nearby cards.

```

1.  on doToEachCardOfBkgnd whatToDo
2.    -- script 4-5: Do something to each card in the
3.    -- current background.
4.
5.    -- This handler repeatedly executes a given
6.    -- statement for each card in the current
7.    -- background.
8.    put the number of this bkgnd into bgNum
9.    repeat with i = 1 to the number of cards of bg bgNum
10.      go to card i of bg bgNum
11.      send whatToDo to this card
12.    end repeat
13. end doToEachCardOfBkgnd

```

Handler name:

doToEachCardOfBkgnd

What it does:

DoToEachCardOfBkgnd, which is possibly the longest name for any handler in this or any other book, broadcasts your message to all the cards in the current background. It does not pass into the next background. It does not collect \$200.

Parameters:

This handler takes a single parameter, which is the message that you want to broadcast to all the cards in the current background. Once again, you can use “the target” in the message to refer to the current recipient of the message. If you pass an empty parameter, you’ll get a quick look at all the cards in the current background and nothing will happen to them.

How it works:

This handler works a lot like `doToEachCard`. In `doToEachCard`, we were able to take advantage of the fact that HyperCard provides a positional number for each card in a stack. In other words, we can say “go to card 12” and we’ll get to the twelfth card in the stack. Because of HyperTalk’s nifty flexible syntax, we can say something like “go to card 4 of this bgnd.”

In `doToEachCard`, we set up the repeat loop that sends the message to go from “1 to the number of cards,” which means the number of cards in the stack. Then, within the loop, a “go to card i” statement takes us to the right card.

This time, before we start, we save the current background number into a variable called `bgNum`. Then, we set up the repeat loop in line 9 to execute once for each card in this background.

Once again, we use the variable “i” to hold the number of the current card, except now it refers to the card in this background. Line 10 takes us to the current card to prepare for the message that’s going to be sent.

The rest of the script behaves in the broadcaster tradition we’ve established with the preceding handlers. The heart of the handler is in line 11, when the message actually gets shipped to its loyal audience. Again, you can use “the target” to refer to the object getting the message.

What about me?

We’ve said that we can refer to the object getting the message by using the function “the target.” What happens if we try to use HyperCard’s special object, “me,” referring to the object containing the current handler? There’s a subtle difference here that will make things work all wrong. The object we’re interested in is the one receiving the message, or “the target.” The object containing the current handler, “me,” is the place where the current handler is executing—in the stack script of the command library. So “me” refers to the command library stack.

Watch out though, because this can be disastrous if you attempt to change a property by using “me” instead of “the target.” For example, let’s say you want to empty out all the card scripts in a stack. If you’re not careful, you might say


```
doToEachCard "set the script of me to empty"
```

Remember that “me” will refer to “command library.” This command will loop once for each card in the current stack, and it will send this message to every card:

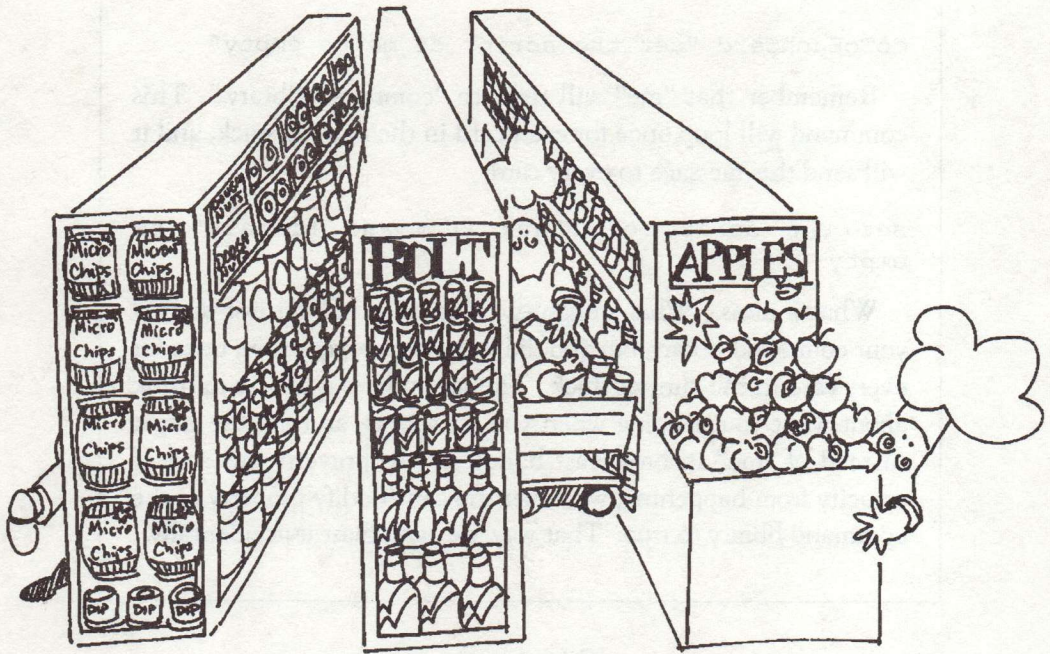
```
set the script of stack "Command Library" to empty
```

What a mess. What indignity. Not only did you just destroy your command library, but you did it over and over again, once for every card in the current stack. The lesson here is always to think about what you’re doing when you set scripts, and use the target instead of “me” in broadcast handlers. To prevent this sort of atrocity from happening, we’ve set the `cantModify` property of the command library to true. That way, we can’t blast it accidentally.

Summary

- In this chapter, we introduced commands to let us conveniently send a message to a group of related objects. These commands are called broadcasters.
- With the handlers in this chapter, we can send a message to every card in a stack, every background in a stack, every field on a card, every button on a card, or every card in the current background.
- This is a short chapter, and so it has a short summary.

Answer to trivia question: The songs used as themes for the Warner Brothers “Loony Tunes” and “Merry Melodies” cartoons were “Merrily We Roll Along” and “The Merry-Go-Round Broke Down.” A healthy dose of Daffy, Porky, Bugs, Road Runner, and the Tasmanian Devil can only improve your stacks. Meep-meep!





Chapter 5

GETTING AROUND

Isn't hypertext fun? The information is non-linear, you can go anywhere you want from anywhere else, and you can get totally and completely lost in a stack before you know what's going on. The well-prepared stack browser always has tools at hand to fight through the jungle of information. This chapter presents some of these tools and shows how you can use them.

In addition, you'll find some other interesting things to help make getting around more interesting. This includes some fun with HyperCard's visual effects, a convenience system for your windoids, and tools to help rearrange cards.

Card Tricks

This section has commands to use for fooling around with the location and order of cards. We have scripts to change the first card in a stack, to make copies of a card in the same and different stacks, and other fun related things. You'll find some of the scripts in this section are just strings of statements normally executed by hand, one at a time. These kinds of tasks make for great command scripts, and you'll probably be inspired to write a few of your own.

Copy a card from one stack to another

When you're together with friends, happily exchanging cards between various stacks (be careful of viruses), you'll often find yourself copying a card, then going to another stack and pasting the card in. This handler gives you an easy one-command way to copy a card without having to take the time to perform separate copy and paste operations.

1. on copyTo where
2. -- script 5-1: Copy a card from one stack to another.
- 3.
- 4.


```
5.    -- This handler provides a quick and easy way to
6.    -- copy a card between stacks.
7.
8.    doMenu "Copy Card"
9.    push card
10.   set lockMessages to true
11.   go to last card of stack where
12.   doMenu "Paste Card"
13.   pop card
14. end copyTo
```

Handler name:

copyTo

What it does:

This handler makes a copy of the current card in another stack. It doesn't affect the original card or stack in any way. When the handler is done, you'll be back at the original card in the original stack, just as if it had all been a dream. But you'll know it was real.

Parameters:

CopyTo takes one parameter, which specifies the stack where the copy of the current card will be placed. This parameter, named "where" in the script, is required.

How it works:

This handler strings together a few steps you'd normally have to do by hand. Line 8 copies the current card by using the Edit menu. When we're done pasting the copy of the current card into the new stack, we'll want to return to where we started, so we use the "push card" in line 9 as a way of saying to HyperCard, "Keep my place, I'll be back in a minute." Line 10 turns off messages in case the stack uses openStack to push the recent card, as many stacks do.

After we've saved our location, line 11 takes us to the end of the destination

stack. We've chosen to paste the new card at the end of the destination stack because that's the most obvious and conspicuous place for it. You can certainly change this if you want. Line 12 pastes a copy of the card into the new stack.

After we've pasted in the copied card, we're ready to go back to the card and stack where we started. Since we were smart enough to push the card before we left, we can now simply "pop card" in line 13 to return, and then end the handler.

And in the end

Whenever you paste a card, it goes after the one that you're currently looking at. This means that if you want a newly-pasted card to be the very last card in the stack, as in this handler, you should go to the last card in the stack and then paste. HyperCard will automatically make the newly pasted card the last one in the stack.

This handler is an example of how you can automate a fairly simple, yet tedious operation with a fairly simple, yet slightly funky script. If you want to try writing some command scripts of your own, this is the kind of handler you should probably be writing first. Find something you do all the time, then automate it by writing a script.

Move a card to another stack

Here's another one of those "macro" commands that strings together a few commands to let you perform some common operation quickly and painlessly. In this case, we provide a fast way to remove the current card and paste it into another stack.

```
1.  on moveTo where
2.    -- script 5-2:  Move a card to another stack.
3.
4.
5.    -- This command provides a quick way to move the
6.    -- current card into another stack.
7.
```



```

8.    doMenu "Cut Card"
9.    push card
10.   set lockMessages to true
11.   go to last card of stack where
12.   doMenu "Paste Card"
13.   pop card
14.   end moveTo

```

Handler name:

moveTo

What it does:

MoveTo cuts the current card out of the current stack, then goes to another stack, which you've specified, and pastes the card into that stack.

Parameters:

This script takes one parameter, named where. The parameter tells HyperTalk the name of the stack where you want the card to go. If you don't pass anything for this parameter, the script won't work right, and don't say we didn't warn you.

How it works:

This script works exactly like copyTo, except that it removes the original card from the first stack, rather than just making a copy of it. The script is also absolutely identical, except in line 8. In copyTo, line 8 copies the card; line 8 of moveTo cuts the card. That's the only difference.

Duplicate a card

This handler implements a single command to quickly make a duplicate copy of the current card and place it right next to the card in the same stack. It's an even simpler example of a shortcut script that takes a common operation involving a few steps and reduces it to a single command.

```

1.   on dupCard
2.     -- script 5-3: Duplicate a card.
3.

```



```

4.
5.    -- This handler provides a quick and easy way to
6.    -- duplicate a card in a stack.
7.
8.    doMenu "Copy Card"
9.    doMenu "Paste Card"
10. end dupCard

```

Handler name:

dupCard

What it does:

This handler makes a duplicate copy of the current card. The duplicate is put into the same stack as the original, and it's positioned right after the original card. Of course, because they're identical, you might think that the new one comes before the original, but as the description of copyTo points out, HyperCard always pastes the new card after the current card, not before it.

Parameters:

None.

How it works:

Simply. If you think you're not understanding anything about HyperTalk, just examine this script. Line 8 copies the current card by using the Edit menu. Line 9 pastes the copy of the current card. That's it.

Make a new card the first card of a stack

Whenever you create a new card, HyperCard inserts it after the current one. This is usually what you want, since it doesn't disturb the number of the current card, but there's at least one time when you'd like to insert the new card in front of the current card: when you want to make a new card appear first in the stack. This handler makes it easy to do that.

```

1.  on newFirst
2.    -- script 5-4: Make a new card be the first card of
3.    -- a stack.

```



```

4.
5.    -- This handler provides a quick and easy way to
6.    -- make a new card be the first card in a stack.
7.
8.    go to card 1
9.    doMenu "New Card"
10.   go to card 1
11.   doMenu "Cut Card"
12.   doMenu "Paste Card"
13.   go to card 1
14. end newFirst

```

Handler name:

newFirst

What it does:

This handler creates a new card and makes it the first one in the stack. It doesn't disturb the order of the other cards, other than to push them all back one position as the new one is inserted at the front.

Parameters:

None.

How it works:

If we were working with a different application than HyperCard, we might call this handler a macro. NewFirst doesn't do anything you couldn't do manually by repeating a series of steps. It just records those steps and you only have to remember the name of the command instead of the steps.

In line 8, newFirst goes to the head of the stack, and then creates a new card in line 9. As we've already mentioned, this new card will be inserted following the current card; in this case, it'll be the second card in the stack. Next, we return to the first card (line 10) and cut it (line 11). Now, we've accomplished most of what we wanted. The new card we created, momentarily the second one in the stack, advances to the number one position with the demise of the

old first card in line 11. Since we just deleted that card, we're now looking at the new first card, the one that was created in line 9.

There's only one detail left to worry about—putting back the old number one card that we cut. The new first card is now current. So, we simply have to paste the old first card and it will be inserted after the new first card, just where we wanted. The “paste card” happens in line 12. Finally, just to make it neat, line 13 returns to the new first card, since the paste in line 12 moved us to the second card, and that's it. Figure 5-1 gives you a picture of the song-and-dance that newFirst goes through.

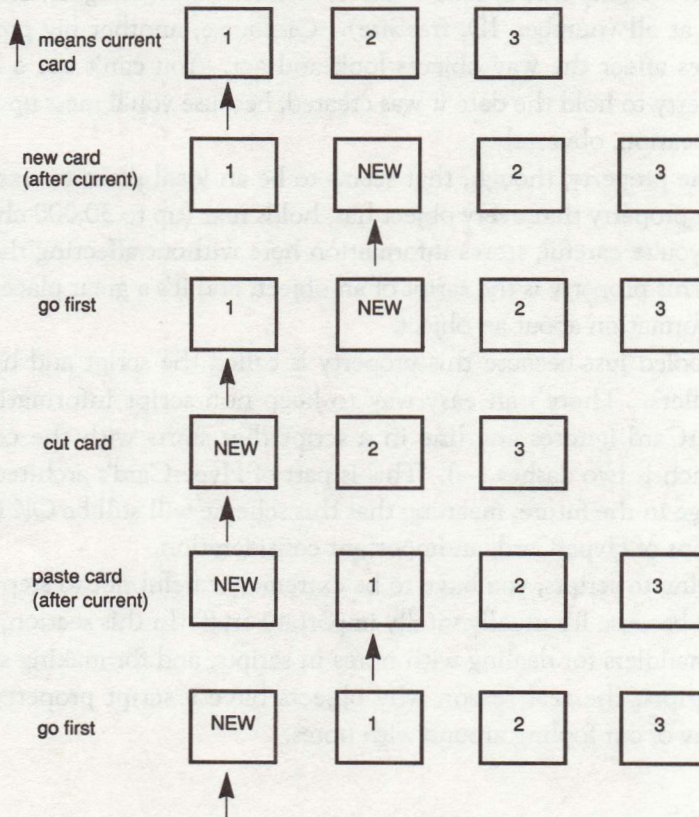


Figure 5-1 newFirst

Script Notes

If you want to put information into a stack and expect to find it when you look at the stack next time, you have to find a permanent home for the information. The most common place to keep data around is in fields, of course. When you put some text into a field, you can expect it to be there the next time you look (the Good Lord willing and the disk don't crash). Although fields are the usual places to hold information, there are actually several spots you can use for this purpose.

Any property of an object will stick around with the object, but most properties don't give you enough room to store much information. There are some properties that are only true or false (`visible`, `lockText`, `wideMargins`) and others you can't set at all (`number`, `ID`, `freeSize`). Of course, another big problem is that properties affect the way objects look and act. You can't use a button's location property to hold the date it was created, because you'll mess up the button's actual location, obviously.

There is one property, though, that seems to be an ideal place to stash information. It's a property that every object has, holds text (up to 30,000 characters of it), and if you're careful, stores information here without affecting the object in any way. This property is the script of an object, and it's a great place to keep additional information about an object.

Don't be fooled just because this property is called the script and holds the object's handlers. There's an easy way to keep non-script information in a script. HyperCard ignores any line in a script that starts with the comment delimiter, which is two dashes (--). This is part of HyperCard's architecture, so it won't change in the future, meaning that this scheme will still be OK in forthcoming versions of HyperCard, an important consideration.

When writing to scripts, you have to be extremely careful not to step on stuff already there, because it's usually vitally important stuff. In this section, we present a set of handlers for dealing with notes in scripts, and for making sure that the actual scripts, the real reason why objects have a script property, aren't affected by any of our fooling around with notes.

Handle with care

As you probably know, HyperCard will ignore any part of a script that doesn't fall between the pair of statements that begin and end a handler, either "on handler...end handler" or "function handler...end handler." However, it's a good idea to stay away from using these stray lines to keep information in a script. In the future, it's conceivable that HyperCard may be extended to include new keywords to define a handler (not just "on" and "function"), and you may be in big trouble if you happen to have a line that starts with the new keyword. If you start a line with --, you'll be immune to HyperCard changes.

Retrieve a note from a script

The first couple of handlers we present in this section provide the basic capabilities for our note system. These are the ones that put notes into scripts (very carefully), and take them out.

```

1.  function getNote noteName,scriptText,returnOffsets
2.      -- script 5-5:  Notes: retrieve a note from a script.
3.
4.
5.      -- This command returns the text of a note imbedded
6.      -- in a script.
7.
8.      get offset("-- NOTE: " & noteName,scriptText)
9.      put it into startChar
10.     delete char 1 to startChar of scriptText
11.     put line 1 of scriptText into scriptText
12.     if startChar is zero
13.     then put zero into endChar
14.     else put startChar+length(scriptText) into endChar
15.     if returnOffsets is true
16.     then return startChar & "," & endChar+1
17.     else return item 2 of scriptText
18. end getNote

```


Handler name:

getNote

What it does:

GetNote is a function that returns the text of a note imbedded in a script. Optionally, getNote can return the offset position of the note instead of the note's actual text.

A note in our note system consists of the comment delimiter "--," a space, the word "NOTE" and a colon, another space, followed by the note's name, a comma, and the note's text. It looks like this:

```
-- NOTE: note name, note text
```

Parameters:

This handler takes three parameters, with the third one optional. The first parameter contains the name of the note. In our note system, each note has a name which identifies it from the rabble of other notes that may be hanging around in the same script. This name is passed as the noteName parameter. The second parameter, called scriptText, is the text of the script with the note in it. This parameter is usually a reference to a script rather than a variable or actual text.

The third parameter tells whether the caller is interested in receiving the complete text of the note, or just the offset of the note (starting and ending location) in the script. This parameter is named returnOffsets, and the handler simply checks to see if its value is "true." If so, the offsets are returned instead of the note's text.

How it works:

The job of getNote is to track down this string of characters. That's easy enough. HyperTalk can find it with one statement (line 8) by using the offset function. The value returned is the start of the note line, and it's put into the startChar variable (line 9).

In line 10, we start to manipulate the parameter containing the script with the note in it. Our object is to extract the note from the script and ultimately to delete everything from the script except the note itself. Remember that the

parameter is just the text of the original script, so we can mess with it however we want, without harming the precious original.

Line 10 begins by deleting all the text before the starting character of the note. If you look carefully at this line, you'll see that it goes a little too far. Not only does it delete all the characters before the note begins, but it also deletes the first character of the note itself. That's because when you tell HyperTalk to

```
delete char 1 to startChar of scriptText
```

the deletion is inclusive; that is, it includes both ends of the range, character 1 and character startChar, as well as everything in between. This will effectively lop off the first character of the comment line. In this case, we really don't care about that, for a couple of reasons. To start with, we know that the first character of a note line will always be a hyphen anyway. Second, the only part of the note we're really interested in is at the very end, after the "-- NOTE: noteName" business. If we wanted to make sure the first character didn't get chopped off, we could simply change line 10 to read

```
delete char 1 to startChar - 1 of scriptText
```

and HyperTalk would back up one character at the end of the deletion, preserving the first character of the note line. We'll just leave things as they are, sacrificing that unneeded first hyphen to the fanatical cause of making the script just a little simpler.

We've now reduced scriptText to just the note line and the remainder of the script. We can easily get rid of the rest of the script with a statement like the one in line 11. The scriptText variable now holds only the note line, with its initial character (a hyphen) victim to our first deletion, as discussed above.

What if we didn't find the note we were looking for? Lines 12 and 13 handle that by making sure both startChar and endChar contain zero, meaning no text will be returned. If startChar isn't zero, the note was found somewhere, and line 14 puts the proper value into endChar.

The last thing to do is to check to see if the caller asked for the entire text, or just the offsets. Line 15 does the checking, and line 16 is executed if the caller just wanted offsets returned. Otherwise, line 17 takes care of returning the text of the note. Notice that line 17 returns just "item 2 of scriptText," which is the note itself. The "-- NOTE:" header and the name of the note aren't returned.

The work of this handler is something like the `getHandler` function in Chapter 2, and the script is similar, but not exactly the same. You might want to take a look there to examine the differences, if you're looking for something to do.

Put a note into a script

This handler is the one that sticks the notes you specify into the script of an object. It has to be careful not to destroy any of the good stuff already there in the stack, and it has to do a few other tricks, as well.

```

1.  function putNote noteName,noteContents,scriptText
2.    -- script 5-6: Notes: put a note into a script.
3.
4.
5.    -- This command imbeds a note into a script,
6.    -- replacing the old note of the same name if
7.    -- there is one.
8.    get getNote(noteName,scriptText,true)
9.    put item 1 of it into startChar
10.   put item 2 of it into endChar
11.   get "-- NOTE: " & noteName & "," & noteContents &
      return
12.   put it into char startChar to endChar of scriptText
13.   return scriptText
14. end putNote

```

Handler name:

`putNote`

What it does:

This function installs a new note into a script. If there is already a note in the script with the same name as this one, the old one is replaced. If there's no old note with this name, the new note is inserted at the top of the script, where it's nice and easy to see.

Parameters:

The `putNote` function takes three parameters. The first one, `noteName`, is really descriptively titled, because it contains the name that will be used for the new note. The second parameter is also cleverly named `noteContents`, and it holds the text you'd like to put into the new note. The third parameter, `scriptText`, is the script you want to manipulate. As usual, this parameter will normally be an expression that refers to a script, like "script of this stack."

How it works:

It might not be obvious that `putNote` would have to call `getNote`, but it does. Consider this: `putNote` must check to see if there's already a note in the script with the same name as the new one. The way to check this is with a call to `getNote`, taken care of in line 8. After line 8, the offsets of the old note (if there was one) are stored in the variable "it." If `getNote` couldn't find an old note with the same name, it returns 0,0 (check the `getNote` description above to see how this happens).

After we get the offsets, lines 9 and 10 put them into `startChar` and `endChar`. Then, line 11 assembles the complete note line together for the first time by adding together all the components. Some of the parts of the note line come from parameters, like `noteName` and `noteContents`, while others are part of the standard format we've set up for notes, like "-- NOTE: ." There's a return added onto the end of the line to make sure the note occupies its own line and doesn't mess up any real script lines or other notes that might be present.

Now the note is assembled, and we're ready to put it in place. Line 12 inserts it into the offsets we got earlier from calling `getNote`. If `getNote` found an old version of the note, this will replace the old note with the new one. If `getNote` didn't find anything, it returned 0 for both `startChar` and `endChar`, which means the new note will be inserted at the start of the script, before anything else that might be there.

After line 12 does its work, `scriptText` contains the text of the script with the new note inserted. Remember, `putNote` is a function designed to return the text of the new script as its result. Line 13 takes care of that, and that's it. The caller of `putNote` has to be responsible for putting `scriptText` into the right place. We'll see an example of that in the automatic bookmark script, coming up next on this same channel.

Automatically save your place in a stack

Whenever you open a stack with the “Open Stack” command, HyperCard shows you that stack’s first card. This is sometimes what you want, and some stacks should *always* be browsed from the first card. For lots of other stacks, however, you don’t necessarily want to start at the first card every time you open them. What you’d really like is for the stack to remember what card you were looking at when you left, and return you to that card when you come back.

There are various schemes for doing this trick in HyperCard. Most of the ideas involve pushing and popping cards. Unfortunately, the problem with this approach is that you don’t have direct control of the pushed cards, since executing scripts might do their own pushing. This can interfere with your own carefully pushed cards and wreck your system. We’ve approached the problem from an entirely different direction. We put a note into the stack script that acts like a bookmark (would you believe a stackmark?) and seems to work pretty well. You can use the Cookbook Reference stack to install this feature (see Appendix A for more on how to install optional command library features).

```

1.  on saveBookmark
2.      -- script 5-7:  Notes: automatically save your place
3.      -- in a stack.
4.
5.      -- When you leave a stack, these handlers will
6.      -- remember the card you were on and will return you
7.      -- there the next time you open the stack.
8.
9.      if cantModify of this stack then exit saveBookmark
10.     get putNote("bookMark",the number of this card,
11.               script of this stack)
12.     set script of this stack to it
13. end saveBookmark
14.
15. on goToBookmark
16.     if number of this card is not 1 then exit
17.     goToBookmark

```



```

17.    put getNote("bookMark",script of this stack) into
        cardNum
18.    go to card cardNum
19.
20.    end goToBookmark

```

Handler name:

closeStack, openStack

What it does:

The saveBookmark handler puts a bookmark into the stack's script whenever you leave a stack. GoToBookmark, as you can probably guess, finds the bookmark and returns to the marked card whenever the stack is opened again. If you install this feature with the Cookbook Reference stack, every stack you have will save its place and return there when you reopen the stack (except for stacks that catch closeStack and openStack and don't pass them, naughty naughty).

The bookmarks we'll create will look like this:

```
-- NOTE: bookMark,101
```

where the number at the end is the number of the card we were on when we left the stack.

Parameters:

None. You won't have to call these handlers directly, since they'll be called whenever there's a closeStack or openStack message.

How it works:

These handlers rely on getNote and putNote to do most of their dirty work. They're just a little bit of glue to make the right calls when stacks open or close. In the saveBookmark handler, line 9 checks to see if the stack is locked by testing its cantModify property. If cantModify is true, we can't put a bookmark into it, so we quit.

If the stack isn't locked, line 10 calls putNote to stick the bookmark onto the beginning of the stack script and line 11 puts the whole thing back into the

stack script. Line 12 then passes the message on so someone farther down can get it. That's all it does, and a bookmark note will appear in the stack's script after you leave the stack.

The goToBookmark handler in lines 15 through 20 gets the information that the saveBookmark handler squirreled away. Line 16 checks to see if you've gone directly to a specific card in the stack, rather than just opened it, as in a case where you used a command like

```
go card 29 of stack "American Dream"
```

meaning you probably really want to go to that card, and don't want the bookmark to whisk you off to another card.

If line 16 determines that we're not on the first card of the stack, line 17 then finds the bookmark and puts the card number into cardNum. Then, line 18 takes us to that card.

Also, notice that these handlers provide a good argument for using "pass" in handlers for system messages. If you faithfully pass each system message when you're done with it, you'll always be able to benefit from features in handlers farther down the inheritance chain. If you install the bookmark feature, it will work on any stack that allows the closeStack and openStack messages to pass through.

The best rule to use is this: if you handle a system message, pass it when you're done, unless there's a specific reason you want to prevent the rest of the inheritance chain from seeing it. If there's nothing you're trying to prevent, go ahead and pass it.

Notes to you

What happens when the bookmark handler goes looking for a note and it doesn't find it? Let's take a look. GoToBookmark calls getNote to find the note. If getNote doesn't find the string it's looking for, the attempt to delete the first part of the script (line 10 of the getNote script) doesn't delete anything, and line 11 reduces scriptText to just its first line. Then, line 17 returns "item 2 of scriptText."

This will have one of two results. If the first line of the script happened to have a comma with some text after it, the text after the comma will be returned, because it's item 2. If there was no comma in the first line, empty will be returned. The openStack handler tries to "go to card cardNum," where cardNum is the result returned by getNote. If getNote returned something because the first line of the script had a comma with text after it, like this handler declaration, for example:

```
on REM berry, buck, mills, stipe
```

the "go" will result in a statement like "go to card buck." This probably won't cause anything to happen, unless "berry" happens to refer to a real card. Otherwise, HyperCard won't go anywhere, and it won't generate an error message, either.

We said that cardNum could also be empty if there was no comma in the first line of the script. This would result in a statement that just says "go to card." There's no harm in that, either. Since "card" will be evaluated to mean the current card, "go to card" means go to the current card, which is pretty much like saying "spend some time doing nothing at all, please."

So, this all means that your bookmark openStack handler will probably work correctly when there's no bookmark present, but our system is a little fuzzy around the edges here. If you want an exercise in improving these scripts, you could work on tightening up the error checking for this case. Have fun!

Set and use manual bookmarks

The automatic bookmark system we just described shows you how to teach HyperCard about remembering your place in stacks. In this section, we'll take a slightly different approach that requires more effort, but gives you more control over where and how your bookmarks are used.

1. on pushMark -- goes with popMark
2. -- script 5-8: Set and use manual bookmarks.


```

3.
4.
5.    -- This handler remembers a bookmark that can be
6.    -- retrieved by the popMark handler.
7.
8.    global bookMarks
9.    put the long id of this card & return after bookMarks
10. end pushMark
11.
12. on popMark
13.    global bookMarks
14.    go to last line of bookMarks
15.    delete last line of bookMarks
16. end popMark

```

Handler names:

pushMark, popMark

What it does:

These handlers behave very much like HyperTalk's built-in push and pop commands, but they set up your own private little list of cards, so you won't have the problems mentioned above in the discussion under "Automatically save your place in a stack," because no other handlers can mess with your list. You call pushMark when you want to remember the current card. Then, when you want to return to that card later, popMark takes you back.

Like the built-in push and pop, these handlers create a last-in, first-out list of cards (in programs other than HyperCard, this is usually called a stack), so you can keep pushing cards with pushMark. PopMark will always go to cards in the reverse order that they were pushed (that is, most recently pushed will be popped first).

Parameters:

PushMark and popMark don't take any parameters.

How it works:

These handlers work by maintaining their own last-in, first-out list of cards that you control as you push and pop cards. The list is kept in a global variable called `bookMarks` used by both of these handlers. In `pushMark`, there's only one line with a HyperTalk command. That line, line 9, puts the long ID of the current card into a new line of the `bookMarks` global variable and sticks a carriage return after it to keep multiple bookmarks separate. The reason it uses the long ID of the card is because it includes the name of the stack, guaranteeing that HyperCard will be able to find the card from any other stack. That's all that `pushMark` does.

When you're ready to go back through your list of pushed cards, you'll use `popMark`. After declaring the global in line 13, `popMark` has two things to do. First, it whisks you off to the most recently pushed card (this happens in line 14), whose name can be found in the last line of `bookMarks`, right where `pushMark` added it. Then, after going there, it deletes that card from the list in line 15.

Daylight come

As usual, let's take a moment to explore the hairy edge of error handling in the bookmark script. What happens if you call `popMark` and the list of pushed cards (that is, the global variable `bookMarks`) is empty? This would cause line 14 to act like

`go to empty`

which HyperTalk really doesn't like, and it brings up an error alert.

If this seems like something you don't want, you could introduce a little check in `popMark` to see if `bookMarks` is empty, and if so, either take no action or put up an appropriately demeaning alert box.

Create random visual effects

If you're like most HyperCard users, you've probably thought that the visual effects you can make when you leave a card, like zooming or dissolving, are pretty neat. You may have played around with visual effects for a while, then started using them in a mature and responsible manner. Well, the script in this section is for those who can't get enough of flashy visual effects in their personal HyperCard world, and who just love to see things checkerboarding, barn-dooring, and generally behaving to excess.

```

1.  on randomVis visCount
2.    -- script 5-9: Create random visual effects.
3.
4.
5.    -- This handler chooses random visual effects that
6.    -- you can execute when you leave a card or stack.
7.
8.    if the shiftKey is up then exit randomVis
9.    global effects, speeds, images
10.   if visCount is empty then put random(3) into visCount
11.   if effects is empty then -- first time, initialize
      globals
12.   put "cut,zoom open,zoom close,dissolve,venetian
      blinds," & ␣
13.   "scroll left,scroll right,scroll up,scroll down,
      " & ␣
14.   "iris open,iris close,checkerboard," & ␣
15.   "wipe left,wipe right,wipe up,wipe down," & ␣
16.   "barn door open,barn door close," & ␣
17.   "shrink to top,shrink to bottom,shrink to center,
      " & ␣
18.   "stretch from top,stretch from bottom," & ␣
19.   "stretch from center" into effects
20.   put "slow,very slow,fast,very fast" into speeds
21.   put "card,black,white,gray,inverse" into images
22.   end if

```



```

23.     repeat visCount times
24.         put any item of effects into thisEffect
25.         put any item of speeds into thisSpeed
26.         put any item of images into thisImage
27.         visual effect thisEffect thisSpeed to thisImage
28.     end repeat
29. end randomVis

```

Handler name:

randomVis

What it does:

This handler generates random visual effects. It uses all of HyperCard's possible effects, including the speed variations and the "to <image>" options. This handler is only useful for adding fun to your life in HyperCard. It has no practical value.

Parameters:

RandomVis takes one parameter which makes it even more decadent than it would be otherwise. It's bad enough that randomVis creates a random visual effect. Its parameter, visCount, allows you to create *any number* of random visual effects and have them all executed, one after another. This can really boggle your mind, so use it with care.

How it works:

One of the handy automatic features that we've provided is the ability to install a call to randomVis in an arrowKey handler. If you choose this option, you'll get random visual effects every time you use an arrow key to go to another card. If you've got this fun feature installed, line 8 lets you override it by holding down the shift key as you press the arrow.

In line 10, randomVis checks to see if the caller passed a value for visCount, which is the number of visual effects to stack up. If no value was passed, randomVis will generate a random number of random effects, up to 3 (that's the random(3) call in line 10). You can change this number if it seems like too many for you to sit through.

RandomVis uses three global variables: effects, speeds, and images. These variables are used to hold HyperTalk's possible values for the three parts of the visual effect command, which is defined as

```
visual effect <effect name> <speed> to <image>
```

RandomVis checks to see if it has executed already, and if not, it loads up the three globals. The names of the effects, which are kept in the global variable "effects," tell what kind of basic effect will happen. This variable is set up in lines 12 through 19. The "speeds" global is initialized in line 20. Line 21 puts the possible fade-to values into the "images" global.

Line 23 begins a repeat loop that will take place once for each count of visCount, and each time through the loop will generate one visual effect. The actual creation of the visual effect comes in lines 24 through 27, through the use of HyperTalk's magic syntax "any item," which selects a random item from effects, speeds, and images, and builds them all together into a "visual" statement. HyperCard allows you to stack up a bunch of visual effects, which are then executed when you leave the current card.

This handler creates the visual effects and piles them up, but it's up to you to call the handler at the right time to make the screen dance. The easiest way to use randomVis is to install the random visual effect arrowKey handler. You could also put a call to randomVis into a handler for a button that's used to go to another stack, or you might place a call to randomVis into a closeStack handler. Just be careful with randomVis, as HyperCard only keeps track of about 10 of them anyway. Also, this health alert: VisCount values greater than 5 have been known to cause insanity and hair loss.

Don't hide menu bar if there's a big screen

On most Macintosh computers, standard-sized 9-inch HyperCard stack windows take up the whole screen. This includes the space normally reserved for the menu bar. In order to give you the full picture, some stacks hide the menu bar when they start up. On small screens, this ensures that the entire card will be visible (if it's standard-sized or smaller), including the part that would be hidden by the menu bar. However, for folks with a bigger-than-average screen or small stack windows, hiding the menu bar is a nuisance that probably shouldn't be done. The short handler in this section will make sure your menu bar doesn't go away unnecessarily.


```

1.  on bigScreenHide hideParams
2.      -- script 5-10: Don't hide the menu bar if
3.      -- big screen.
4.      -- This handler refuses to hide the menu bar if
5.      -- HyperCard is
6.      -- running on a screen with room for the whole card.
7.
8.      if "menuBar" is not in hideParams then exit
          bigScreenHide
9.      if the shiftKey is down then exit bigScreenHide
10.     if the screenRect is rect of card window
11.         then exit bigScreenHide
12.     exit to HyperCard
13. end bigScreenHide

```

Handler name:

bigScreenHide

What it does:

In this script, we're intercepting the built-in command named `hide`. `Hide` is used to accomplish several things in HyperCard: it can hide or show individual parts, windows, or the menu bar. This handler prevents HyperCard from hiding the menu bar if the screen is big enough to accommodate the full HyperCard window and the menu bar.

If we're absolutely sure we want to hide the menu bar, we can call "hide menubar" with the shift key held down, or hold down the shift key while typing command-space bar, HyperCard's command key for hiding the menu bar. Our script makes sure this is guaranteed to work no matter what the size of the screen.

Parameters:

This handler is called by a "hide" handler that's installed in the Cookbook Reference stack's script. It gets the parameters of that hide command passed in as its own parameters.

How it works:

There are various complicated formulas we could use to figure out if the screen is large enough to hold the entire HyperCard window and the menu bar. We can't hard-code the size of the menu bar, because its size can vary on systems for large Asian fonts or for visually impaired users. However, rather than getting really tricky, we can get a lot of mileage out of simply checking to see if the screen is different from the card window. This way is clean, fast, and easy, so we'll use it.

In the handler, line 8 checks to see if we're in the middle of a request to hide the menu bar, which is the only kind of "hide" we're interested in. If it's not, we don't want to touch it, so we exit the handler. Line 9 performs another check. If the user is holding down the shift key, we want normal hiding to take place, so again we give up on the message rather than handling it here.

Now that we've gotten rid of two of the cases we're not interested in, line 10 makes the final check to see if the screen's rectangle, returned by the HyperTalk function `screenRect`, is the same as the card window's rectangle. If they are, we've got a small screen, or a big screen and a big stack, and we don't want to prevent "hide" from doing its thing, so once again we exit the handler.

Finally, if the message that got this handler called can make it through all these tests, we get to do...nothing! The handler exits to HyperCard after the third "if" statement. This simply means that if the parameter is "menuBar," and the shiftKey is not down, and the screen's rectangle is not the same as the card window's rectangle, then nothing happens when you call hide. That's what we're after, all right.

How to handle radio buttons

The Macintosh interface uses radio buttons to allow a user to pick one of several options at a time. In programming jargon, we say these options are "mutually exclusive," meaning you only get to choose one of them, and if you pick another, your first choice turns off. The designers of the interface chose the term "radio buttons" because they behave like the buttons on an old-fashioned car radio: if you push one of them, your previous choice pops out (turns off).

Although radio buttons are a neat idea, support for programmers is not quite fully implemented. You can create and set radio buttons, but there's no way to indicate how they're grouped so that the on/off traffic can be handled automatically. HyperCard inherits this little deficiency from the Macintosh ROM, so

you have to do your own thing when it comes to working with groups of radio buttons. The script in this section will take care of radio buttons for you.

```

1.  on mouseUp
2.    -- script 5-11:  How to handle radio buttons.
3.
4.
5.    -- This mouseUp handler can go in the card script
6.    -- and will take care of radio buttons.
7.
8.    if "button" is not in the target then pass mouseUp
9.    put "radio 1,radio 2,radio 3,radio 4" into radioList
10.   put the short name of the target into hitButton
11.   if hitButton is not in radioList then pass mouseUp
12.   set hilite of the target to true
13.   repeat with i = 1 to the number of items in
       radioList
14.     get item i of radioList
15.     if it is hitButton then next repeat
16.     set hilite of button it to false
17.   end repeat
18. end mouseUp

```

Handler name:

mouseUp

What it does:

This mouseUp handler is designed to go into a card script. When you place it there, you can put a set of radio buttons onto the card and not put a mouseUp handler into them. For the script to work right, the buttons should have their autoHilite property set to true. This script in the card handler will take care of turning them on and off properly, and you can just sit back and watch.

Note that this handler is limited to taking care of just one set of radio buttons per card. An interesting exercise would be to extend it to handle more than one collection of radio buttons.

Parameters:

None. You won't have to call this handler, since it's called when you click in the radio buttons it's managing.

How it works:

This handler will live in the card script, and we might catch mouseUps from all over the place, so we have to make sure we're interested before going any farther. The best way to filter out the mouseUps we don't want is to see if a button was clicked. Line 8 checks for that, and if no button was clicked, we pass the message.

If the script makes it to line 9, we know that a button was hit, but we're not sure if it was one of the radio buttons we're interested in. To find out, we put the names of the radio buttons in our group into the radioList variable. You should use your own radio buttons' names instead of "radio 1, radio 2, radio 3, radio 4," and you can have any number of them. Line 10 gets the target's name into a variable, and line 11 checks to see if the clicked button was one we're interested in. If not, we shoot the message down the path.

By the time the handler reaches line 12, we're sure that the click is for us, and we're ready to do some real work. Setting the buttons' autoHilite property to true means the button will "track" the mouse correctly. That is, when you mouse down in a button with autoHilite turned on, the button will light up, and it will stay lit up until you release the button or move the mouse outside the button's rectangle. The autoHiliting ends when the user lets go of the button, so we need line 12 to make sure it will stay on.

When dealing with radio buttons, turning on the one clicked is only part of the battle. The rest involves turning off all the other buttons in the same group. That's what happens in lines 13 through 17. This repeat loop executes once for each radio button in the group. Line 14 grabs the button's name, line 15 skips over the button if it's the one we just turned on, and line 16 makes sure it's turned off if not. That's it.

As you can see, being polite doesn't take much extra effort at all, and it helps you be sure that you'll be invited back again and again.

Script tips: Saving and restoring global things, or Miss Manners's guide to stack etiquette

HyperTalk keeps track of lots of global properties and states, most of which can be changed from scripts. This tip is about putting things back the way you found them—see, it turns out that Mom was right again. When you write scripts that other people are going to use with their own stacks, it's nice to keep these principles in mind. We'll talk about various global settings in HyperCard and some ideas about how to treat each one.

The menu bar. As we've already discussed, it's a good idea to hide the menu bar only on Macintoshes with a small screen. You can use the strategy shown in script 5-10 to handle this. If you hide the menu bar, should you put it back when the stack is closed? Probably the best thing to do is to check to see if the menu bar is visible when the stack is opened, then restore it to that state on closing.

One problem with this approach is that there's no way to tell from HyperTalk if the menu bar is showing or not, although you can do it with an XCMD or XFCN. (To do this, call the toolbox function GetMBarHeight. If it returns zero, the menu bar is hidden.) If you hide the menu bar when a stack starts, it's also pretty reasonable to show it on your way out of a stack, since most stacks that need to show the screen space under the menu bar will just hide it again.

Palettes. These are the little windows, also known as windoids, that you get when you pull the Tools or Patterns menus down and drag them away from the menu bar. The rule here is this: if a user wants to have palettes around, you should leave them alone, rather than hiding them when your stack is opened. It's considered quite rude to mess around with palettes that the user has opened, so don't do it. Of course, if your script is designed to do something with palettes, like track them around the screen (see "Windoid tracking" in this chapter), the user is expecting palettes to move, and it's OK to change them around.

Message box. The best policy for the message box is to remember if it was shown or hidden at openStack time. Then do whatever you want with it, and restore its state, hidden or shown, at closeStack time. This is very easy to do from HyperTalk. Hiding the message box is useful for making sure the card isn't covered up on small-screen Macintoshes.

Another good etiquette point concerning the message box is to leave its location alone, rather than moving it somewhere else. Moving the message box is likely to confuse your user, so don't do it unless there's a specific reason for it, or if you really do want to confuse your user.

If you want to follow these tips on global settings, you can use these handlers to do the job:

```
on openStack
    global msgShowing
    put visible of the message box into msgShowing
    hide message box -- or show message box here
    if the rect of card window is the screenRect
        then hide menuBar -- if you draw on that part
            of the card
end openStack

on closeStack
    global msgShowing
    set visible of the message box to msgShowing
    show menuBar
end closeStack
```

The openStack handler saves the state of the message box before hiding it, then hides the menu bar if there's a small screen. Note, you can also force the message box to be shown here, if that's what you want for your stack. On the way out of the stack, the message box is returned to its saved state, and the the stack, the message box is returned to its saved state, and the menu bar is turned back on.

If you use these handlers, you'll find that going between two stacks will cause the menu bar to flicker as it keeps getting hidden and shown. If you're shipping a group of stacks and you don't want the menu bar to act crazy when you switch between them, you can set `lockMessages` to true when you jump between stacks. This will stop `closeStack` and `openStack` from being sent and the menu bar won't be touched. Of course, you can't get away with this if you do some other necessary work on `closeStack` or `openStack`.

Another idea is to check the visibility of the menu bar with an `XCMD` or `XFCN`, as mentioned earlier, by calling the toolbox function `GetMBarHeight`. If `GetMBarHeight` returns zero, the menu bar is hidden. If it's not zero, the menu bar is visible.

Automatically move windoids

HyperCard has a user interface innovation called floating windows, more popularly known as windoids. The message box, the tool and pattern palettes, and the linking box (pictured in Figure 5-2) are all windoids. It's handy to keep some floating windows around all the time, especially the tool and message windoids. The handlers in this section help keep your windoids in the right place.

Our windoid system works like this: whenever you open or move the tool, pattern, or message windoids, their new positions are remembered. Whenever you move the card window, the open windoids snap to new locations that are at the same relative location from the card window. So, for example, if you like to keep the message box centered just below the card window, these handlers will make sure it stays there, even when you move the card window.

What if you don't have a big monitor, and don't move the card window around? With a small screen and standard-sized (9-inch) stacks, you may not have even realized that the card is in a window and can be moved. But even if you never move your card window, you can still take advantage of the windoid tracking system's other feature. It remembers windoid positions and visibility even when you quit and restart HyperCard. So, you won't have to reposition your windoids every time you restart HyperCard. The computer, which has a

great memory, will remember these things for you. To make windoid tracking work, go to the “Automatically move windoids” card in the Cookbook Reference stack and click the “Enabled” button.

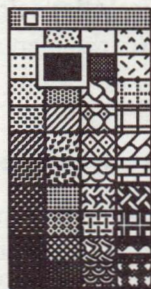
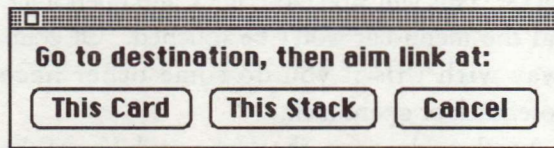


Figure 5-2 Windoids

```

1. on initWindoidTracking
2.   -- script 5-12: Automatically move windoids.
3.
4.
5.   -- The handlers in this script implement a system
6.   -- for automatically repositioning windoids around
7.   -- the card window.
8.   global windoidTracking,msgLoc,toolLoc,patLoc
9.   lock screen

```



```

10.  push card
11.  go to card "windoid preferences" of stack
      "Cookbook Reference"
12.  if the result is not empty then
13.      put "Can't find windoid tracking card."
14.      exit initWindoidTracking
15.  end if
16.
17.
18.  put "enabled" into windoidTracking
19.  put card field "cardLoc" into cardLoc
20.  put card field "msgLoc" into msgLoc
21.  put card field "toolLoc" into toolLoc
22.  put card field "patLoc" into patLoc
23.  set loc of card window to cardLoc
24.  set loc of message box to msgLoc
25.  set loc of tool window to toolLoc
26.  set loc of pattern window to patLoc
27.  set visible of message box to hilite of button "Show
      Message Box"
28.  set visible of tool window to hilite of button "Show
      Tool Window"
29.  set visible of pattern window to hilite of button
      "Show Pattern Window"
30.  pop card
31.  unlock screen
32.  end initWindoidTracking
33.
34.  on trackWindoids
35.      globalwindoidTracking,updateWindoids,cardLoc,msgLoc,
          toolLoc,patLoc
36.      if windoidTracking is not "enabled" then exit
          trackWindoids
37.      put false into updateWindoids
38.      if loc of card window is not cardLoc then
39.          -- make windoids track card window

```



```

40.      put loc of card window into cardLoc
41.      set loc of msg box to msgLoc
42.      set loc of tool window to toolLoc
43.      set loc of pattern window to patLoc
44.      put true into updateWindoids
45.  else
46.      -- update windoid prefs
47.      checkWindoid "msg box", "msgLoc"
48.      checkWindoid "tool window", "toolLoc"
49.      checkWindoid "pattern window", "patLoc"
50.  end if
51.  if updateWindoids then
52.      lock screen
53.      push card
54.      go to card "Windoid Preferences" of stack "home"
55.      put cardLoc into card field "cardLoc"
56.      put msgLoc into card field "msgLoc"
57.      put toolLoc into card field "toolLoc"
58.      put patLoc into card field "patLoc"
59.      pop card
60.  end if
61.  end trackWindoids
62.
63.  on checkWindoid windoidName, prefName
64.      global updateWindoids, msgLoc, toolLoc, patLoc
65.      if value("loc of " & windoidName) is value(prefName)
66.          then exit checkWindoid
67.          put true into updateWindoids
68.          do "put loc of " & windoidName & " into " & prefName
69.      end checkWindoid

```


Handler name:`initWindoidTracking`**What it does:**

This is the first part of the windoid tracking system. It initializes variables that hold the locations of your windoids and gets things ready to go.

Parameters:

None. You don't have to call this handler. It'll be called when HyperCard first gets started.

How it works:

We'll need a bunch of global variables to keep track of where we want our windoids to be located. Line 8 declares the globals that will hold the locations for the three windoids we're tracking: the message box, the tool palette, and the pattern palette. There's also a global called `windoidTracking` that's used to tell if the tracking system is turned on.

We're going to look for the "windoid preferences" card, so we lock the screen to hide the card-to-card movement in line 9. Line 10 will remember where we started so we can return quickly, and line 11 is the command that takes us to the preferences card.

If there is no card named "windoid preferences," HyperTalk's "result" function will hold an error message. Line 12 checks to see if that happened, and if so, we politely say that we can't do windoid tracking (line 13) and we give up (line 14). If the result is empty, then we can assume we reached the preferences card safely, and we can start getting values out of it.

The next hunk of statements methodically extracts information from the preferences card, sets up the windoids, then hides or shows the windoids as preferred by the user. Line 18 sets up the `windoidTracking` global, which we'll check on later.

Lines 19 through 22 recall the desired positions for the three windoids and put their values into the corresponding globals. We also put the value for the card's location into a variable called `cardLoc`. Lines 23 through 26 then take those values and move the windoids, plus the card window, to the specified locations. Lines 27 through 29 show or hide the windoids according to the settings

of the three check boxes on the preferences card (see below for a description of the preferences card).

That's the end of all the setting up. All that's left to be done is to return to the card we started at (line 30) and unlock the screen (line 31).

Like the "Typing in shorthand" scripts we discussed in Chapter 2, the windoid tracking system uses some information that's recorded on a card in the Cookbook Reference stack. Rather than stash this information away in hidden fields, we've made the numbers visible and completely accessible, since that's an easy way for users to set and change their values.

The check boxes on the card simply show or hide a windoid according to their setting. In addition, we need to update the windoids and the globals if the user changes the fields directly. To handle this, we have an "on closeField" handler in the card script that simply calls `initWindoidTracking`.

The next script takes care of watching carefully for windoid movement, and doing something about it when it occurs.

Handler name:

`trackWindoids`

How it works

This handler is called at idle time. It watches to see if the card window moves. If it does, the positions of the windoids are changed to keep them at the same relative location to the card window. `TrackWindoids` also updates the fields on the preferences card to the correct values.

Parameters:

None. You won't have to call this handler. It should be called in an "on idle" handler so that it can watch for things being moved.

How it works:

`TrackWindoids` needs access to all the familiar globals, so it declares them in line 35. Line 36 makes sure that the tracking system is turned on before we go monkeying around. This handler uses a variable called `updateWindoids` to know if anything has to be updated, and it initializes that variable in line 37.

Line 38 checks to see if the card window has been moved. If so, a whole load of stuff happens (lines 39 through 44). First, the card window's new location is recorded (line 40). Then, the positions of all the windoids are reset to their old values (lines 41 through 43), and `updateWindows` is set to indicate that something changed. Wait—the windoids didn't move, the card window did. So why do we need to reset the locations of the windoids?

The answer lies in the way that HyperCard keeps track of windoid positions. Windoid locations give the windoid's distance from the *card window*, not the absolute screen location. The positions are in card-relative coordinates; point 0,0 is wherever the top left corner of the card is. When you move the card window, HyperCard sneakily changes the windoids' location properties so that the windoids don't move.

Our tracking system is actually undoing this sneaky work of HyperCard! We're jamming the old location values back into the windoids. You could say that HyperCard's method changes the windoids' *relative* positions to preserve their *absolute* positions, while our system changes their *absolute* positions to preserve their *relative* positions.

The “else” part of the statement, which is in lines 47 through 49, is run if the card window didn't move. These lines check to see if the user moved any of the windoids. The checking is done by calling a utility named `checkWindoid`, which we'll describe next. If any of these windoids has moved, the `updateWindoids` global will be set to true.

Finally, the last task of our handler is to fix up the preferences card of the Home stack in case anything has changed. If the card window or any of the windoids moved, `updateWindoids` was set to true, so lines 52 through 59 will do their work. These lines just lock the screen (line 52) and save our place (line 53), then take us to the preferences card (line 54). Once we're there, we update the location fields to the new values for each variable. We go back to where we were (line 59), and we're done, relying on idle time to unlock the screen again for us.

Handler name:

checkWindoid

What it does:

This is a utility called by trackWindoids at idle time. It checks to see if a specified windoid has moved from its location as recorded on the preferences card. If it has, the updateWindoids global is set to true, and the new location of the windoid is thrust into the variable. TrackWindoids calls this handler three times, once each for the message, tool, and pattern windoids.

Parameters:

There are two parameters. The first one, windoidName, tells which windoid to check on. The second parameter, prefName, is the name of the variable holding the (maybe former) location of the windoid. Normally, you won't call this handler. It's called by trackWindoids.

How it works:

Line 64 tells us which globals checkWindoid needs to know about. Line 65 executes a “value” statement that evaluates an expression. If this pass through checkWindoid is looking at the message box, for example, the expression will look like this

```
loc of message box is msgBox
```

and will result in either true or false. The “if” statement that calls value will return true or false. If it's false, the values matched, and line 66 exits the handler if the windoid hasn't moved. Otherwise, the windoid has changed its location (probably with the help of that unpredictable user) and we need to flag it (line 67). Line 68 is a “do” statement that puts the new value of the windoid's location into the global variable holding it. And that's it!

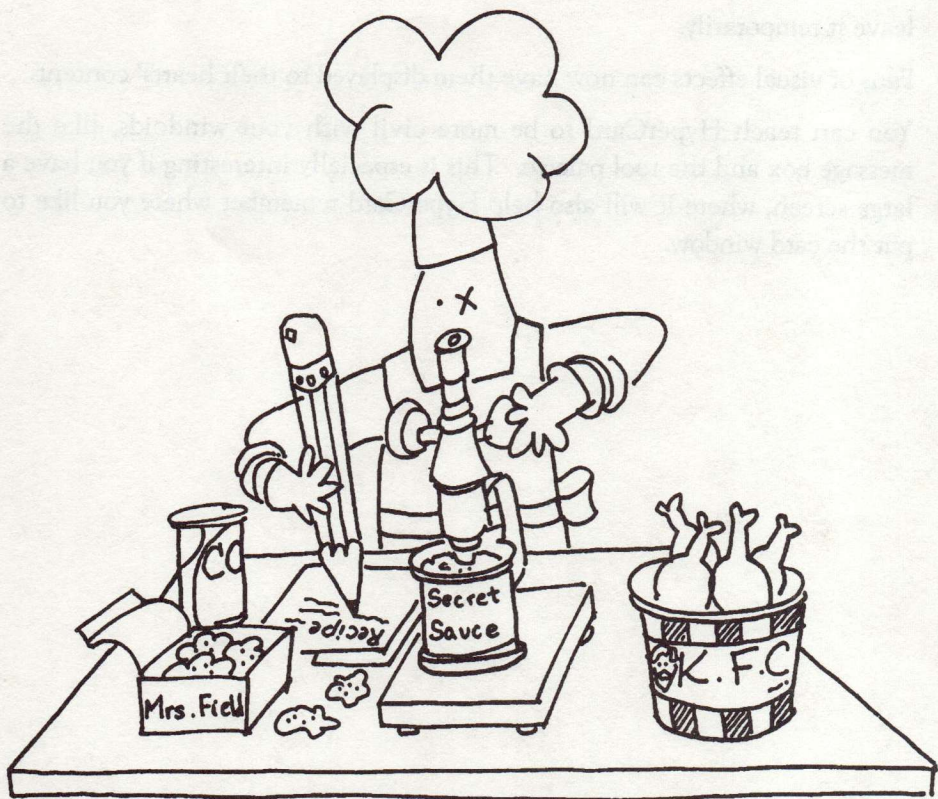
Summary

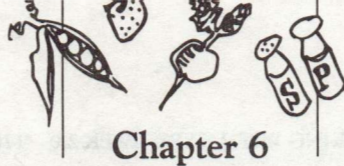
- We looked at a few handlers to help with moving and rearranging cards.
- There's a set of handlers that implement a system for leaving notes in the stack script. We can use these notes to hold our place in a stack when we

leave it temporarily.

- Fans of visual effects can now have them displayed to their hearts' content.
- You can teach HyperCard to be more civil with your windoids, like the message box and the tool palette. This is especially interesting if you have a large screen, where it will also help HyperCard remember where you like to put the card window.







Chapter 6

CHECKING THE INGREDIENTS

In this chapter we present some handlers to use when creating your own stacks or tromping around in other people's stuff. These scripts provide you with a variety of information that might be valuable as you travel through your stacks.

Show All Buttons

This handler provides a quick and fancy way to make sure you can see all the buttons in the current background and card. It guarantees that all of the buttons have their "visible" property set to true, and forces each button's location to be somewhere on the Macintosh-screen-sized card.

```
1.  on showAllButtons
2.      -- script 6-1:  Show all buttons.
3.
4.
5.      -- This handler shows all hidden buttons and moves
6.      -- them all into the visible part of the card, too.
7.
8.      put localRect(rect of card window) into cardRect
9.      put 0 into stagger
10.     repeat with i = 1 to the number of buttons
11.         show button i
12.         if loc of button i is not within cardRect then
13.             set loc of button i to 100+10*stagger,
14.                 100+10*stagger
15.             add 1 to stagger
16.         end if
17.     end repeat
18.     repeat with i = 1 to the number of bkgnd buttons
19.         show bkgnd button i
```



```

19.      if loc of bkgnd button i is not within cardRect
           then
20.      set loc of bkgnd button i to 100+10*stagger,
           100+10*stagger
21.      add 1 to stagger
22.    end if
23.  end repeat
24. end showAllButtons

```

Handler name:

showAllButtons

What it does:

This handler enables you to see all the current card and background buttons. It ensures first that they're visible, and then that they're all within the rectangle determined by the card's window. This command is useful when you question you're looking at all of a card's or background's buttons and want a quick way to see them all.

Parameters:

None.

How it works:

The handler starts by asking how big the current card window is and then saves that value in cardRect. Next, it zeroes a variable called stagger (hic) used for showing buttons positioned somewhere beyond the visible portion of the card. After that, the work in this handler is done in two separate repeat loops. The first loop is in lines 10 through 16, and takes care of card buttons. The repeat statement in line 10 instructs the loop to run once for every card button. Line 11 sets the current button's "visible" property to true, which shows the button unless it's off the card somewhere, in which case lines 12 through 15 come in.

Line 12 checks to see if the button has indeed been banished beyond the edge of the card. If a wayward button is found, we have to move it back onto the visible part of the card. Since we might find more than one button to relocate, we have to be sure we don't move them in on top of each other. So, we use the stagger variable to define an offset from 100,100 (an arbitrary location) in line

13, and increment stagger in line 14 to make sure the next one we find won't be placed on top of this one.

After the card buttons are handled, the repeat loop in lines 17 through 23 does exactly the same thing for all the background buttons. The statements in this loop are the same as in the first one, except that all references to buttons are now about background buttons instead of card buttons. After this second repeat loop ends, the handler is all done.

Notice, this handler wantonly disregards whether or not you wanted to keep some buttons hidden. Even if they were hidden before you called `showAllButtons`, they're all going to be visible after you call it, and some of them may even have had their locations changed. A fun exercise (no, really) would be to modify this script so that it leaves things the way they were. You'd have to preserve two properties (at most) for every button: visible and location. Good luck!

Show All Fields

This handler lets you see all the fields that belong to the current card and background. It works just like `showAllButtons`.

```

1.  on showAllFields
2.      -- script 6-2: Show all fields.
3.
4.
5.      -- This handler shows all hidden fields and moves
6.      -- them all into the visible part of the card, too.
7.
8.      put localRect(rect of card window) into cardRect
9.      put 0 into stagger
10.     repeat with i = 1 to the number of fields
11.         show field i
12.         if loc of field i is not within cardRect then
13.             set loc of field i to 100+10*stagger,100+10*
                stagger
14.         add 1 to stagger
15.         end if
16.     end repeat
17.     repeat with i = 1 to the number of card fields

```



```

18.      show card field i
19.      if loc of card field i is not within cardRect then
20.          set loc of card field i to 100+10*stagger,
              100+10*stagger
21.          add 1 to stagger
22.      end if
23.  end repeat
24. end showAllFields

```

Handler name

showAllFields

What it does:

This handler makes sure you can see all the fields on the current card and background (unless they're covered up by paint, other fields or buttons).

Parameters:

None.

How it works:

Well, guess what? It works exactly like showAllButtons, the preceding handler. Now that may not be very exciting, and just to make it up to you, here's a suggestion that is: create a new stack. Make an additional card in the new stack, so there are two cards in it. Create a background field to cover just about the whole card. Set its textFont to Monaco and its textSize to 9 points. Find the script for showAllButtons, copy it, and paste it into the field on one of the new stack's cards. Paste the script for showAllFields into the field on the other card. Make sure you don't have any blank lines at the top of either field.

Now, you're ready for action. Using the arrow keys (option-arrow if you have textArrows turned on, and you know who you are), move from card to card. As you do, stare first at the parts of the script that don't change, which comprise most of it. Then watch closely as each "button" becomes "field," and vice versa. Isn't that neat? Yet another aspect of the hidden entertainment value that HyperCard provides.

Show All Hidden Things

Now that we've got a couple of scripts to go on a thorough mission to discover and display all hidden buttons and fields, we'll put them together with a couple of extras to form a one-stop shopping service for displaying hidden information.

```

1.  on showEverything
2.    -- script 6-3:  Show all hidden things.
3.
4.
5.    -- This handler makes visible all buttons and
6.    -- fields, plus the card and background
7.    -- picture, too.
8.    show card picture
9.    show bkgnd picture
10.   showAllButtons
11.   showAllFields
12. end showEverything

```

Handler name:

showEverything

What it does:

This handler makes sure four kinds of things are visible on the current card: buttons, fields, the background picture, and the card picture. It's kind of fun to hide a bunch of things on a card, then use showEverything to pop all the pieces back into place, one at a time.

Parameters:

None.

How it works:

This handler executes four statements to display some possibly hidden things. The first two, in lines 8 and 9, are simply HyperTalk commands to make sure the card and background picture are visible. The commands in lines 10 and 11, of course, call the command library to make sure all buttons and fields, whether

they're just hidden or actually off the visible part of the card, are made to show themselves.

Create Statements to Show Hidden Buttons

The next series of handlers use a different technique for dealing with hidden buttons and fields. Instead of just blasting them into the visible world, these handlers return information about the shy items in the form of statements that will show the hidden things, plus comments about where they're hidden.

```

1.  function reportHiddenButtons
2.      -- script 6-4:  Create statements to show hidden
3.      -- buttons.
4.
5.      -- This function returns statements that will show
6.      -- any buttons that are hidden or are off the
7.      -- visible part of the card.
8.      put localRect(rect of card window) into cardRect
9.      put empty into report
10.     repeat with i = 1 to the number of buttons
11.         if not visible of button i
12.             then put showStmnt("button " & i) after report
13.             get the loc of button i
14.             if it is not within cardRect
15.                 then put setStmnt("button " & i,it) after report
16.         end repeat
17.     repeat with i = 1 to the number of bkgnd buttons
18.         if not visible of bkgnd button i
19.             then put showStmnt("bkgnd button " & i) after report
20.             get the loc of bkgnd button i
21.             if it is not within cardRec
22.                 then put setStmnt("bkgnd button " & i,it) after
23.                     report
24.         end repeat
25.     return report
26. end reportHiddenButtons

```



```

27. function showStmnt objName
28.     return "show " & objName & " -- " & objName & " is
        hidden" & return
29. end showStmnt
30.
31. function setStmnt objName,oldLoc
32.     return "set loc of " & objName & " to 100,100 -- " &
33.         & objName & " is at " & oldLoc & return
34. end setStmnt

```

Handler name:

reportHiddenButtons

What it does:

ReportHiddenButtons scans through all buttons, including card and background buttons, looking for those with their “visible” property set to false and those with their “location” property set to a value not visible on the card. If it finds any of those, it constructs statements that will make them visible.

Parameters:

None.

How it works:

This function works something like showAllButtons, but with some differences. Instead of simply making all the hidden buttons visible, this handler creates a report, which is text containing statements for making the buttons visible. Each statement generated also comes with a comment describing why the button can’t be seen.

This handler’s similarity to showAllButtons is in the loop going through all the card buttons (lines 10 through 16) and all the background buttons (lines 17 through 23). Within each loop, it checks to see if the button is visible (line 11 for card buttons, line 18 for background buttons). If not, it builds a “show” statement for that button using the showStmnt handler (explained below) and puts it into the report variable (line 12, line 19).

After this check, reportHiddenButtons looks to see if the current button is located off the visible part of the card by getting the button’s location (line 13,

line 20) and checking it against the card's rectangle (line 14, line 21). If it's off the beaten path, a statement to bring it in is added to report (line 15, line 22) using the `setStmnt` handler (also described below).

After the repeat loops for card and background buttons are finished, the function just returns the report as its result, and then ends. To see how the `setStmnt` and `showStmnt` handlers build their commands, just keep on reading!

Handler name:

`showStmnt`

What it does:

`ShowStmnt` builds a statement that makes a hidden object visible. It adds a comment after the statement that tells what it's doing.

Parameters:

This function takes one parameter, which is the name of the object to be shown. You don't normally call this handler directly. It's an assistant to `reportHiddenButtons` and `reportHiddenFields` (described in the next section).

How it works:

There's just one operative line in `showStmnt`, so it can't be all that complex, right? Right. Line 28 simply strings together the word "show," the name of the object passed to it, the comment delimiter "--," the name of the object again, the words "is hidden," and a carriage return. The function returns this string as its result, and that's all it does.

Handler name:

`setStmnt`

What it does:

This is another little auxiliary function used by the `reportHidden` handlers. It's similar to `showStmnt`, except that it builds statements to show objects located off the visible part of the card, rather than objects with their "visible" property turned off.

Parameters:

This one takes two parameters. The first is `objName`, the name of the object that's going to be used, just like in `setStmnt`. The second parameter is the current (invisible) location of the object. Once again, this is a handler you probably won't call directly, unless you have some insatiable desire to call every handler around just to see if they work like we say they do, which you certainly have the right to do.

How it works:

Like the previous handler, this one has only one line. It puts together some text and returns it as the function result. In this case, the text is the "set loc of" command, plus the name of the object, the words "to 100,100 --," the object name again, "is at," the object's location, and a carriage return. All off-planet objects are set to move to the same location (100,100) instead of being staggered as they were in the `showAll` scripts because the `showAlls` do their work directly. The `reportHiddens` just give a report you can change or deal with before it causes any action.

Create Statements to Show Hidden Fields

This function, like `reportHiddenButtons`, creates a list of statements used to show hidden stuff. Just like its button counterpart, this handler checks card fields and background fields during its travels.

```

1.  function reportHiddenFields
2.      -- script 6-5: Create statements to show hidden
3.      -- fields.
4.
5.      -- This function returns statements that will show
6.      -- any fields that are hidden or are off the visible
7.      -- part of the card.
8.      put localRect(rect of card window) into cardRect
9.      put empty into report
10.     repeat with i = 1 to the number of fields
11.         if not visible of field i
12.             then put showStmnt("field " & i) after report

```



```

13.      get the loc of field i
14.      if it is not within cardRect
15.      then put setStmnt("field " & i,it) after report
16.  end repeat
17.  repeat with i = 1 to the number of card fields
18.    if not visible of card field i
19.    then put showStmnt("card field " & i) after report
20.    get the loc of card field i
21.    if it is not within cardRect
22.    then put setStmnt("card field " & i,it) after
        report
23.  end repeat
24.  return report
25. end reportHiddenFields

```

Handler name:

reportHiddenFields

What it does:

It creates statements used to show any fields that have their visible property set to false or have their location set to a place not visible.

Parameters:

None.

How it works:

Just like the showAll handlers, the reportHiddens are exactly, perfectly, completely identical in every detail, except for two small differences. First, every occurrence of the word "button" in reportHiddenButtons is replaced by the word "field" in reportHiddenFields. Second, reportHiddenFields has a tiny heart-shaped mole on the middle of its left thigh. Otherwise, they're the same.

Create Statements to Show Hidden Fields and Buttons

This little handler is a quick and easy way to create statements that will show both hidden fields and hidden buttons.


```

1. function reportHidden
2.   -- script 6-6: Create statements to show hidden
3.   -- buttons and fields.
4.
5.   -- This function returns statements that will show
6.   -- any buttons and fields that are hidden or are off
7.   -- the visible part of the card.
8.   return reportHiddenButtons() & reportHiddenFields()
9. end reportHidden

```

Handler name:

reportHidden

What it does:

This handler strings together the responses from reportHiddenButtons and reportHiddenFields.

Parameters:

None.

How it works:

ReportHidden calls reportHiddenButtons and reportHiddenFields, puts their answers together by using the & operator, and then returns the strung-together result.

Show Each Button with Its Number

When you start to build complex cards with lots of buttons, you may need to know which button has which number as you're working on it, in case you need to hide a button or talk to a button in a script ("Hey, you, button" won't work). This can be especially interesting when some of the buttons are hidden, since invisible things are very hard to see. This handler helps you figure out which button is which.

```

1. on showButtons string
2.   -- script 6-7: Show each button with its number.
3.

```



```

4.
5.    -- This handler selects each card button and tells
6.    -- you what its button number is and whether it's
7.    -- visible.
8.    if the number of buttons is zero then exit
      showButtons
9.    put word 2 of string into buttonNum
10.   if "hidden" is in string then hide button buttonNum
11.   add 1 to buttonNum
12.   if buttonNum > the number of buttons then put 1 into
      buttonNum
13.   put "visible" into status
14.   if not visible of button buttonNum then put "hidden"
      into status
15.   show button buttonNum
16.   select button buttonNum
17.   put "showButtons " & quote & "Button " & buttonNum &
      " is " & status & quote
18. end showButtons

```

Handler name:

showButtons

What it does:

This handler selects each card button, one by one, and displays some information about the button in the message box. It doesn't matter if the button is hidden—it can't hide from showButtons, which displays it (along with the fact that it's hidden), then hides it again when you show the next one. Each line of information put in the message box actually has two purposes. It also sets up a message to showButtons causing the next button in order to be processed.

For example, when you call showButtons, it will select button number 1 and put a line like this

```
showButtons "button 1 is visible"
```

into the message box. Then, you can simply press the return key, which ships off the contents of the message box. This time, showButtons is called again, but the

parameter "button 1 is visible" helps showButtons figure out that it should now show button 2.

There are several interesting, subtle, and relatively devious ideas in this script. The handler uses the message box as a place to write a message that is later *sent to itself* when the user presses return. See if the techniques in this handler inspire you to come up with new ways to use them.

Parameters:

Well...this is interesting. When you call showButtons, you don't pass any parameters. That's how the handler figures out you're calling and want to see the buttons starting with number 1. On subsequent calls, you'll be passing the parameter that showButtons automatically generated for you in the message box. So, the definition for showButtons includes one parameter, named string, but you usually won't pass this parameter when you call.

It's tricky

If you want to be really clever, you can tell showButtons to start at a button other than the first one. ShowButtons looks for the button number in the second word of the parameter (see "How it works" below for details). If you use something like this,

```
showButtons "button 4"
```

you'll start out, er, at button 5, since the script adds one to the button number before beginning. It does this because it's usually passed an empty value, which evaluates to zero, and it has to get it to one, or else it's passed the number of the previously displayed button, which has to be incremented.

How it works:

This handler, like most of the ones in this book, is very picky about wasting your time, so the first thing it does is check to see if there are any buttons at all (line 8). If there aren't any, it quits the handler. Note: it *only* looks for card buttons because it doesn't specifically mention background buttons. Assuming there's

something to look for, line 9 gets the button number that's in the parameter, if there is one. If there's no parameter, buttonNum is simply set to empty.

Remember, there are two ways that showButtons gets called: directly by the user, usually with no parameter, and by a command in the message box produced by a previous call to showButtons, which will have a parameter containing the number of the last button we looked at, as well as telling whether it was visible or hidden. If it was hidden, we want to put it back the way we found it, so line 10 checks for that condition and hides the button if necessary.

Moving right along, line 11 moves us to the next button's number. Line 12 checks to see if we've gone too far and produced a number that's not the number of any button. If so, we'll simply wrap around and start over again with button number 1. Lines 13 and 14 set up a variable called status to hold the appropriate word, "visible" or "hidden," that describes the condition of the button we're going to show.

It's showtime in line 15. You might be surprised that line 15 doesn't look like this:

```
if not visible of btn buttonNum then show button buttonNum
```

Well, it's actually more trouble for HyperTalk to test the visibility of the button than to simply call "show" for every button, whether it's already visible or not. Since we're always out for speed, we take this route. Line 16 selects the button, which makes it appear with that cool shimmer around it, so we know it's either been selected or possessed by demons.

Finally, line 17 is the coup de grâce, putting the visibility of the current button into the message box, wrapping it in a call to showButtons that will cause the next button to be displayed when the whole mess starts again. In this way, showButton helps make sure it will be called again. This is sort of recursive nepotism.

Display Mouse Coordinates

When you're trying to figure out where to position things on the Macintosh's screen, or you're attempting to learn the location of something already on a card, it's useful sometimes to know the current x-y location of the mouse. This handler helps you by constantly showing the mouse's coordinates in the message box.


```

1.  on xy
2.      -- script 6-8: Display the current mouse
3.      -- coordinates.
4.
5.      -- This handler shows the current x-y position of
6.      -- the mouse. The user can see global coordinates by
7.      -- holding down shift.
8.      repeat until the mouse is down
9.          get the mouseLoc
10.         if the shiftKey is down then -- global coords
11.             add left of card window to item 1 of it
12.             add top of card window to item 2 of it
13.         end if
14.         put it
15.     end repeat
16. end xy

```

Handler name:

xy

What it does:

This handler lets you see a continuous display of the mouse's x-y position as you (presumably) merrily move it around the screen. HyperCard really uses two different coordinate systems. For some values, such as the location of the card window and the screen rectangle, it uses the top left corner of the screen as 0,0. For most coordinates, it treats the upper left corner of the card window as 0,0. Our xy handler will report coordinates in both forms. Normally, it shows you the card window-relative coordinates, which is what you usually want. By holding down the shift key, you'll get the global coordinates.

When you're all done watching the coordinates go by, you can end the handler by clicking the mouse.

Screen me

On any Macintosh that uses a built-in monitor, and that's most of them, the card window is the same size as the screen for standard-sized (9-inch) stacks. On these Macintoshes, the card window is usually never moved, so there's no difference between local and global coordinates. The upper-left corner of the screen is the same as the upper-left corner of the card, so the two coordinate systems are right on top of each other.

Parameters:

None.

How it works:

Line 8 starts the repeat loop that contains the entire handler. This repeat will make sure the handler keeps running until the user clicks the mouse to return to regular programming. Of course, you can also end this (or any) handler by typing command-period. Line 9 finds out where the mouse is by asking HyperTalk, via its mouseLoc function.

Like most of HyperTalk, mouseLoc deals in coordinates relative to the upper-left corner of the card window. So, in line 10, we check to see if the user is interested in global coordinates—that is, coordinates relative to the upper-left corner of the screen, beyond HyperCard's world. The user signals for global coordinates by pressing the shift key, tested in line 10.

If we need to make the coordinates global, then lines 11 and 12 are executed. At first thought, converting local (card window-based) to global coordinates may seem impossible, or hard, or non-trivial. In fact, it's about the easiest thing we've done all day, and here's how.

We know the local coordinate system is at a certain position relative to the global coordinates. For example, let's say the card window is at 80,40 (this would probably be on a big-screen Macintosh, but it would work on any Macintosh). If we put the mouse at the exact upper-left corner of the card window, under the window's title bar, we're at point 0,0 in local coordinates, and 80,40 in global coordinates. We can convert from the local to the global system just by adding 80 to the x and 40 to the y coordinate.

How do we know that 80 and 40 are the right values to add? Easy—we got them from the easily-accessible properties left and top of the card window. And, not surprisingly, that's exactly what we need to do in lines 11 and 12 of the xy script. The variable "it" already has the mouse location, thanks to line 9, so we can convert it to global coordinates by adding the values of the left and top properties in lines 11 and 12. The mouseLoc function put the mouse's location into "it" as two items with a comma between them, so we can use HyperTalk's slick "item" chunk specifier to add the right numbers.

At the end of the if statement (line 13), "it" has the right coordinates, which will have been converted to global values if the shift key was held down. That's all the work we need to do, and line 14 puts the coordinates into the message box for the user to ponder. Line 15 closes the repeat statement that holds control until the user clicks the mouse button.

Quick Information about Current Stack and Card

The first five items in the Objects menu provide a lot of useful information about your stack and the things in it. These are the Button, Field, Card, Bkgnd, and Stack Info items. They tell you the object's name and ID, and how many other objects it contains (for cards, backgrounds, and stacks). There are a lot of times when it's useful to know how many cards, fields, and buttons there are around, so we created this little handler to show you the information that you'd normally have to look through three dialogs for (or type in three separate HyperTalk commands).

```

1.  on info
2.      -- script 6-9: Quick information about the current
3.      -- stack and card.
4.
5.      -- This handler lets you see several useful pieces
6.      -- of information about the current card and stack
7.      -- with one command.
8.      put number of cards && "cards," && number of bgs↵
9.      && "bgs," && number of bg flds && "bg fields,"↵
10.     && number of bg btns && "bg btns," && ↵
11.     number of cd flds && "card fields," && number of cd
        btns↵

```



```

12.    && "card btns"
13.  end info

```

Handler name:

info

What it does:

This handler simply tells you how many of several of our favorite things there are. Specifically, it reports in the message box the number of cards in the stack, backgrounds in the stack, background fields and buttons in the current background, and the number of card fields and buttons on the current card. In that order, of course.

Parameters:

None. This is supposed to be a quick and easy way to get the information, so minimizing typing is an important goal. Parameters would just screw that up.

How it works:

Info wins the award for the easiest handler in the book. It consists entirely of one very long line, although it has to be broken up with option-returns to make it legible. The single line of HyperTalk is a put statement that strings together the items we're interested in by calling the "number" function a few times. We use the && operator to stick them all together with a space in between each one, and in the end, the whole thing fits into the message box.

We created this handler because we frequently needed to know this information. If you find yourself constantly checking some other value or property that HyperTalk can get to easily, go ahead and make your own command scripts to display that information for you. If you're frequently interested in knowing the positions of the tool and message windows, for example, don't kill yourself typing "loc of tool window" and "loc of msg" all the time. Just write a little command script that sticks these two properties into the message box and put the script into your command library. That's what it's there for!

Summary

- This chapter is for people who like to go poking around in stacks to see how things work.

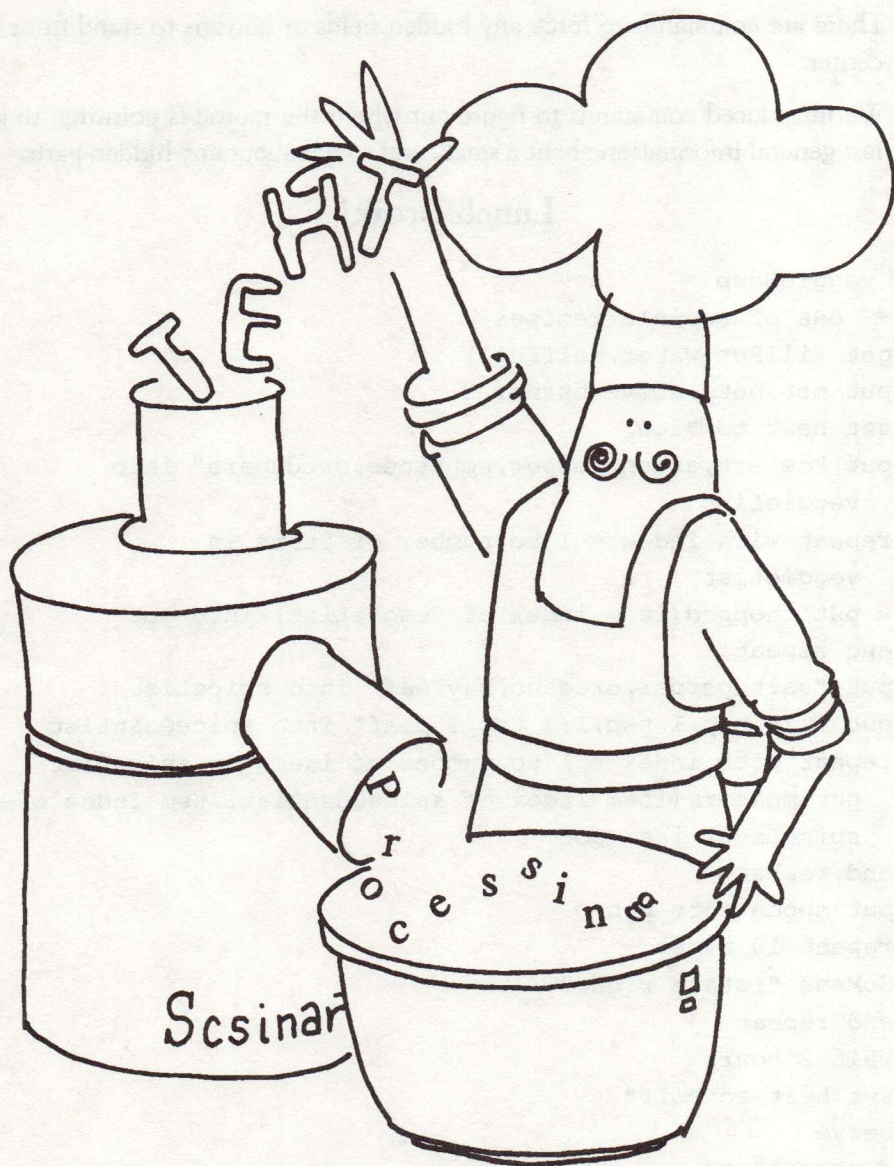
- There are commands to force any hidden fields or buttons to stand front and center.
- We introduced commands to figure out where the mouse is pointing, to get fast general information about a stack, and to tell about any hidden parts.

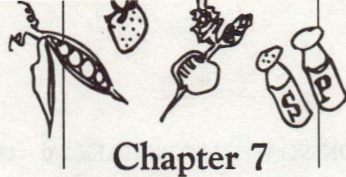
Lunch Break!

```

on veggieSoup
  -- one of Robin's recipes
  get fillPot(water, halfFull)
  put pot onto stove burner 3
  set heat to "low"
  put "celery, green pepper, potatoes, cucumbers" into
    veggieList
  repeat with index = 1 to number of items in
    veggieList
    put chopped(item index of veggieList) into pot
  end repeat
  put "salt, pepper, oregano, bayleaf" into spiceList
  put "1/4 tsp, 1 tsp, 1/4 tsp, 1 leaf" into spiceQuanList
  repeat with index = 1 to number of items in spiceList
    put measure(item index of spiceQuanList, item index of
      spiceList) into pot
  end repeat
  put spoon into pot
  repeat 10 times
    doMenu "rotate right"
  end repeat
  wait 2 hours
  set heat to "off"
  serve
end veggieSoup

```



Chapter 7

PROCESSING TEXT

There's a lot of talk and action these days about getting computers to work with all kinds of media: graphics, sound, animation, smell—well, maybe not *all* kinds of media, but lots of different ones. This work is very real, but a lot of what we deal with on computers is still pure text. The scripts in this chapter help you organize the mass of text that gets thrown your way.

Most of the commands in this chapter operate on the selection. In order for these to work, you have to access the command library through the command box, by typing shift-tab. If you type a command into the message box, the selection will go away and nothing much will happen.

Indent Selection As if It Were a Script

It's often convenient to work with scripts in fields, rather than in their natural script editor habitat. This is especially useful if you want to make sure the scripts aren't anywhere in the current message path while you're working with them. The handler in this section helps you by indenting text as if it were a script.

```
1. on indentScript
2.   -- script 7-1: Indent the selection as if it were a
3.   -- script.
4.
5.   -- This command handler helps you work with scripts
6.   -- in fields by indenting them properly.
7.
8.   get the selection
9.   put script of this card into saveScript
10.  set script of this card to it
11.  put script of this card into it
```



```

12.    set script of this card to saveScript
13.    replaceSelection it
14.    end indentScript

```

Handler name:

indentScript

What it does:

This handler formats text according to the script editor's indentation rules. First, select the text that you want to be indented. Then, bring up the command box and type indentScript.

Parameters:

None.

How it works:

You might guess that this handler would involve some long, complicated formatting code. Ha! Why should we do all the work when HyperTalk already knows what to do? The answer is, we shouldn't. Here's how we get the script editor to do it for us.

We begin by getting the selection into the variable "it" in line 8. We're going to use the card script as a place to format our script, so we save the card's real script in line 9. Then, line 10 jams our text into the card script.

We just need to grab the text back (line 11), and HyperTalk's magic script-grabbing code will format the script for us while it's getting it! We then carefully replace the card's real script (line 12) so that nobody will know we've been fooling around in there. Finally, line 13 puts the text back into the selection, using the replaceSelection handler described in Chapter 8. This handler proves that it's not what you know that counts, but how clever you can be with it.

Put Numbers at Beginning of Lines

When you're working with text consisting of a list of things, one thing per line, like groceries to buy or your comic book collection, it's often useful to number them. This handler will number lines in a field for you automatically. We like this script so much that we used it to number all the script listings you see in

this book. We've also got a handler to take the numbers away, and a utility function used by both `numberLines` and `unNumberLines`.

```

1.  on numberLines lineNumber
2.      -- script 7-2: Put numbers at the beginning of
3.      -- lines.
4.
5.      -- This handler numbers lines of text. We used it
6.      -- for the script listings in this book. It saved
7.      -- us a lot of work!
8.      if lineNumber is empty then put 1 into lineNumber
9.      put the selection into list
10.     repeat with i = 1 to the number of lines in list
11.         set cursor to busy
12.         get stripNumber(line i of list)
13.         put lineNumber & ". " into prefix
14.         add 1 to lineNumber
15.         repeat until length of prefix >= 5
16.             put " " before prefix
17.         end repeat
18.         put prefix & it into line i of list
19.     end repeat
20.     replaceSelection list
21. end numberLines
22.
23. on unNumberLines
24.     get the selection
25.     repeat with n = 1 to the number of lines in it
26.         put stripNumber(line n of it) into line n of it
27.     end repeat
28.     replaceSelection it
29. end unNumberLines
30.
31. function stripNumber string
32.     put 1 into n
33.     repeat while char n of string is " "
```



```
34.      add 1 to n -- skip leading spaces
35.  end repeat
36.  repeat while char n of string is in "0123456789"
37.      add 1 to n -- skip number
38.  end repeat
39.  if char n of string is "."
40.  then delete char 1 to n+2 of string
41.  return string
42. end stripNumber
```

Handler name:

numberLines

What it does:

This handler takes the current selection and puts sequential numbers in front of each of its lines. We used this script (among others) in the creation of this book! Every time you see a script listed with its lines numbered, you're looking at something produced by numberLines, untouched by human hands.

Parameters:

There's one parameter, named startNumber, used to tell the handler what number to start with when numbering the lines. The parameter is optional. If you don't specify a starting number, the handler will just start numbering from 1.

How it works:

If you don't pass a value at the optional parameter, line 8 discovers that and supplies the reasonable default starting value of 1. NumberLines operates on the selection, and line 9 puts the selection into a local variable called list where we can manipulate it.

Most of the handler happens in lines 10 through 19, which make up a repeat loop that executes once for each line in list. Numbering things can take some time, especially for large hunks of lines, so we're going to spin the famous beach ball cursor while the numbering is going on. Line 11 handles that operation. In line 12, we make sure to remove any old line numbers that might be present. We accomplish this with a call to the stripNumber function, which simply

chops off any line number we might previously have attached (it's described below, if you're curious). This leaves the current line, unnumbered, in "it," due to the "get" statement in line 12.

Line 13 takes the line number chosen to be affixed to the current line, adds a period and two spaces to the end, and puts the resulting value into the "prefix" variable. Line 14 then bumps the line number up by one.

Line 15 starts a clever little repeat loop, if I may say so (after all, I didn't write it), that sticks spaces in front of the prefix (line 16) until the length of the entire prefix is 5 characters. The effect of this is to right-justify the line number, with the padding spaces coming first. This makes the numbered lines look very nice. Line 18 puts the prefix, containing the line number in front of the line that's been stripped of any old number (in "it") and replaces the old line with the freshly numbered one in list. The repeat loop processing every line finally ends in line 19. Line 20 replaces the selection with the newly numbered lines. (The replaceSelection script is described in Chapter 8.)

Handler name:

unNumberLines

What it does:

This command removes any line numbers from the lines in the selection. You can use this handler to get rid of the nifty line numbers created by numberLines. This is useful for returning a handler to its original condition so you can place it into a script and actually have a shot at it working (if it has line numbers, you're guaranteed to have no shot at all).

Parameters:

None.

How it works:

The operation of this handler is pretty straightforward. Most of the work is done by the stripNumber function, described next. In line 24, the selection is loaded into "it" for our hacking convenience. Lines 25 through 27 move through the selection one line at a time, replacing each line with its number-stripped equivalent courtesy of stripNumber. When the stripping is done, line 28 calls replaceSelection (described in Chapter 8) to put the new version back.

Handler name:`stripNumber`**What it does:**

This function removes the line number prefix from the beginning of a line. Specifically, it looks for a string of digits followed by a period, and possibly preceded by one or more spaces (techie types might say “preceded by zero or more spaces”). It’s called by both `numberLines` and `unNumberLines`. If the line passed in doesn’t have a number prefix, nothing happens to it.

Parameters:

The parameter named `string` holds the line of characters due to get its number prefix removed. If there’s no prefix on the line, it’s passed through unchanged.

How it works:

`StripNumber` uses a variable called “`n`” to denote the current character position of the string it’s processing. Line 32 initializes `n` to 1 so that we can start looking for line number prefixes. Lines 33 through 35 look for and skip over any blanks that occur at the beginning of the line. In the same way, the repeat loop in lines 36 through 38 skips over the number which should come right after the leading blanks.

By the time the script arrives at line 39, the variable “`n`” should hold the position of the period that comes after the blanks and the line number. If this position isn’t a period, then there’s no line number—at least, not in our format—so we don’t want to mess with it.

The if statement in lines 39 and 40 checks to see if the current line has a number in our standard format. If so, it deletes everything up to the current position (the period), plus two more characters, since the standard line numbers we put in with `numberLines` have two spaces after the period. If you change the way you number lines, you should make the corresponding change in `stripNumber` as you step through the line. Line 41 returns the string, which may have had a number prefix excised, or may be unchanged if it didn’t look just right.

Make Arrows Move Insertion Point if There's a Selection

Pity the poor arrow keys. What are they supposed to do? A stack is composed of lots of cards, so it seems only natural that the arrow keys, pointing forward and backward as they do, should move you to the next and previous cards. But lots of folks think the arrow keys should move the insertion point through a text selection, as they do in other applications.

HyperCard's solution is to provide a global property called `textArrows`. If `textArrows` is false, the arrow keys take you to new cards. When `textArrows` is true, the arrow keys move the insertion point through text, or you can type option-arrows to move from card to card.

This is a pretty good start, but a more elegant solution would be nice. For one thing, if `textArrows` is true and there's no selection or insertion point, pressing the arrow keys does nothing at all, which seems like a waste of good typing. With the tiny handler in this section, the arrow keys will move the cursor if there's an insertion point or selection, and they'll move through the stack otherwise. To get this handler to do its work, use the Cookbook Reference stack to install it. To learn more about how to do this, see Appendix A.

Inserts

When we say "insertion point or selection," we're actually being redundant. A plain insertion point is considered a selection. Sure, it's a selection that includes no characters, but it's a full-fledged selection anyway. So why do we say "insertion point or selection" in this section? We do it to be completely clear.

```

1.  on smartArrowKey
2.      -- script 7-3: Make the arrows move the cursor if
3.      -- there's an insertion point (or any selection).
4.
5.
6.      -- If there's an insertion point or selection, arrow
7.      -- keys move through text; otherwise, they move
8.      -- through cards.
9.      set textArrows to the selectedChunk is not empty
10. end smartArrowKey

```


Handler name:

`smartArrowKey`

What it does:

When you press the forward or backward arrow keys, this handler makes sure that you go from card to card, unless there's an insertion point or selection, in which case the insertion point moves through the text.

Parameters:

None. You won't need to call this handler directly, of course, because it's called when you press the arrow keys.

How it works:

A great little trick is used to implement this handler's functionality in just one line. The trick, found in line 9, calls the HyperTalk function `selectedChunk`, which tells what the selection is. If there is no selection, including no insertion point, the `selectedChunk` is empty, and line 9 sets `textArrows` to false, meaning that arrow keys will change cards. If there is a selection or insertion point, `textArrows` is set to true, and the arrow keys will move the insertion point.

After line 9 sets `textArrows` to the desired value, we're done. If you can see the elegance and sheer way-coolness in this handler, then you're becoming a true stackhead. Congratulations!

Recover Text from Any File

No matter how good your karma is, once in a while you'll have a document that suddenly refuses to be opened by its very own application. Sometimes, even HyperCard will inadvertently ruin a stack, and you may wind up losing some information.

What can you do when this happens? There are all sorts of file recovery tricks and bring-'em-back-alive software packages for the Macintosh. In this section we contribute a little handler that tries to pick out intelligible text from any file, assuming it had any in the first place. The result is not necessarily pretty, but it sure beats having to recreate an entire document from scratch. The neat thing about this handler is that it will work with any document, not just HyperCard stacks.


```

1.  on rescueText fileName,fieldName
2.    -- script 7-4: Recover text from any file.
3.
4.
5.    -- This handler opens any file and tries to read
6.    -- text from it. This is especially useful for dead
7.    -- stacks or other files that can't be read by their
8.    -- own application.
9.    if fieldName is empty then put "data" into fieldName
10.   open file fileName
11.   repeat
12.     doMenu "New Card"
13.     read from file fileName for 30000
14.     put it into field fieldName
15.     if it is empty then exit repeat
16.   end repeat
17.   close file fileName
18. end rescueText

```

Handler name:

rescueText

What it does:

This handler opens a file and reads characters (bytes) from it. It assumes the characters are text. If they're not, nothing will happen to the file, but you'll be looking at a lot of gibberish when you see what's been read from the file. Once you've read what's in the file, you can edit it and, hopefully, salvage some information.

Parameters:

RescueText takes two parameters. First is the name of the file you're going to try to read stuff from. The second parameter, `fieldName`, is the name of a background field used to hold the text it pulls out of the file. This parameter is optional, and the default field name "data" is plugged in if you don't say otherwise.

Mystery to me

When most applications save text into a file, lots of other information is mingled in that's not readable by us poor human beings. For example, a word processor has to save all the stuff about fonts, styles, and so on; a database program keeps track of records, fields, links, and other good things; a spreadsheet remembers facts about each cell other than just the text of its entry. So, when you use `rescueText`, you'll probably find lots of arcane garbage mixed in with your words. The only way to deal with this is to read through what you get and pick out the important information by hand.

How it works:

In line 9, we make sure that the user passed us a `fieldName` to hold the recovered text. Maybe the user was too much in shock about the damaged file to pass a parameter here. In that case, "data" is used as the field name, so you should be sure to have such a background field (or the field that you specify) before calling this handler.

Line 10 uses `HyperTalk` to open the file. The trick here is that `HyperTalk` isn't picky about what kind of files it will open—anything at all will do. It can be a stack, a word processing opus, a page-layout extravaganza, or a list of your family's shoe sizes written in Swahili and encoded in octal.

Lines 11 through 16 are a repeat loop that does the main work of this handler. Line 12 creates a new card to hold the data you're about to read. Line 13 does the actual reading, and it tries to grab up to 30,000 characters, the most text a field can hold. If it read this much, it tries again (but wait, there's more). Every time more stuff is read in, another new card is created and the new stuff is put into the specified field on that card. When there's no more to be read (line 15), the repeat loop ends, and line 16 closes the file before the handler ends.

Dead, dead, deadski

Every kind of data recovery has its limitations. Our technique, which is very primitive, will open any file that the Macintosh operating system will open. This can include slightly weird files that are basically OK, but contain bad stuff preventing their applications from opening them. For example, a word processor can accidentally trash some information in a file, making it impossible for the word processor to open it. The file would probably still be all right to the operating system, and `rescueText` could open it. But if you have a disk that doesn't show up on the desktop in the Finder, or one that your dog munched on, `rescueText` probably won't help much.

Shift Selection Left

The next couple of handlers are useful in helping you indent blocks of text—what HyperCard does with scripts. These handlers take the selection and move it to the left or the right by a specified number of spaces.

```
1.  on shiftLeft howMany
2.    -- script 7-5: Shift the selection left.
3.
4.
5.    -- This command moves the text in the selection
6.    -- to the left, deleting leading spaces.
7.
8.    if howMany is empty then put 1 into howMany
9.    get the selection
10.   repeat with i = 1 to the number of lines in it
11.     repeat howMany times
12.       if char 1 of line i of it is " "
13.         then delete char 1 of line i of it
14.     end repeat
15.   end repeat
16.   replaceSelection it
17. end shiftLeft
```


Handler name:

shiftLeft

What it does:

This handler shifts lines in the selection to the left, chopping off leading spaces, which is the same as removing indentation. If there aren't any leading spaces on a line, shiftLeft doesn't do anything to that line.

Parameters:

You can send one parameter to shiftLeft, which is the number of spaces to shift the text. This parameter is optional. If you don't bother to send it, the handler will assume you wanted to shift by one position.

How it works:

Line 8 is our familiar old technique of filling in a default value for a parameter that wasn't passed. In this case, we're saying we want to shift the text by one position if howMany wasn't passed by the caller. Line 9 puts the selection into "it."

Line 10 starts a repeat loop to examine each line of the selection, and line 11 creates another loop inside that one. The inner loop repeats once for every position that we're going to shift the text. Each time through the inner loop, we check to see if the first character of the line is a space (line 12), and if so, we chop it off (line 13).

When we drop out of the inner loop, we've shifted one line, and it's time to move on. When the outer loop finishes, it means we've checked the whole selection, and we can replace the selection with our new, shifty text (line 16). That's it. In the next section, you can read about how to shift the selection exactly the other way—to the right!

Shift Selection Right

If you want to get anywhere in the indentation business, you have to be able to move text to the right. This handler will take care of that for you. It's not exactly like the shift-left script in the preceding section, since there are some different problems to solve.


```
1. on shiftRight howMany
2.   -- script 7-6: Shift the selection right.
3.
4.
5.   -- This command moves the text in the selection
6.   -- to the right. What could be more fun?
7.
8.   if howMany is empty then put 1 into howMany
9.   put empty into spaces
10.  repeat howMany times
11.    put " " after spaces
12.  end repeat
13.  get the selection
14.  repeat with i = 1 to the number of lines in it
15.    put spaces before line i of it
16.  end repeat
17.  replaceSelection it
18. end shiftRight
```

Handler name:

shiftRight

What it does:

This handler indents (that is, shifts to the right) the selection by a specified number of character positions.

Parameters:

Like shiftLeft, this handler takes one parameter, which tells how many spaces to shift to the right. If you don't pass a value for this parameter, it moves the text one space over.

How it works:

Line 8 fills in the default value for howMany if it has to. The main job of this script, moving lines to the right, really means inserting spaces in front of each line. A variable called spaces will be used to help with this job, and it's initialized in line 9. Lines 10 through 12 set up spaces to contain one blank (space)

character for each character position that we're going to shift the line.

After building up spaces, we need to relax a little, so we grab the selection in line 13. In lines 14 through 16, we simply go through the selection, one line at a time, and insert the spaces before each line. Very smooth. Finally, line 17 replaces the old selection with the new, indented text, and we're done.

Take Returns Out of Selection

An interesting trick in text processing is to take some text from an unenlightened place that uses carriage returns to separate lines (like a primitive text editor), and throw away all the returns. Modern software, like virtually every Macintosh program that deals with text, supplies its own line wrapping without carriage returns. This handler will ruthlessly remove the carriage returns from the selection.

```

1.  on returnsOut
2.      -- script 7-7: Take the returns out of the
3.      -- selection.
4.
5.      -- This command is used to remove "carriage return"
6.      -- characters from the selection. Who's old enough
7.      -- to remember a real carriage return?
8.
9.      get the selection
10.     repeat while offset(return,it) <> 0
11.         put " " into char offset(return,it) of it
12.     end repeat
13.     replaceSelection it & return
14. end returnsOut

```

Handler name:

returnsOut

What it does:

This handler looks for returns and blasts them all away, replacing each return with a space character.

Parameters:

None.

How it works:

This one is pretty simple. In line 9, the selection is loaded into the variable "it." Line 10 starts a loop that will keep repeating until it can't find any more returns in the text. The operative statement in the loop, line 11, finds the first carriage return remaining in the text with this fragment

```
char offset (return, it) of it
```

and replaces it with a space

```
put " " into
```

like that. Line 13 replaces the selection with the new stuff and adds a single carriage return at the end, just to leave some space between the now-emasculated text and whatever comes next.

Put Returns into Selection

Taking returns out of the selection, as we did in `returnsOut`, is a pretty straightforward task that HyperTalk is perfectly well-equipped to handle. Let's say that we want to do the reverse: take some text with no returns in it, and add returns so that the text will be displayed correctly in a text-only application that doesn't automatically break words (most of these are on other, more primitive computers).

This problem is much trickier than taking out the returns. To know where to insert carriage returns, you have to know where a line of text wraps to the next line. HyperTalk's "line" chunk expression looks just for carriage returns. It doesn't know anything about where lines are broken in a field without carriage returns. What makes this hard is that Macintosh fonts have characters of different widths, and HyperTalk doesn't have any way of finding out how wide a particular character is.

When there's no way for HyperTalk to do something, the next step is to see if an external command or function (that's an XCMD or XFCN, stack fans) can help. Usually, it can, and that's the case here, too—otherwise, we wouldn't even mention it! The Macintosh Toolbox has routines to measure the width of

text characters for you, and we can get to them from an XFCN. To solve our problem, we've written an XFCN that does the hard work, and even some of the easy part, too.

```

1.  on returnsIn
2.      -- script 7-8: Put returns into the selection.
3.
4.
5.      -- This handler (and its XFCN) insert returns to
6.      -- format the text according to its font and field
7.      -- width.
8.      get width of the selectedField - 8
9.      if style of the selectedField is "scrolling"
10.     then subtract 16 from it -- account for scroll bar
11.     replaceSelection wordWrap(the selection,it, ↵
12.     textFont of the selectedField, ↵
13.     textSize of the selectedField, ↵
14.     textStyle of the selectedField)
15.  end returnsIn

```

Handler name:

returnsIn

What it does:

This handler works on the selection, which is assumed to be in a field. It inserts carriage returns at the end of each line as it appears on the screen. This doesn't change the appearance of the text in the field, because the text is already wrapped at the places where the returns are inserted. The returns themselves are invisible, of course. This trick is ideal when you need to know the end of a line on the screen, since HyperTalk's "line" chunk will tell where carriage returns occur.

Parameters:

None. The text to be "fixed" is the selection when you call this handler.

How it works:

Most of the hard stuff here is in the XFCN, but there's a short HyperTalk wrapper that sets everything up. Line 8 subtracts 8 from the field's width to allow for the margin. If the field has a scroll bar, line 9 subtracts 16 more pixels, because that's a scroll bar's width.

The call to our XFCN, named `wordWrap`, is buried in the next command, which also calls our `replaceSelection` handler. The call is too lengthy to fit on one line, so we've stretched it out (lines 11 through 14) and you can clearly see all the parameters to `wordWrap`.

Here's what's going on. `WordWrap` takes five parameters. The first parameter is the text to be formatted ("the selection"); next is the desired width of the formatted lines ("it"); then the font, size and style of the text being formatted ("textFont of the selectedField," and so on). `WordWrap` returns the newly-formatted text, and our handy `replaceSelection` script sticks it in the field. You can find the source code for the `wordWrap` XFCN in Appendix B.

Compare the Text in Two Fields

A common task in computing is to compare two big hunks of text that appear to be the same, just to see if they really are identical. This handler will examine the text in two fields to see if it's the same.

```

1.  on compare fieldName1,fieldName2
2.      -- script 7-9: Compare text in two fields.
3.
4.
5.      -- This handler compares two fields to see if
6.      -- the text they contain is the same.
7.
8.      put "Comparing..."
9.      put field fieldName1 into x
10.     repeat while last line of x is empty
11.         delete last line of x
12.     end repeat
13.     put field fieldName2 into y
14.     repeat while last line of y is empty

```



```

15.    delete last line of y
16.  end repeat
17.  repeat with i = 1 to max(number of lines in x,number
    of lines in y)
18.    set cursor to busy
19.    if line i of x <> line i of y then
20.      put "Fields differ at line " & i
21.      select line i of field fieldName2
22.      select line i of field fieldName1
23.      exit compare
24.    end if
25.  end repeat
26.  put "Fields are the same"
27. end compare

```

Handler name:

compare

What it does:

When you want to compare two fields or two runs of text, the computer can be a great assistant, since it won't miss anything due to blurry eyes or carelessness (just bugs). This handler provides an easy, automated way to compare text in two fields. It compares the fields line-by-line. As soon as it finds a difference, it reports the line number and quits. If it gets all the way through without finding a difference, it tells you that.

Parameters:

There are two parameters, and they provide the names of the fields that hold the text to be compared. The order doesn't matter.

How it works:

The handler starts off in line 8 by communicating to the user, saying what's going on, since the comparison process may take a while if there's a lot of text to look at. Line 9 obtains the text from the first field (note that it doesn't say "card field," so it must be a background field). Lines 10 through 12 get rid of any

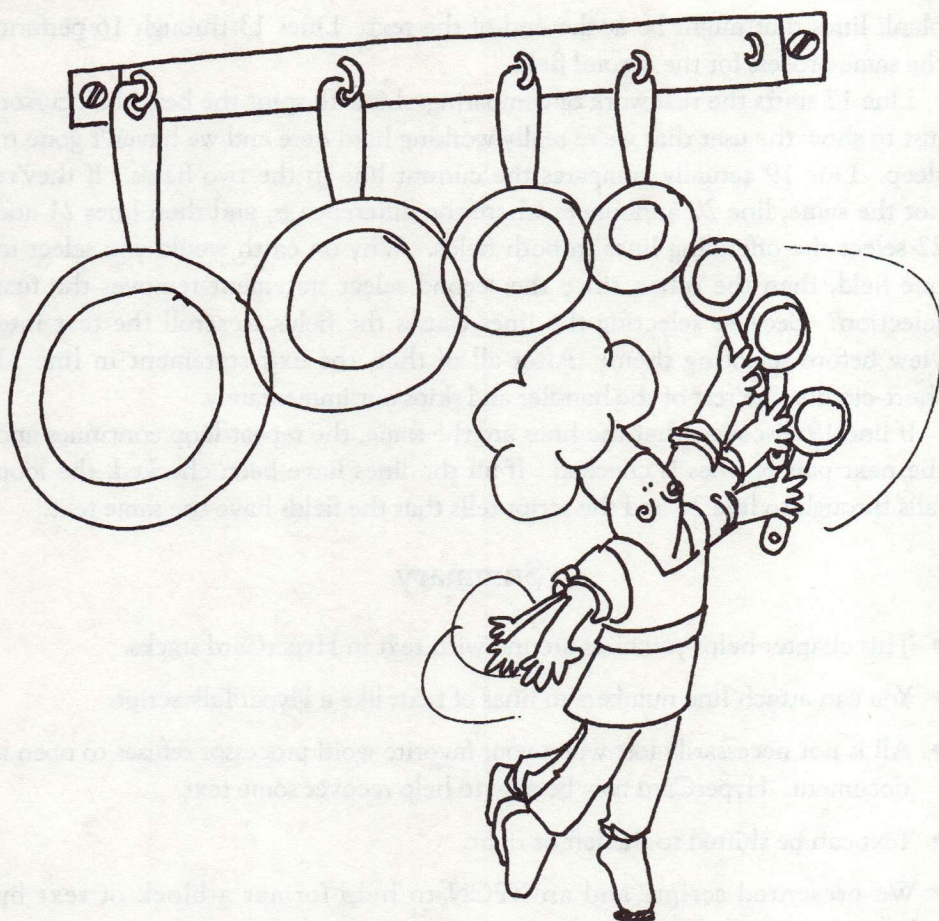
blank lines that might be at the end of the text. Lines 13 through 16 perform the same process for the second field.

Line 17 starts the real work of comparing. Line 18 spins the beachball cursor, just to show the user that we're really working hard here and we haven't gone to sleep. Line 19 actually compares the current line in the two fields. If they're not the same, line 20 announces where the difference is, and then lines 21 and 22 select the offending lines in both fields. Why on earth would you select in one field, then the other, since the second select statement removes the first selection? Because selecting the lines causes the fields to scroll the text into view before selecting them! After all of that, the exit statement in line 23 short-circuits the rest of the handler and skips out immediately.

If line 19 discovers that the lines are the same, the repeat loop continues and the next pair of lines is checked. If all the lines have been checked, the loop falls through to line 26 and the script tells that the fields have the same text.

Summary

- This chapter helps you mess around with text in HyperCard stacks.
- You can attach line numbers to lines of text, like a HyperTalk script.
- All is not necessarily lost when your favorite word processor refuses to open a document. HyperCard may be able to help recover some text.
- Text can be shifted to the left or right.
- We presented scripts and an XFCN to help format a block of text by indenting it and processing the carriage returns in it.





Chapter 8

GETTING ORGANIZED

HyperTalk provides you with a few tools to help you locate and change text in your stacks. In this chapter, we look at ways to make searching and replacing more powerful by adding onto the things that HyperCard already provides.

Post-process a Find Message

You've probably written handlers that intercept a built-in HyperTalk message and therefore get to handle it before HyperCard ever sees it. In this section, we talk about a technique for getting control of a message *after* HyperCard has already handled it.

1. on find
2. -- script 8-1: Post-process a Find message.
- 3.
- 4.
5. -- This handler demonstrates how to customize a
6. -- message "on the other side" - that is, after
7. -- HyperCard gets it.
8. send the params to HyperCard
9. if the result is empty then exit find
10. lock screen
11. push card
12. go to stack "My other address book"
13. send the params to HyperCard
14. if the result is empty then exit find
15. pop card
16. end find

Handler name:

find

What it does:

Every time you write a HyperTalk handler, you're creating a potential stopping place for a message that's riding down HyperTalk's inheritance path. For example, if you put a handler into a card script, and a message matching that handler arrives for the card, the handler gets control. Once your handler gets a message, and you're through with it, there are two basic ways to deal with the message: you can use the pass statement to allow the message to continue through the inheritance path, or you can stop the message from going any farther, which you do implicitly if you don't have a pass statement.

As we've discussed and as you probably know by now, you can write a handler to define a message of your very own invention, or you can write a handler to intercept a message that HyperCard already knows how to deal with, such as find, go, play, or anything else you want. By intercepting a message, doing something with it, then using pass, you can pre-process a command, then send it on to do its normal thing. For example, you might use the script in Figure 8-1 to pre-process the "About HyperCard..." menu command, putting up your own "About" box before passing the message down to let the real "About" box appear.

```
on doMenu what
  if what is "About HyperCard..."
    then answer "Ignore what follows. I wrote HyperCard."
    pass doMenu
end doMenu
```

Figure 8-1 A small handler

There's another, less obvious trick you can pull to mess around with the inheritance path. This trick, which we demonstrate in this section, lets you get control of a message after HyperCard has already handled it. This is accomplished by using the "send" keyword.

Parameters:

Find takes its usual parameters, as described in *HyperTalk 2.0: The Book*.

How it works:

The basic trick to these shenanigans is HyperTalk's "send" keyword. This keyword is used to send a message directly to a HyperCard object, without going through the usual inheritance path. Another important element here is the HyperTalk function "params," which returns the complete current message along with its parameter list to provide a convenient way for you to ship the message you're handling off to HyperCard, accomplished by the electronic poetry in line 8.

Now comes the fun part. After line 8 is executed, HyperCard has already handled the message, but control comes back to us right here in the handler, of course, since we sent the message! So, we can now continue through the rest of the handler and do things that take place after HyperCard is through with the message. In this case, we're interested in seeing if HyperCard found what it was after. If not, we're going to give it another place to look.

In line 9, we check to see if HyperCard found the string. We check by looking at HyperTalk's result function. If some kind of problem occurred, and the find failed, the result will not be empty. If the result is empty, the find worked, and we can just exit. If it's not empty, then the find didn't find anything, and we're going to try again. We're going to another stack, so line 10 hides the visual disturbances from the user. Line 11 remembers where we are, and line 12 jumps to the new stack to try the find again.

Once we're at the new stack, we need to try the find again, so line 13 repeats the magic incantation. Again (line 14) we check to see if it found anything. This time, however, if nothing was found, we're going to return to the card we came from (line 15) and then give up. If we had more stacks to search, we could tack them on here. In fact, if you wanted to, and if you were looking for something to do, you could probably make a general purpose handler that looks for something in a whole list of stacks by using this technique.

The catch

The post-processing technique we've described here by using "send the params to HyperCard" is not as generally useful as it might seem at first glance. The reason is that "the params" doesn't work well for all HyperTalk commands with parameters. It works OK for find, as you've already seen, but not for some other commands. For example, if you insert this handler into your card script

```
on add
  put the params -- to see what it looks like
  send the params to HyperCard
end add
```

and then type a command like "add 1 to field 1", you'll get this in the message box:

```
add
```

and you'll also get an error message from HyperCard, because "add" all by itself isn't enough. The reason this happens is because the implementation of the params function hasn't been completed yet in HyperCard. For some commands, you can't use "send the params to HyperCard." Of course, this situation is one that will improve over time as HyperTalk is enhanced. The best way to tell if "params" works for a message that you're interested in is just to try it. Use the "add" handler above, but substitute your favorite message and see what happens. In future versions, HyperTalk will do better with "params."

Search for Names and Nicknames

HyperCard's ability to find text in a hurry is great if you know exactly what you're looking for. In that way, it behaves just like a computer. It will do exactly what you tell it to do, no more or less. If you're looking for Ronald Reagan's phone number in your phone list, for example, and you happen to have the number listed under his old nickname, Dutch, you won't be able to find it. The handler in this section attempts to humanize HyperCard by teaching it about nicknames.

```

1.  on find how,what
2.      -- script 8-2: Search for names and nicknames.
3.
4.
5.      -- This handler searches for first names in various
6.      -- forms, using a nickname list that you supply.
7.
8.      send "find " & quote & what & quote to HyperCard --
          try it straight
9.      if the result is empty then exit find -- found
10.     put script of button "Sloppy Names" of card "Sloppy
          Search" into names
11.     if word 1 of what is not in names then return "not
          found"
12.     put offset(word 1 of what,names) into hitChar
13.     put hitChar into endChar
14.     repeat until char endChar of names is return
15.         add 1 to endChar
16.     end repeat
17.     put hitChar into startChar
18.     repeat until char startChar of names is return
19.         subtract 1 from startChar
20.     end repeat
21.     put char startChar + 1 to endChar - 1 of names into
          names
22.     repeat with i = 1 to the number of items in names

```



```

23.      put item i of names into word 1 of what
24.      put "find " & quote & what & quote
25.      send "find " & quote & what & quote to HyperCard
26.      if the result is empty then exit find
27.  end repeat
28.  return "not found"
29. end find

```

Handler name:

find

What it does:

This handler searches for all varieties of a first name, using a list of nicknames you create. The list of equivalent names should be kept somewhere in the script of a card button named “sloppy search” (of course, you can modify this handler if you want to put it somewhere else).

Parameters:

In this handler, you’re intercepting the find message, which takes two parameters. The first one tells what flavor of find it is: normal, whole, string, and so on. The second parameter, which is much more important for us in this script, tells what you’re looking for.

How it works:

Find works by not giving up (at least for a while). It tries the “find” command it gets, and if that doesn’t work, it assumes the first word of the thing you’re looking for is a first name, and keeps trying nicknames until it finds something or runs out of nicknames.

In line 8, HyperCard gets one chance to find the name exactly as specified by the caller. If it doesn’t find anything, the if in line 9 is false, and our nickname machine takes over. Line 10 grabs the button script that holds the name list. Right away, we check to see if our elusive name is in the list of names and nicknames (line 11). If it’s not, there’s no reason to continue, so we give up. Otherwise, we continue in line 12.

If the name we're looking for is in the list, here's our strategy. The name list is constructed so that each line is a comma-separated list of alternatives for the same name. Here's an example of a name list:

```
Bob, Robert, Bobby, Robby, Robbie, Rob
Barbara, Barb, Babs, Barbra
Bill, Billy, William, Will, Willy, Willie
Dan, Daniel, Danny, Danforth
George, Georgie, Poppy
```

When line 11 reveals that the name we're looking for is in the name list, we'll check all the other names on the same line, assuming they all refer to the same name. If we've checked all the names on a line and none of them find anything, we'll give up.

OK, let's get back to live action. Line 12 finds exactly where the name was found and records the location in `hitChar`. The little dance in lines 13 through 20 is designed to find the beginning and end of the line of names that we're interested in.

First, in lines 14 through 16, we look for a carriage return, which signifies the end of the set of alternatives for one name, searching forward from `hitChar` and recording the position of the carriage return in `endChar`. Then, lines 18 through 20 look backwards from `hitChar` for the carriage return that starts this line of names. After line 20, when we've located the carriage returns that bracket the line of names, we grab the actual names between the returns in line 21.

Now that we've retrieved the list of alternate names for the name we're working with, we can repeat the search, which is what the repeat loop in lines 22 through 27 does. Line 23 replaces the first name we searched for with an alternate name from the list, and lines 24 and 25 repeat the search. If the search succeeded this time, line 26 exits the handler, but otherwise we keep going. If none of the alternate names does the trick, we'll fall out the bottom of the loop, and line 28 will note that we didn't find anything. But boy, we sure tried.

Replacers

One of HyperCard's most revolutionary features is its fast searching for any text, anywhere. HyperCard users are always complaining about it, wishing it did even more, a sure sign of how neat and popular this feature is. (They wouldn't care so

much if they didn't like it already.) In this section, we'll present a few handlers to build on the find command and add various ways of replacing text after you've found it.

Déjà vu

In line 23, we get an alternate name from the list of names. On one pass through the loop, this won't be an alternate, but instead will be the name we already searched for and failed to find. Of course, the search will fail again. So why bother repeating the search? The alternative would be to save the original name earlier in the script, then insert an "if" statement after line 22 which would check to see if item i of names was the same as the original name. This "if" statement would be executed every time through the loop, probably slowing things down more than simply redoing the find for the original word. So, it makes more sense just to repeat the search.

Find and replace text everywhere

This handler adds a simple global search and replace capability to HyperCard. It will let you search for a particular string in a stack and replace all occurrences of it with another string.

Be very careful when you're using this command. It's very dangerous, because you can't undo the changes once you've changed everything. Watch out, or this handler will change your life.

```
1.  on replace old,new
2.    -- script 8-3: Find and replace text everywhere.
3.
4.
5.    -- This handler searches for text anywhere in a
6.    -- stack and replaces it with a new string.
7.
8.    repeat
9.      send "find string" & quote & old & quote to
        HyperCard -- no overriding
```



```
10.      if the result is not empty then exit replace --
          done
11.      do "put new into " & the foundChunk
12.  end repeat
13. end replace
```

Handler name:

replace

What it does:

This handler finds a given string and replaces it with another string. Replace is as relentless as a heat-seeking missile, searching for text in any field in a stack. Also, it doesn't care about word boundaries, so if you're searching for "tick," you'll get matches with "ticker" and "sticky fingers." Make sure you really want this level of replacement before you use this handler, or change "find string" to "find whole," which will find and replace only whole words that match your input string.

Parameters:

There are two parameters. The first is the string you want to search for (it's called "old"), and the second is the string that will replace all the occurrences of "old." Both parameters are required, unless you really want to mess your stack up good.

Forever and a day

The syntax used in line 8 does exactly the same thing as if we'd said

```
repeat forever
```

Although "repeat forever" sure looks like a lot more fun, just saying "repeat" gets the job done just as well.

How it works:

Replace has a repeat loop, starting in line 8, that goes forever, or until it doesn't find any more text, whichever comes first. In line 9, we ask HyperCard to find the old string. There are two notable things at work here. First, we're using "find string," a special flavor of "find" that finds exactly what we're looking for, while plain "find" only finds stuff if it's at the start of a word, at which point it goes ahead and selects the whole word anyway.

The second important thing in line 9 is to make sure we don't get stuck in any local varieties of "find" that may be in place in the current message path. To be sure that we get pure, unadulterated HyperCard find (the freshest kind), we use the "send" command. This bypasses the message path and goes straight to HyperCard.

In line 10, we check to see if we found anything. If not, our job is done, and we exit the handler. If we did find something, we have to blast the new string in its place. Line 11 does that by using HyperTalk's "foundChunk" function, which returns a chunk expression describing what "find" found. After that, the loop just keeps repeating until it's changed everything in sight.

Find and replace with prompting

This handler works a lot like replace, the one we just discussed, but it behaves a little more cautiously. It gives you a chance to think about each replacement before it actually changes anything.

```

1.  on replacePrompt old,new
2.    -- script 8-4:  Find and replace with prompting.
3.
4.
5.    -- This handler searches for text, asks for
6.    -- permission, then replaces it with a new string,
7.    -- if the user wants.
8.    repeat
9.      send "find string" & quote & old & quote to
        HyperCard -- no overriding
10.    if the result is not empty then exit replacePrompt
        -- done
11.    answer "Replace this one?" with "No" or "Yes"
```



```

12.      if it is "No" then next repeat
13.      do "put new into " & the foundChunk
14.      end repeat
15.      end replacePrompt

```

Handler name:

replacePrompt

What it does:

This handler searches high and low for text, just like `replace`, but instead of blindly changing every match it finds, it asks you, via an “answer” box, if you want to replace that particular occurrence. You can tell which one it’s talking about because it is in the box that “find” draws.

Parameters:

There are two parameters, just as in the `replace` handler. The first parameter contains the old text to search for, and the second one has the new text that’s supposed to replace it.

How it works:

Most of this handler works like `replace`, which we just described. In fact, the only difference comes in lines 11 and 12, right before the command that replaces text. After it finds the desired text, but before it does anything to it, line 11 puts up a dialog asking the user whether this occurrence of the text should be zapped (see Figure 8-2 for a picture of this lovely dialog). Based on which button the user clicks, line 12 decides whether to skip to the end of loop (if it is “No”) or continue on and clobber the text. Other than lines 11 and 12, you’ll find this handler identical to `replace`.

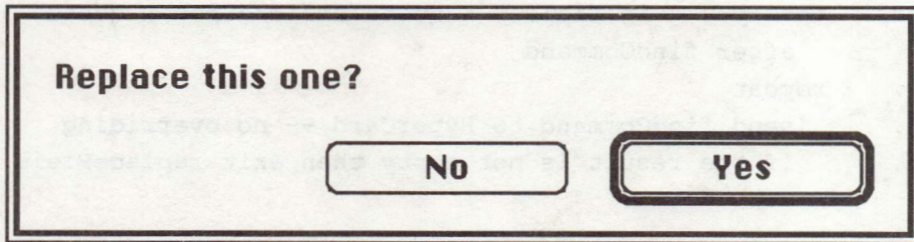


Figure 8-2 replacePrompt dialog

Get out of the way

When the answer box comes up and asks you whether you want to replace, you might not be able to see the text, because the dialog may be covering it up. There's no easy way around this, unfortunately, since you don't get to choose where the answer dialog goes—HyperCard always centers it for you. If you wanted to get very, very fancy, you could change the `replacePrompt` script to check the location of the field containing the `foundChunk`, then make sure it's somewhere where it won't be covered up by the answer dialog. Of course, you'd have to remember where the field really is and return it there when you're done. Good luck with this one—it could be a lot of fun!

Find and replace text in one field

Our next variation on the replacer theme is to limit our search to just one field. This is another way to limit the domain of our replacing so that we don't get burned by a runaway handler blasting text everywhere.

```
1. on replaceField old,new,fieldSpec
2.   -- script 8-5: Find and replace text in one field.
3.
4.
5.   -- This replacer works on text that's all in
6.   -- one field.
7.
8.   put "find string" & quote & old & quote into
       findCommand
9.   if fieldSpec is not empty
10.  then put " in field " & quote & fieldSpec & quote
       after findCommand
11.  repeat
12.    send findCommand to HyperCard -- no overriding
13.    if the result is not empty then exit replaceField
       -- done
```



```
14.      do "put new into " & the foundChunk
15.      end repeat
16.  end replaceField
```

Handler name:

replaceField

What it does:

This handler searches for and replaces text within a single background field. Even though it only searches in one field, it checks that field in all the cards in the stack. If your stack has multiple backgrounds, this will cause replaceField to do its thing in an unexpected place when it gets to the next background. If this isn't what you had in mind, you may want to check the current card or background each time through the loop to make sure you don't go out of bounds.

Parameters:

This handler has the familiar "old" and "new" parameters which contain the text. This time, there's also a third parameter, named fieldSpec, which contains the name of the field you want to search and replace in.

How it works:

In line 8, we construct a find command, using find string, as usual. Line 9 checks to make sure the caller passed in something for the field specification. If there is something there, it's tacked onto the end of the find command, and if fieldSpec is empty, the find command just doesn't change.

That's all that's novel in this handler. The rest of the script is just the same as in replace. The find command is sent directly to HyperCard (line 12), we check to see if we're done finding things (line 13), and if not, we replace the old text (line 14).

Replace selection with something else

Now it's time for us to talk about the most-used handler in the entire book: replaceSelection. This handler, which lets you put new text into the selection, is used in more of our command library scripts than any other handler. It's a

great example of a how a short, clever script can provide you with hours of productivity and enjoyment.

```
1.  on replaceSelection newSelection
2.    -- script 8-6: Replace the selection with something
3.    -- else.
4.
5.    -- This very handy handler installs some new text
6.    -- in place of the current selection.
7.
8.    put the selectedChunk into chunk
9.    if chunk is empty then exit replaceSelection --
        nothing selected
10.   put word 2 of chunk + length(newSelection) - 1 into
        word 4 of chunk
11.   put newSelection into the selection
12.   select chunk
13. end replaceSelection
```

Handler name:

replaceSelection

What it does:

This handler installs new text in place of the current selection. The text that's in the selection before you call replaceSelection is blown away, lost forever. The new text is selected after it's installed, in case you want to do something nasty to it, too.

Parameters:

There's one parameter, named newSelection. This holds the new text that's going to take the place of the current selection.

How it works:

This useful handler is pretty short and action-packed. First, we save the `selectedChunk`, which is an expression describing the current selection (line 8). We do a quick reality check in line 9 to make sure there's something selected, and we quit the handler if not.

The `selectedChunk` function always gives you a range of characters currently selected, like this:

```
char 23 to 37 of bkgnd field "Landing on Water"
```

We want the newly installed text to be selected when we're done, so we need to adjust the expression we've saved in `chunk` (back in line 8) to reflect the new selection. We know that the starting point of the new selection will be the same character as in the old selection. To figure the new ending point, we need to account for the length of the new text, add it to the starting point, and put it into word 4 of the `chunk` expression (count to word 4 and you'll see that it holds the location of the last character selected). All this fun stuff happens in line 10. It's all downhill from here. In line 11, we stick the new text into "the selection," which automagically replaces the old stuff with the new stuff. Line 12 uses our newly computed `chunk` expression to select the new text, and then it's all over.

Find the Selection

This handler provides a neat shortcut for finding text that's already selected on the current card. Since it operates on the selection, it requires that you use shift-tab to bring up the command box.

1. on findSelection
2. -- script 8-7: Find the selection.
- 3.
- 4.
5. -- This handler sees what the selection is
6. -- and then searches for it.
- 7.


```
8.    get the selection
9.    lock screen
10.   go to next card
11.   find it
12.   put "find " & quote & it & quote
13.   end findSelection
```

Handler name:

findSelection

What it does:

If you've got text selected in a field, findSelection searches for that text elsewhere in the stack. It also types another find command into the message box so that you can just press return to keep finding more occurrences of the same text.

Parameters:

None.

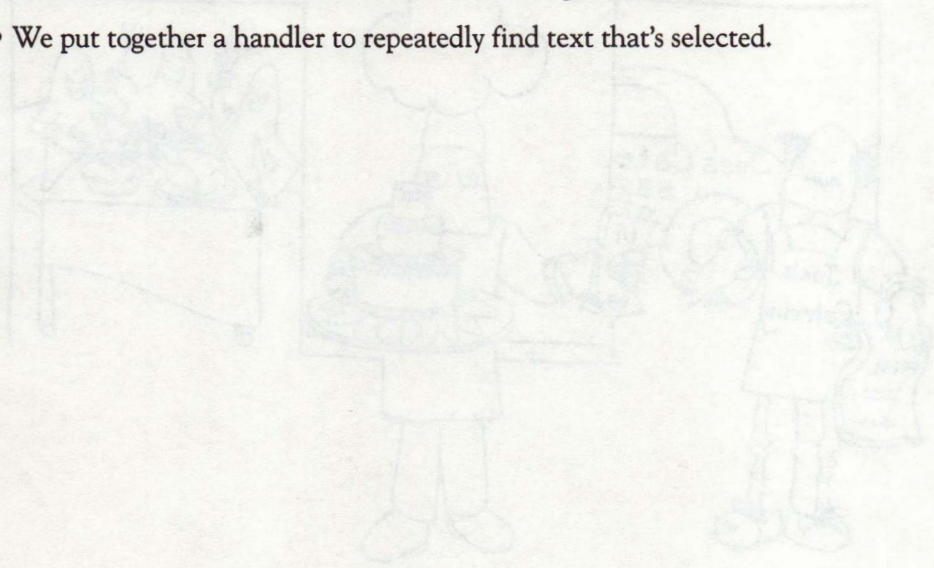
How it works:

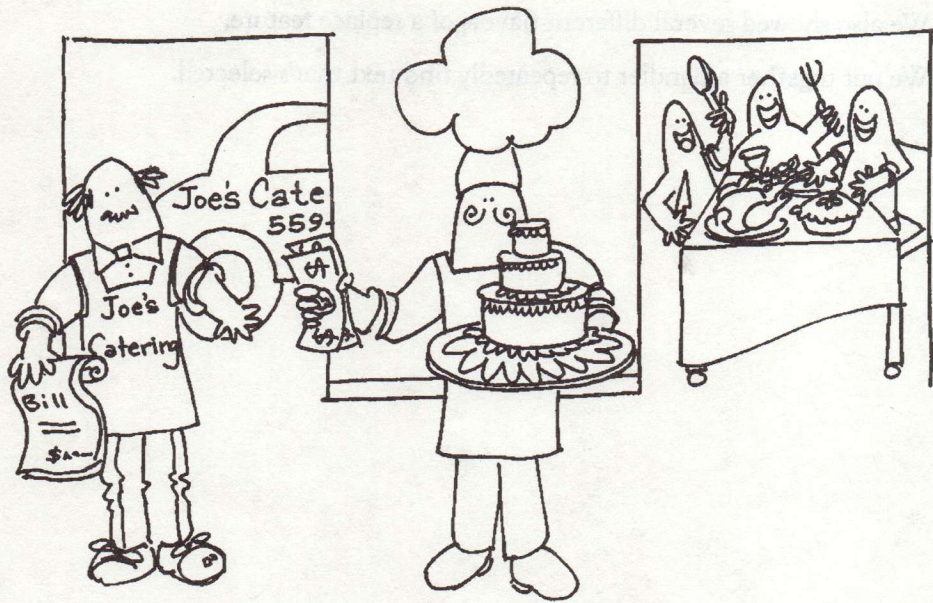
This handler begins by getting the selected text in line 8. If we immediately did a find, HyperCard would reward us by finding the text on the very same card, which is not what we want. So, line 9 locks the screen before line 10 advances to the next card to begin our search. Then, line 11 simply finds the text that line 8 grabbed. To finish things off, line 12 sticks the appropriate find command into the message box, which allows the user to press return to find again.

Summary

- This chapter introduces some commands that add onto HyperTalk's searching capability, and give it some ability to replace, too.
- We presented an interesting new variation on find, in which we actually let HyperCard do some finding, then look at what it did and see if we can improve on it.

- We also showed several different flavors of a replace feature.
- We put together a handler to repeatedly find text that's selected.







Chapter 9

READY-MADE SOLUTIONS

The handlers you've read about so far are all commands you might want to use in developing your own stacks, or when fishing through other peoples' stacks to see how they work. The scripts in this section are subtly different. They provide prepackaged ways of doing things that you might find useful in your stacks. Some of them, especially the graph-drawing scripts near the end of the chapter, are great guinea pigs for the HyperTalk debugger. By stepping through these scripts with the debugger, you can really get a feel for how they work.

Converting Between Coordinate Systems

One of the neatest features of the Macintosh is its ability to do graphics. When you're working with graphics, it's often handy to figure out where on the screen something is located. In HyperCard, as in most Macintosh applications, this is done by imagining a grid of numbered lines placed over the screen (if you try this at home, remember, it's an *imaginary* grid). Everything on the screen is located at an intersection of lines on this grid, called a coordinate plane. The numbers attached to the plane are called the coordinate system.

In HyperCard, there are actually two different coordinate systems at work. The first system uses the upper-left corner of the screen as location 0,0 (which, as you geometry fans know, is called the origin). This is known as the global coordinate system, since it extends across the whole world—well, the whole screen, anyway. HyperCard uses global coordinates for the card window rectangle property and the screenRect function.

For everything else, HyperCard uses a coordinate system based on the upper-left corner of the card window. This is how you refer to locations of buttons, fields, the message box, and everything else with a location property. This is important, because imagine what would happen if buttons and fields were in global (that is, screen-relative) coordinates. If you moved the card window, the

buttons and fields would stay where they were on the screen and would “move” within the card!

So, HyperCard has both global and local coordinate systems, and each one is useful for certain things. If you have a Macintosh Plus or Macintosh SE, the built-in screen is exactly the same size as the standard card window, so you probably won’t find yourself moving your card window around very much, as big-screen users do. Still, everything we’ve been talking about here applies to these Macintoshes, too.

Sometimes it’s convenient to be able to convert points or rectangles between the two systems, so we’ve developed some handlers for doing that, and here they come now!

Convert a rectangle to local coordinates

As we’ve discussed, HyperCard keeps some values in global coordinates. They’re measured in the distance from the top left corner of the screen. Other items are positioned in local, or card-relative coordinates, indicating that they’re computed from the top left corner of the card window, not the screen. This handler will take a rectangle given in global coordinates and convert it to its card-relative, local equivalent. We’ll find this handler useful in some of the other scripts later in this chapter.

```

1. function localRect rect
2.   -- script 9-1: Convert a rectangle to local
3.   -- coordinates.
4.
5.   -- This handler converts a rectangle in global
6.   -- coordinates to one in local (card
7.   -- window-relative) coordinates.
8.   subtract left of card window from item 1 of rect
9.   subtract left of card window from item 3 of rect
10.  subtract top of card window from item 2 of rect
11.  subtract top of card window from item 4 of rect
12.  return rect
13. end localRect

```


Handler name:

localRect

What it does:

This handler converts a rectangle in global coordinates to its equivalent in local coordinates. The top left corner of the local coordinate system (known to graphics enthusiasts everywhere as the origin) is defined by the top left corner of the card window.

Parameters:

LocalRect takes one parameter, which is actually four numbers separated by commas. These values indicate the left, top, right, and bottom (respectively, of course) global coordinates of the rectangle that's going to be converted.

How it works:

Converting between a local and a global coordinate system is actually very easy. The trick is simply to subtract the difference between the two systems from each coordinate, as we did in the xy handler back in Chapter 6. Line 8 adjusts the left and right coordinates of the rectangle to the local system by subtracting the value of the card window's left edge.

On the edge

Why doesn't this handler subtract the rectangle's left from the card's left and the rectangle's right from the card's right? The card's left edge tells how far away the card is from the edge of the screen. By subtracting this value from the rectangle's left and right, we're removing this screen-to-card difference from the rectangle's left and right coordinates. This leaves the card-to-rectangle difference, which is just what we want. The same logic applies to lines 10 and 11 (read on).

In lines 10 and 11, we use this technique to localize (there's a nifty word) the top and bottom coordinates of the rectangle that we passed in. Finally, line 12 returns the newly fixed rectangle as the function's result, and we're done.

Convert a point from local to global coordinates

This handler helps you convert a point in local coordinates to its equivalent in global coordinates.

```

1.  function globalPoint point
2.      -- script 9-2: Convert a point from local to global
3.      -- coordinates.
4.
5.      -- This handler takes a local (card-relative) point
6.      -- and converts it to global coordinates.
7.
8.      add left of card window to item 1 of point
9.      add top of card window to item 2 of point
10.     return point
11. end globalPoint

```

Handler name:

globalPoint

What it does:

This handler gives you a subroutine that converts a point in local coordinates to its equivalent location in global coordinates. Since HyperCard keeps track of all the relevant values, it's a pretty straightforward process, as you'll see.

Parameters:

GlobalPoint takes one parameter, which is the point you want to convert. It returns the global version of the point as the function result.

How it works:

This conversion is even easier than converting a rectangle, because there's only one point to worry about instead of two. In line 8, we add the card window's "left" property to the first item of the point—its horizontal coordinate. In line 9, we (surprise!) do the very same thing to the point's vertical coordinate, adding the card's "top" this time. Line 10 just returns the new point.

Rect on the highway

In lines 8 and 9, we get to use the neat and easy-to-remember syntax “left of card window” and “top of card window” to refer to the edges of that rectangle. But when we talk about the point, we have to use “item 1” and “item 2,” rather than “left” or “horizontal” or something else mnemonic. Why? In the case of the card window and other HyperCard objects with a rectangle property, HyperTalk lets you use left, top, right, and bottom because it knows that these things are rectangles. If you just happen to string some numbers together as items and call them a point or a rectangle, HyperTalk doesn’t know about that, and so you have to use the less clever “item 1” and “item 2” syntax, but stay tuned for future versions, as always.

Convert a mixed list to an array of items

You’ll find that HyperTalk’s nifty chunk expression syntax is very useful for working with text when you write your scripts. HyperTalk gives you several options for dealing with text in chunks. You can look for text separated by commas (these are called items), spaces (these are words), or returns (lines). You can run into trouble if you’re trying to work with some text that’s a mix of all of these things, though. Consider the following text as one list of things:

```
fly, Lydia, Juno
Beetle-snake
Day-O
one two three
```

If we consider every space, comma, and return as a separator between items, this list contains eight items. In this form, it would be pretty tough to pull out those items and process them. If we could put all eight of these items into a standard HyperTalk item list, it would look like this:

fly, Lydia, Juno, Beetle-snake, Day-O, one, two, three

In this section, you’ll find a script that transforms the mixed list of things into the nicely comma-separated list that you can access with HyperTalk’s “item” chunk type.


```

1. function itemArray values
2.   -- script 9-3: Convert a mixed list to an array of
3.   -- items.
4.
5.   -- This function takes a list that can include
6.   -- multiple lines, items, and words, and homogenizes
7.   -- it into items.
8.   put empty into array
9.   repeat with i = 1 to the number of lines in values
10.    put line i of values into valueLine
11.    repeat with j = 1 to the number of items in
        valueLine
12.      put item j of valueLine into valueItem
13.      repeat with k = 1 to the number of words in
            valueItem
14.        put word k of valueItem & "," after array
15.      end repeat
16.    end repeat
17.  end repeat
18.  delete last char of array -- strip trailing comma
19.  return array
20. end itemArray

```

Handler name:

itemArray

What it does:

This handler accepts a string that may consist of multiple items, words, and lines. It transforms this string into a list of comma-separated items.

Parameters:

ItemArray takes one parameter, which is the string of values to be processed into a list of items.

How it works:

This handler works by repeatedly searching for values, first as lines, then items, and finally words. The strategy here is this: grab a line at a time. Within each line, grab an item at a time (that means all the text up to the next comma). After grabbing an item, search through it for words. This whole trick is done with a three-deep nesting of repeat loops.

ItemArray builds the nice comma-separated list in a variable called array, initialized in line 8. The searching for items is accomplished deep within the three-level loop running from line 9 to line 17.

The outermost loop, which searches for lines, begins in line 9 and executes once for each line in the input parameter. Line 10 grabs the current line and prepares for the next level of searching, which looks for items (line 11). Now we're in the second level of the loop (line 12) and we're pulling out one item at a time. Finally, in the innermost loop, which starts in line 13, we take one word at a time and put it into array, with a comma after it (line 14).

After line 14, which actually transfers one word at a time from the input to the output, we just need to back out of the three levels of loop that we've dug into. We do this with the three consecutive "end repeat" statements in lines 15 through 17.

When we're all done, we'll have an extra comma at the end of array, since line 14 always tacks on a comma. Line 18 snips it off. That's the end of the real work. All that's left is to ship the new list off as the function result (line 19), and the handler's done.

Sometimes it's useful to transform a set of text values into a group of lines, not comma-separated items. The script in the next section takes care of that for you.

Convert a mixed list to an array of lines

This handler works a lot like itemArray, the script we just described. It takes a mixed input list (some might call such a list heterogeneous) and turns it into a list of lines, each one separated by a return.

1. function lineArray values
2. -- script 9-4: Convert a mixed list to an array
3. -- of lines.
- 4.


```

5.    -- This function takes a list that can include
6.    -- multiple lines, items, and words, and homogenizes
7.    -- it into lines.
8.    put empty into array
9.    repeat with i = 1 to the number of lines in values
10.   put line i of values into valueLine
11.   repeat with j = 1 to the number of items in
      valueLine
12.   put item j of valueLine into valueItem
13.   repeat with k = 1 to the number of words in
      valueItem
14.   put word k of valueItem & return after array
15.   end repeat
16. end repeat
17. end repeat
18. return array
19. end lineArray

```

Handler name:

lineArray

What it does:

LineArray is a function that takes a mixed list of words, items, and lines as input, and produces a list of lines as output. Every word, item, or line in the input is turned into a line in the output.

Parameters:

LineArray takes one parameter, values, which contains the mixed list of text that's going to get fed through our homogenizer and turned into lines.

How it works:

This handler is virtually a clone of the one we just described, itemArray. That's not surprising, since they do pretty much the same thing. There are only three differences, and here's what they are:

1. LineArray is named lineArray, and itemArray is named itemArray (well, it's a difference).
2. In line 14, lineArray puts the current word and a return into array, the function result; in itemArray, line 14 puts the current word and a comma into array.
3. In itemArray, line 18 takes out the extra comma that winds up at the end of the output array. LineArray produces an extra return at the end of its output, but we just leave it alone. This causes the script for lineArray to be one line shorter, by the way.

That's it. They're absolute twins otherwise. It might be a fun rainy-day project to try to combine them together, with a parameter to determine which kind of list to produce.

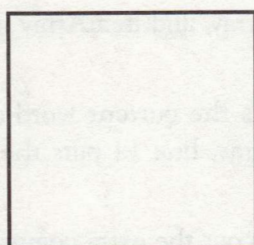
Inset or outset a rectangle

Our next subroutine handler, suitable for inclusion in your favorite scripts, helps you change the size of rectangles easily. This script is so handy that we use it ourselves in a few of the handlers you'll see later in this chapter.

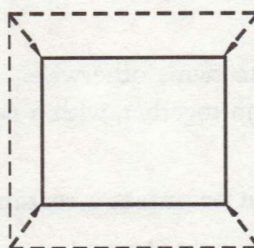
```

1.  function insetRect rect,insetAmount
2.      -- script 9-5:  Inset or outset a rectangle.
3.
4.
5.      -- This handler resizes a rectangle by the
6.      -- amount indicated.
7.
8.      if insetAmount is empty then put 10 into insetAmount
9.      add insetAmount to item 1 of rect
10.     add insetAmount to item 2 of rect
11.     subtract insetAmount from item 3 of rect
12.     subtract insetAmount from item 4 of rect
13.     return rect
14. end insetRect

```

Before



After

Figure 9-1 Inset a rectangle

Handler name:

insetRect

What it does:

Given a rectangle and a number, this handler resizes the rectangle by adding the value to its top and left edges, moving them down and in, and subtracting the inset amount from its bottom and right edges, which moves them up and in (see Figure 9-1). Note that you can “outset” the rectangle (a fancy non-word meaning “make it bigger”) by passing negative values.

Parameters:

The first parameter, `rect`, is the rectangle you want to shrink or stretch. The second parameter, `insetAmount`, tells how many pixels you want each edge to move. The second parameter is optional. If you don’t use it, `insetRect` will use 10 as the inset amount.

How it works:

This script starts off with the familiar check to see if an optional parameter has been passed (line 8), and if not, the default value is filled in. After that, there are four lines of pure arithmetic. The `insetAmount` parameter is added to the left and top coordinates of the rectangle, and then it's subtracted from the right and bottom coordinates. Line 13 then returns the modified rectangle, and the handler can go back to sleep.

Create a calendar

The wise Macintosh knows how we humans work with months, days, and years (at least according to the Gregorian calendar, the one most of the world uses). In this section, we get a little help from HyperCard in constructing calendars for any month we want. We'll supply the dates, but you have to add your own pretty pictures for each month.

```

1.  function calendar month,year
2.      -- script 9-6:  Create a calendar.
3.
4.
5.      -- This function creates a calendar, given a
6.      -- month and year.
7.
8.      put "Sun Mon Tue Wed Thu Fri Sat" into dayHeader
9.      convert (month & " 1, " & year) to long date
10.     put it into currentDate
11.     delete word 1 of it
12.     delete word 2 of it
13.     repeat with i = 1 to (length(dayHeader)-length(it))
14.         div 2
15.         put " " before it
16.     end repeat
17.     put it into monthHeader
18.     convert currentDate to dateItems
19.     put item 2 of currentDate into monthNum
20.     put empty into days
    repeat with dayNum = 1 to (item 7 of currentDate) - 1

```



```

21.      put "      " after days
22.  end repeat
23.  repeat while item 2 of currentDate is monthNum
24.      get " " & item 3 of currentDate & " "
25.      if length(it) is 3 then get " " & it
26.      if item 7 of currentDate is 7 then get it & return
27.      put it after days
28.      add 1 to item 3 of currentDate
29.      convert currentDate to seconds -- to force next
        line to reconvert
30.      convert currentDate to dateItems
31.  end repeat
32.  return monthHeader & return & dayHeader & return &
    days & return
33. end calendar

```

Handler name:

calendar

What it does:

This function takes a month and a year, and returns a lovely string that consists of the name of the month at the top, and the days below in standard one-week-across calendar format. The calendar looks best if it's displayed in a monospaced font, like Monaco or Courier, as in Figure 9-2.

```

      August 1980
Sun Mon Tue Wed Thu Fri Sat
                        1  2
  3   4   5   6   7   8   9
10  11  12  13  14  15  16
17  18  19  20  21  22  23
24  25  26  27  28  29  30
31

```

Figure 9-2 Month produced by calendar script

Parameters:

There are two parameters, the month and the year that you want for the calendar. You can specify the month with its full name or the standard abbreviation, which is the first three letters of the month's name. Note that you can't use a number to specify the month. You'd have to use "August" or "Aug," not "8," if you want August, for example. You should specify the year fully—use "1980," not just "80."

How it works:

This script is fairly long and intriguing, so get ready for some fun stuff. As always, you may want to follow along with the HyperTalk debugger. The first part of the handler, all the way through line 16, is used to create the nice headings that label the calendar with the month, year, and names of the days of the week. First, we get the names of the days together in line 8. These names are also conveniently isolated here in case you need to translate them to another language. Next, we're going to make sure that we've got the full name of the desired month.

Lines 9 through 16 have the job of creating the "August 1980" heading that's centered nicely at the top of the calendar. Line 9 takes the month and year that were passed in and sticks a "1" and a comma between them, indicating the first day of that month. If we're asking for August 1980, we'd get a string like this:

Aug 1, 1980.

This happens to be HyperTalk's abbreviated date format, and line 9 uses the "convert" command to change it to long date format,

Friday, August 1, 1980.

Now we've got the full month name, as well as the day of the week, and we store it away in line 10. The only reason we have to go through this charade is to make sure that we have the full name of the month.

Now that we've gone through all that to get the long date, we can throw a lot of it away. The first thing in the elongated date is the day of the week, which we don't need, so line 11 removes it. We also don't need the "1" that signifies the first day of the month and it's taken out in line 12.

On the side

Line 9 uses an obscure but powerful alternate form of `convert`. In the usual syntax of `convert`, the date you want to convert is in a variable, and it looks something like this:

```
convert firstDate to long date
```

This command actually changes `firstDate` itself. In programming terms, this is called converting `firstDate` *in place*. What if you have a date that's not in a variable and you want to change its format, as we've done in `calendar`? One solution is to put it into a variable first, but HyperTalk defines something more convenient. If you call `convert` with an expression, not a container, the converted form will appear in our old friend "it." So, a statement like this:

```
convert "12/26/59" to long date
```

will put the converted date into "it," and we don't have to bother creating an unneeded variable. This is another handy way to make your scripts just a little smaller and faster.

Lines 13 through 16 may look hairy, but all they do is make sure the month-year title line is centered across the days. It works by first computing the difference in lengths between the day names and the month-year name (`length(dayHeader)-length(it)`), then divides that by 2 to find the center point (`div 2`). This result tells how many spaces to put in front of the month-year title in order to center it above the day names.

Now that all the fancy titling jazz is over, we can concentrate on the calendar itself, although formatting it will require a little fancy footwork, too. The first interesting task is to make the first week of the month look good. The first week is tricky, since it probably doesn't begin on a Sunday, so we have to know which column (that is, which day of the week) to start with. Lines 17 to 22 handle this task.

Chicken delete

Lines 11 and 12 delete the name of the day of the week and the "1," but there's a little subtlety going on that you might not see at first. We start with the long date, in this form:

Friday, August 1, 1980

We want to get rid of the "Friday," and the "1," parts. We'd really like to command HyperTalk to "delete word 1 and word 3 of it," but unfortunately, there's no "and" in chunk expressions, so we have to do two separate deletions. First, we chop off word 1 in line 11, which leaves this:

August 1, 1980

Now we just have to delete word 3 and....wait a minute! Word 3 isn't "1," any more! Because we deleted the first word, the old word 3 has moved up and is now word 2, as indicated in line 12. Situations like this can easily be the source of bugs in your scripts, so watch for them.

First, we convert the date to the dateItems form (line 17), which turns each part of the date, including the day of the week, into a number. Line 18 retrieves the month number from the dateItems list, and line 19 initializes the variable that's going to hold the string of day numbers with the spaces and returns in the right places.

The little loop in lines 20 through 22 formats the first week of the month. The technique will be to add spaces to the beginning of the string of days until it's padded to the correct starting day for the first week. For example, Figure 9-2 shows that August 1980 starts on a Friday, so we need to add enough spaces to cover the positions for Monday through Thursday.

This technique uses item 7 of the date, which tells the day of the week. Sunday is day 1, Monday is 2, and so on through Saturday, which is 7. If the month starts on a Sunday (day 1 of the week), we don't need any padding at all. If it's Monday (day 2), we pad for one day, Tuesday (day 3) requires two days of padding, and so on. This means the formula for padding is:


```
(item 7 of currentDate) - 1
```

which is the upper limit of the repeat loop in lines 20 through 22. Coincidence? I don't think so. In line 21, we add four spaces of padding for each pass through the loop, since four spaces is the width of one day on the calendar.

After we're past line 22, we've taken care of padding the day numbers to make the first week start in the right place. All that's left is to tack the day numbers onto the "days" variable. This happens in the repeat loop in lines 23 through 31, the heart of this function.

This loop has a couple of jobs. It has to spit out the number of each day of the month, at the same time it lines up the day numbers on their right sides (they're right-justified, in other words), as you see in Figure 9-2. Then, it has to continue through the month, making sure it stops when the month is over. Here's how that all happens.

The loop begins in line 23 by saying it will continue until the date item indicates the month has changed. This works because we stashed the original month's number in the monthNum variable. Line 24 puts the current day's number into "it," sandwiched between two spaces. If the current day has only one digit (if it's earlier than the 10th day of the month), we have to pad it with another space in front (line 25), because each day number occupies four spaces.

Every time we're about to emit a Saturday, we're at the end of a line in the calendar, so we need to add a carriage return. Line 26 handles this chore, and line 27 tacks the whole line onto the "days" variable. Tomorrow is another day, so our next task is to make currentDate refer to the next day. Doing this requires a little work.

At this point, currentDate is expressed as date items. There's no way to say "add 1 day to dateItems." The closest we can get is to add 1 to item 3, which is the item in dateItems that holds the date. This will work as long as it's not the last day of the month. In other words, if currentDate is August 31, adding 1 to item 3 will make it, uh, August 32.

Although this date is bogus, HyperTalk's convert command doesn't mind. Convert knows how many days are in each month, including leap years, and it will fix August 32, turning it into the more reasonable September 1. Convert performs this fix whenever we convert to any format *except* dateItems. So, if we convert currentDate to another format, then right back to dateItems, the end-of-the-month overrun will be fixed.

Time enough at last

The ability to overrun the end of the month and have HyperTalk fix it is actually a very powerful feature. It's especially useful for figuring out the date for a number of days past an event. For example, to figure out which day is the 197th day of 1992, you could write

```
convert "Jan 197, 1992" to date
```

You would then find "7/15/92" in the variable "it," telling you that the 197th day of 1992 is July 15th. When did your son, born on October 18, 1985, turn 1000 days old?

```
convert "October 1018, 1985" to date
```

will tell you that the magic day was July 14th, 1988. Gee, those first thousand days sure went by fast.

The back-and-forth conversions in lines 29 and 30 will make sure that we'll be OK when we try to move past the last day of the month. Line 31 ends the repeat loop. Remember that we get out of this loop when the month changes, as specified in line 23. As soon as we move past the last day of the month, the loop will be over, and we'll have our entire month.

Line 32, where we return the calendar as the function result, is like a curtain call for all the stars of our handler. First is monthHeader, which is the name of the month plus the year, centered, of course, followed by a carriage return. Next comes dayHeader, the names of the days of the week, and another carriage return. Last and certainly not least are the day numbers, kept in the "days" variable, and one last carriage return. That's it. If you made it this far, congratulations!

Erase all or part of card or background picture

When you're writing scripts to do fancy drawing on the screen, you may find the need for some macro-style scripts that do things easily done when operating HyperCard by hand, but are a little trickier when done from a handler. This

section introduces one of those kinds of scripts, a routine to help clear part of the screen.

```

1.  on clearPicture frame
2.      -- script 9-7: Erase all or part of the card or
3.      -- background picture.
4.
5.      -- This handler clears out any part of the picture.
6.      -- It works for card and background, depending
7.      -- on editBkgnd.
8.      if frame is empty then put localRect(rect of card
          window) into frame
9.      put item 1 to 2 of frame into topLeft
10.     put item 3 to 4 of frame into botRight
11.     choose select tool
12.     drag from topLeft to botRight
13.     doMenu "Clear Picture"
14.     choose browse tool
15. end clearPicture

```

Handler name:

clearPicture

What it does:

This subroutine handler clears part of the card or background picture. It doesn't automatically select between card and background layers, but it relies entirely on the current setting of editBkgnd to tell whether the card or background picture will be zapped.

Parameters:

ClearPicture takes one parameter, named frame. This is a rectangle which gives the local coordinates of the area to be erased, and is optional. If you don't pass it, the entire picture will be erased. That'll teach you.

How it works:

This handler works by stringing together a few menu items and HyperTalk commands into a nice, automated sequence that can be called from other scripts. The most interesting point of discussion is probably what happens in line 8, where we fill in the default value if the parameter wasn't passed.

When the parameter, frame, is empty, we want to erase the entire picture. You might think we could just use the card window's rectangle property to set the frame, but we can't. The card window's rectangle is in global coordinates—it gives the distances from the upper left corner of the screen. The area we want to erase must be specified in local coordinates—the distance from the upper left corner of the card window—because we're going to call "drag" to select the rectangle, and drag expects local coordinates. So, we have to call our utility function localRect to make the conversion. See "Converting between coordinate systems" earlier in this chapter for more details.

After line 8, everything else is pretty straightforward. We load the upper left and lower right corners of the rectangle into variables (lines 9 and 10). Line 11 gets ready to drag by choosing the select tool, the one that looks like a rectangular theatre marquee.

In line 12, we tell a phantom mouse to drag across the rectangle we've specified, using HyperTalk's drag command (you can just picture a phantom mouse with a cute little mask, capturing a female mouse and playing the organ for her. Then she creeps up from behind and removes his mask, revealing...). Line 13 clears the picture by calling on a menu item, and line 14 goes back to the browse tool before the script returns.

Calculate the center of a rectangle

Finding the center of a rectangle is a handy graphic function. Rectangles are used a lot on the Macintosh, not just for their basic geometric wonderfulness, but also to specify boundaries of things, like windows, buttons and fields. The utility script in this section is handy for dealing with rectangles.

```
1. function centerOfRect rect
2.   -- script 9-8: Calculate the center of a rectangle.
3.
4.
5.   -- This little utility script is used to find the
```



```

6.      -- point that's at the center of a rectangle.
7.
8.      put item 3 of rect - item 1 of rect into width
9.      put item 4 of rect - item 2 of rect into height
10.     return round(item 1 of rect + width/2) & "," &
        round(item 2 of rect + height/2)

```

Handler name:

centerOfRect

What it does:

This function gives you the coordinates of the point at the center of a rectangle.

Parameters:

There's one parameter, a rectangle, known to the world only as "rect." The centerOfRect function will return the center of this rectangle as a list of two items, with the x-coordinate first.

How it works:

This little handler has only three things to do. First, it figures the width of the rectangle by subtracting its left edge from its right edge (line 8). Then, in the same way, it computes the height by subtracting the top from the bottom (line 9). Then, it computes the horizontal and vertical centers in the math-heavy statement in line 10. Here's how it figures the center.

First, for the horizontal center, also called the x-coordinate, it adds half the rectangle's width ("width/2") to the left edge ("item 1 of rect") and rounds off the result. To calculate the vertical center, or y-coordinate, it adds half the height ("height/2") to the top of the rectangle ("item 2 of rect") and again rounds off. Finally, it places a comma between these values and ships the whole thing off as the function's result.

Sum a list of numbers

There are all sorts of math-type utilities you might need when creating stacks. For example, lots of stacks need to add up lists of numbers. This section has a script which does that and lets you be flexible about how the numbers are passed

to it (OK, so there's not that many exciting things you can do with a script for summing numbers, but we're sure giving it our best shot).

```

1.  function sum
2.      -- script 9-9: Sum a list of numbers.
3.
4.
5.      -- This function adds up a bunch of numbers.
6.      -- They can be separate parameters, or an item list.
7.
8.      put 0 into total
9.      repeat with paramNum = 1 to the paramCount
10.         put param(paramNum) into list
11.         repeat with i = 1 to the number of items in list
12.            add item i of list to total
13.         end repeat
14.     end repeat
15.     return total
16. end sum

```

Handler name:

sum

What it does:

This function adds up numbers and returns their sum. The interesting thing is that you can pass any number of values as separate parameters and each parameter can be a comma-separated list of numbers. Sum will find all the individual values and add them up correctly.

Parameters:

Although there aren't any parameters declared, through the magic of HyperTalk's param function, you can actually have as many as you want. For more about how this trick works, you can read the script tip called "Parameters" in Chapter 3.

How it works:

As we said earlier, each parameter to sum can be a single value or a list of values separated by commas. So, the main work of this function is to figure out where all the numbers are and add them together. In line 8, we initialize a variable to hold the running total of the values.

Line 9 starts a repeat loop that will examine each parameter to the function. It does this by using HyperTalk's paramCount function, which returns the number of parameters that were passed. Then, we put the current parameter into the "list" variable (line 10). This statement uses HyperTalk's "param" function, which returns whichever parameter that you ask for.

After line 10 puts a parameter into "list," we need to find out if it really was a list of items or just one value. This detective work happens in lines 11 through 13. This loop works because a single value behaves exactly the same as a list of one item. For example, if you ask HyperTalk for

```
item 1 of "Madness"
```

it will return "Madness," not really caring that there's only one "item" in the whole thing. So, when line 12 looks for an item within "list," it will work whether "list" is really several items or just one. Either way, it finds a value and adds it to the running total.

After we get out of the repeat loops, we've summed the entire list of values into the "total" variable. Line 15 returns total as the function result, and we're all done.

Drawing Charts and Graphs

One of HyperTalk's most intriguing features is its ability to take control of HyperCard's paint tools. Many new HyperCard users are surprised and delighted to find out that a HyperTalk script can simulate most of the painting things you can do directly without a script, and there's nothing more fun than a surprised, delighted new HyperCard user.

In this section, we present scripts that take advantage of HyperTalk's scriptable painting ability. These scripts can be used to draw bar charts, line charts, and pie charts from your scripts. In addition, they've got lots of interesting techniques you can steal for your own programmed painting masterpieces. Because the scripts in this section are so graphical, it's a lot of fun to step through them

with the HyperTalk debugger. The debugger lets you see each dot and line as it's being drawn, so you can really follow the action at your own pace.

Draw a bar chart

When people see the neat graphics that a Macintosh can do, drawing charts seems like a natural ability. In this handler, we'll present a script to help you draw a bar chart anywhere on the screen. Figure 9-3 shows you an example of a chart drawn by this handler.

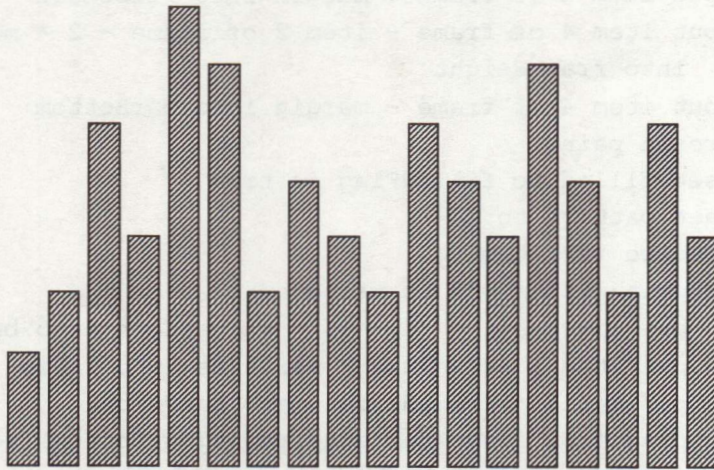


Figure 9-3 Sample bar chart

```

1. on barChart values, filledFlag, frame
2.   -- script 9-10: Draw a bar chart.
3.
4.
5.   -- This handler draws a bar chart for you when
6.   -- you pass it some values.
7.
8.   if frame is empty
9.     then put insetRect(localRect(rect of card window),
10.      10) into frame
10.  clearPicture frame

```



```

11.    put itemArray(values) into values -- convert to
        comma separated list
12.    put max(values) into maxValue
13.    put number of items in values into valueCount
14.    put item 3 of frame - item 1 of frame into
        frameWidth
15.    put frameWidth/valueCount into valueWidth
16.    put 3/4 * valueWidth into barWidth
17.    put (valueWidth-barWidth)/2 into margin
18.    put item 1 of frame + margin into firstLeft
19.    put item 4 of frame - item 2 of frame - 2 * margin
        into frameHeight
20.    put item 4 of frame - margin into barBottom
21.    reset paint
22.    set filled to filledFlag is true
23.    set pattern to 14
24.    choose rect tool
25.    repeat with i = 1 to valueCount
26.        put firstLeft + (i - 1) * valueWidth into barLeft
27.        put item i of values / maxValue into ratio
28.        put ratio * frameHeight into barHeight
29.        drag from round(barLeft),round(barBottom) to -
30.        round(barLeft + barWidth),round(barBottom -
            barHeight)
31.    end repeat
32.    choose browse tool
33.    end barChart

```

Handler name:

barChart

What it does:

This handler draws a bar chart. You specify the values to be graphed (the y-axis values), your choice of hollow or filled bars, and where you want the chart drawn. The bars are automatically spaced for you along the x-axis, and the scaling is automatically computed, too.

Parameters:

The first parameter holds the values you want to graph. The values can be separated by any HyperTalk chunk separator you choose—spaces, commas, returns, or any combination of these. The second parameter, `filledFlag`, is used to tell whether you want sparse hollow bars or gaudy filled ones. The third parameter gives the rectangle that will enclose the chart. This last parameter is optional. If you don't pass it, the entire card window, inset by 10 pixels, is used for the chart.

How it works:

This handler relies on some of the utility scripts we've been describing so far in this chapter. First, line 8 checks to see if the caller passed a value for "frame," the parameter that tells where to put the chart. If it wasn't passed, the default value of the card window inset by 10 is supplied. Next, we clear the way for the chart we're about to draw (line 10). In line 11, we make sure that the values we're going to chart are in a nice comma-separated list.

To figure out the scaling for the bars, we need to know the largest value of any bar we're going to draw. Line 12 accomplishes this with HyperTalk's `max` function, and line 13 records the number of values along with the number of bars we're going to draw. Line 14 figures out the width of our frame, and line 15 computes how much space will be allotted to each bar in `valueWidth`.

There has to be some white space between bars, of course, and we've arbitrarily decided that the bars will take up three-fourths of the space in each position, with the white space taking the other fourth. So, we set up a variable called `barWidth` to tell how wide each bar will be (line 16). We need to put equal amounts of white space before and after each bar, so line 17 takes the total white space (`valueWidth - barWidth`) and puts half of it into a variable called `margin`.

We need to compute a couple more values before we start blasting out bars. We set up the location where the first bar will begin by adding the left edge of the frame to the white-space margin (line 18), then calculate the maximum height of a bar by subtracting the top of the frame from the bottom and allowing for the margin (line 19). We determine the baseline for the bars by subtracting the margin from the bottom of the frame (line 20).

Now we're set—almost. There are a few more housekeeping chores that have to be done. Line 21 resets the paint tools to their defaults, to avoid any embar-

passing problems that might result from things like having “centered” set to true, or having a wacky line size. Line 22 sets up the filled property according to the filledFlag parameter, and line 23 chooses the modest pattern 14, which is diagonal stripes. Our last preparation is line 24, which gets ready to draw rectangles.

Finally, we get to the repeat loop in lines 25 through 31, where all the bar drawing is done. Each pass through the loop will draw one bar. Line 26 figures out where to start drawing the current bar. Each bar takes up “valueWidth” spaces, as we said earlier. Line 27 computes the height of the current bar by dividing the current value by the maximum bar value and putting the result into a variable called ratio.

Now that we know how high the current bar is supposed to be, we need to scale it to the frameHeight that we’ve figured out (line 28). Finally, the long drag statement in lines 29 and 30 actually draws the bar, dragging from the bottom left corner of the bar (barLeft, barBottom) to the top right corner (barLeft + barWidth, barBottom - barHeight). There’s a “round” in front of each value to make sure all the numbers are integers, which the drag command insists on.

Going up?

To draw the bar, we dragged from the bottom-left corner to the top-right corner. You may have noticed that the top-right corner specification looks a little weird:

```
barLeft + barWidth, barBottom - barHeight
```

Adding the left edge to the width to get the right edge makes sense, but why do we *subtract* the height from the bottom to get the top? The answer, as you may have guessed, is that HyperCard’s coordinates increase from the top of the screen to the bottom, so the top of the bar actually has a lower-numbered coordinate than the bottom. This coordinate system comes from QuickDraw, the Macintosh’s built-in graphics package, created by a guy named Bill Atkinson.

In this script, we've truly saved the best for last. After the drag command takes care of drawing the bar, we're pretty much done. The repeat loop that draws all the bars ends in line 31. The only other thing the handler does is choose the browse tool, and then it's finished. But its beautiful handiwork remains on the screen.

There are lots of places where you could change or customize this script. Right now, it's hard-coded to draw using pattern 14, and you could make that a parameter if you wanted. You might also introduce more control over scaling, and you could add some axis-labeling if you really wanted to soup it up.

Draw a line chart

In this section, we'll describe a handler that draws line charts. It uses a lot of the same tricks and techniques as barChart, but it solves some new problems, too. You can see an example of a chart produced by this handler in Figure 9-4.

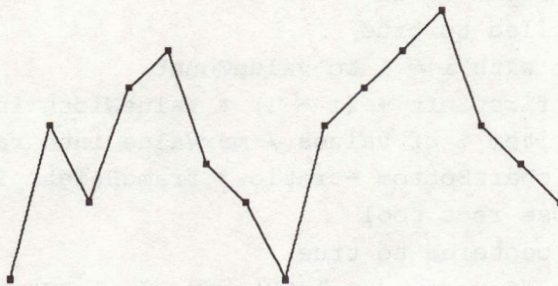


Figure 9-4 Sample line chart

```

1. on lineChart values, frame
2.   -- script 9-11: Draw a line chart.
3.
4.
5.   -- This handler draws a line chart for you when
6.   -- you pass it some values.
7.
8.   if frame is empty
9.   then put insetRect(localRect(rect of card window),
10.    10) into frame

```



```

10.  clearPicture frame
11.  put itemArray(values) into values -- convert to
      comma separated list
12.  put max(values) into maxValue
13.  put number of items in values into valueCount
14.  put item 3 of frame - item 1 of frame into
      frameWidth
15.  put frameWidth/valueCount into valueWidth
16.  put valueWidth/2 into margin
17.  put item 1 of frame + margin into firstLeft
18.  put item 4 of frame - item 2 of frame - 2 * margin
      into frameHeight
19.  put item 4 of frame - margin into chartBottom
20.  put 2 into pointSize
21.  reset paint
22.  set pattern to 12
23.  set filled to true
24.  repeat with i = 1 to valueCount
25.      put firstLeft + (i - 1) * valueWidth into pointH
26.      put item i of values / maxValue into ratio
27.      put chartBottom - ratio * frameHeight into pointV
28.      choose rect tool
29.      set centered to true
30.      drag from round(pointH),round(pointV) to ↵
31.      round(pointH)+pointSize,round(pointV)+pointSize
32.      if i is not 1 then
33.          choose line tool
34.          set centered to false
35.          drag from round(oldH),round(oldV) to
              round(pointH),round(pointV)
36.      end if
37.      put pointH into oldH
38.      put pointV into oldV
39.  end repeat
40.  choose browse tool
41.  end lineChart

```


Handler name:

lineChart

What it does:

This handler draws a line chart. You pass it the values you want to graph, and the card location where you want the graph to appear.

Parameters:

The first parameter, `values`, contains the numbers you want to graph. As in `barChart`, the values can be separated by spaces, commas, lines, or any combination of those three, and the handler will sort it all out. The second parameter is optional and gives the coordinates to be used for displaying the graph. If it's empty, the whole card will be used. Again, this is just how it works in `barChart`.

How it works:

A lot of the initial setup for this handler is the same as in `barChart`, so we'll be pretty brief about describing those lines. In particular, lines 8 through 15 are identical to `barChart`. These lines are responsible for computing the frame, clearing the area on the screen, making sure the values are in a comma-separated list, and calculating `maxValue`, `valueCount`, `frameWidth`, and `valueWidth`. To review how these lines work, take a look at the "How it works" section for `barChart`, the previous handler.

In `barChart`, we had to allow for white space between bars. We don't have to do that this time, since the line will be continuous across the chart. This accounts for an extra line (line 16) in `barChart`. After that line, the next four are the same. They set up some variables: `margin`, the space around the edges of the chart; `firstLeft`, the location of the start of the line; `frameHeight`, the highest possible location on the line; and `chartBottom`, the baseline of the chart. Again, there's more on these in the `barChart` description if you're interested.

The rest of the handler, which is more directly involved in actually drawing our work of art, is pretty different from `barChart`. Line 20 sets up a variable that holds the size of one edge of a point on the graph. If you want bigger points, you can change this. We reset the paint settings (line 21), choose our pattern, and

turn on “filled,” which makes the points solid boxes. You can change this if you’d rather have hollow boxes or a different pattern for points.

Line 24 starts the loop that draws each data point and a line segment. Line 25 determines the horizontal location for the current point, and line 26 figures out how high the current point should be, just as in `barChart`. We compute the vertical coordinate for the point in line 27 by multiplying the ratio by the maximum height, then subtracting from the chart’s bottom (remember we subtract here because lower-numbered y-coordinates are higher on the screen).

Lines 28 and 29 get us ready to draw a data point, and the `drag` command in lines 30 and 31 actually draws the little bitty rectangle with the same technique used for drawing bars in the `barChart` handler. But this time around, drawing the rectangle is only half the battle. We also have to draw the line segments to connect the data points. Let’s talk about that problem for a minute.

We have several choices about drawing the line segments. The most obvious is probably to draw a data point, then the line segment that connects it to the next point. This is a little tough because when drawing a point, we haven’t done all the calculation for the next point yet, and don’t know where it’s supposed to be.

Another idea is to run through a loop and pre-calculate all the points. Then, we can loop through again and actually do the drawing of points and lines. This idea would work, but it requires us to loop through and process each point twice. There must be a better way...

And there is. We’ll calculate and plot each point then save it in a variable. The next time through the loop, after we’ve plotted the new point, we’ll draw the line segment from the “old” point (the previous one) to the new one just plotted. Great! All we have to do is special-case for the first point, since there won’t be an “old” point or a line segment to draw. This is all handled in lines 32 through 36.

On the first pass through the drawing loop, the “if” statement in line 32 will be false and we’ll skip ahead to line 37, after the “end if.” Lines 37 and 38 record the current point as “oldH” and “oldV” before the handler ends. When we pass through the drawing loop after the first time, the “if” in line 32 will be true. This will set us up for normal line drawing (lines 33 and 34) and the line will be drawn from the previous point (in `oldH`, `oldV`) to the current one (this all happens in line 35). By drawing just a point on the first time through, then a line segment and a new point on every other pass, our strategy takes care of

drawing the entire chart with just one pass through the data. After we're done drawing the chart, we clean up a little by selecting the browse tool, as usual.

Draw a pie chart

For our next and last charting trick, we're going to draw a pie chart. To whet your appetite, there's a picture of one in Figure 9-5.

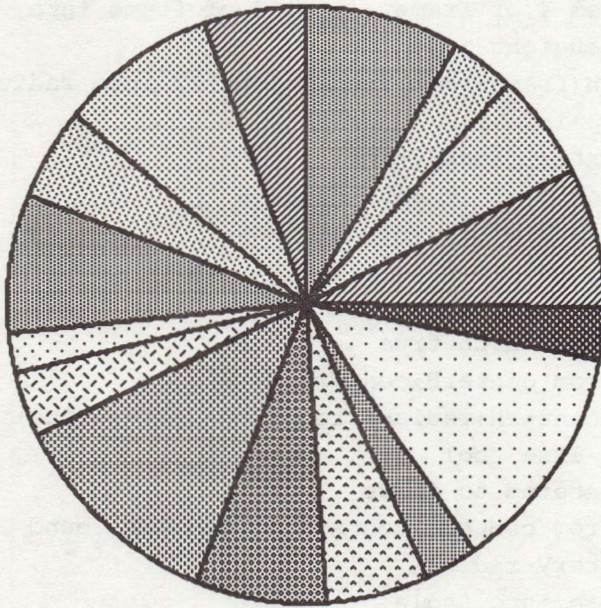


Figure 9-5 Sample pie chart

```

1. on pieChart values, frame
2.   -- script 9-12: Draw a pie chart.
3.
4.
5.   -- This handler draws a pie chart for you when
6.   -- you pass it some values. Yummy.
7.
8.   if frame is empty
9.   then put insetRect(localRect(rect of card window),
      10) into frame

```



```
10. clearPicture frame
11. put itemArray(values) into values -- convert to
    comma separated list
12. put sum(values) into total
13. put number of items in values into valueCount
14. put item 3 of frame - item 1 of frame into
    frameWidth
15. put item 4 of frame - item 2 of frame into
    frameHeight
16. put min(frameWidth,frameHeight)/2 into radius
17. reset paint
18. get centerOfRect(frame)
19. put item 1 of it into centerH
20. put item 2 of it into centerV
21. choose oval tool
22. set lineSize to 2
23. set centered to true
24. drag from centerH,centerV to -
25. round(centerH+radius),round(centerV+radius)
26. choose line tool
27. set centered to false
28. drag from centerH,centerV to centerH,round
    (centerV-radius)
29. put zero into angle
30. repeat with i = 1 to valueCount
31.     put pi * item i of values / total into halfChange
32.     add halfChange to angle
33.     put centerH + round((radius-3) * sin(angle)) into
        halfH
34.     put centerV - round((radius-3) * cos(angle))
        into halfV
35.     add halfChange to angle
36.     put centerH + round(radius * sin(angle)) into h
37.     put centerV - round(radius * cos(angle)) into v
38.     choose line tool
39.     drag from centerH,centerV to h,v
```



```

40.      if halfChange > .01 then -- don't try to fill very
        small wedges
41.      set pattern to item (i mod 12)+1 of
        "2,13,11,3,14,6,21,17,31,18,4,32"
42.      choose bucket tool
43.      click at halfH, halfV
44.      end if
45.      end repeat
46.      choose browse tool
47.      end pieChart

```

Handler name:

pieChart

What it does:

This handler draws a pie chart from data that you pass it, and I promise, no pie jokes. It works a lot like the other chart handlers, automatically scaling for you, and allowing flexibility in the format of its parameters. Well, maybe just a few pie jokes.

Parameters:

PieChart takes two parameters and they work the same as lineChart. The first one is the list of values, separated by spaces, commas, or returns. The second is a frame, and if you don't pass a frame, one will be appointed for you—the card window's rectangle.

How it works:

PieChart does some things just like barChart and lineChart, but the drawing technique is radically different, of course. Lines 8 through 11, which set up and clear the frame and make sure the values are comma-separated, are identical to the other chart handlers. Line 12 adds up all the values, which it needs to do to compute the size of each pie slice, and line 13 records the total number of slices. Lines 14 and 15 compute the width and the height of the area that'll enclose the pie.

Next, we have to figure the radius of the pie. To do this, we first determine which is smaller, the width or the height of the frame, then take half that value as

the radius (line 16). We have to use the smaller value so that the whole pie will fit on the screen. We reset the paint tools (line 17), find the center of the frame (line 18), and put its coordinates into `centerH` and `centerV` (lines 19 and 20).

As we get closer to the magic moment when we get to draw, line 21 selects the oval tool to make the pie itself. We select a nice line width for the pie (line 22) and set up to draw the pie from the center (line 23). Lines 24 and 25 bake the pie by dragging from its center to a point on the circle. By now, we've got the pie drawn and its time to start thinking about the slices.

To do the lines, we begin by selecting the line tool (line 26), then turning off centered drawing (line 27) to get normal lines for the slice edges. Line 28 draws the first edge of the first slice, which always extends from the center of the pie straight up to a point on the circle at angle zero.

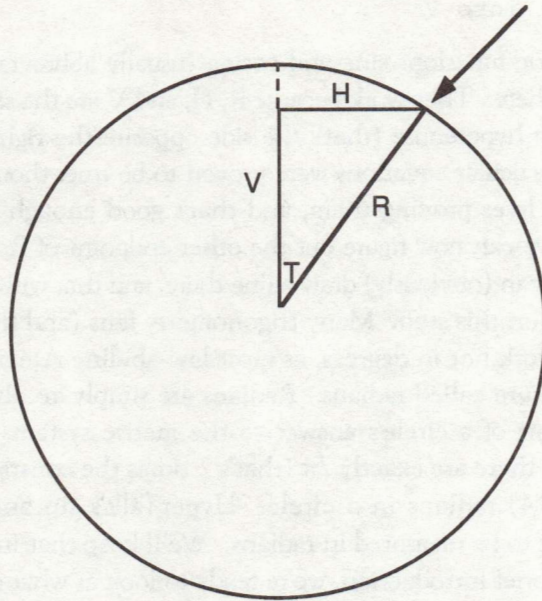
Now we get to the fun part of the script—where the slices are actually drawn. Because of the intricacy of this particular script, we're going to do something here that we didn't do for any other handler in this book: we'll provide two descriptions of what happens in lines 30 through 39, and you can choose which one to read.

Description #1: The slices are drawn with some magical trigonometry stuff and it all comes out right. Please skip the next twelve paragraphs.

Description #2: Let's take a look at what we're trying to figure out. We've drawn a circle, and we want to divide it up into sections that correspond to their share of the pie. So, for example, if one value is a fourth of the total, its slice should be a fourth of the pie. We know all the values that will share the pie (they're in the "values" variable) and we've added them together in the "total" variable. We can now determine what percentage of the pie should be taken up by each value by dividing the value by the total of all the values. This operation will tell us the percentage of each slice, but *doesn't* tell us where to draw the edges of each slice.

Take a look at Figure 9-6. Let's say that the line labeled *R* is the line that we want to draw to close the first slice. This line extends from the center of the circle to a point on the circle itself (the one that that gaudy arrow points to). We already know where the center is (it's at `centerH`, `centerV`), and we know the length of *R* (it's the radius of the circle, which we already stored in the "radius" variable). We also know the size of the angle *T*, since we know that the

whole circle measures 360 degrees, and we know what fraction of the circle is consumed by this slice.



$$H = R \sin T$$

$$V = R \cos T$$

Figure 9-6 Fun with trigonometry

What if we knew the lengths of the lines marked H and V? If we added H to the center point's horizontal location (in centerH) and subtracted V from the center's vertical location (in centerV), we'd get the coordinates of the point the arrow indicates. How can we figure out the length of H and V?

Aha! Trigonometry to the rescue! We take advantage of some well-known relationships among angles, circles, and triangles (at least, they're well-known among folks who are into this stuff). Anyway, trig gives us the following equations, using the labels from Figure 9-6:

$$H = R \sin T$$

$$V = R \cos T$$

Translated into HyperTalk, we can say

```
put R * sin(T) into H
put R * cos(T) into V
```

The trigonometric functions sine and cosine (usually abbreviated sin and cos) perform the magic here. They work because R, H, and V are the sides of a right triangle, and R is the hypotenuse (that's the side opposite the right angle). Never mind how these particular equations were proven to be true; thousands of ancient Greeks gave their lives proving them, and that's good enough for me. What's important is that we can now figure out the other endpoint of R. Once we know both ends of R, we can (obviously) draw a line there, and that will define our slice.

One more twist on this stuff. Many trigonometry fans (and there are millions of 'em) prefer to work not in degrees, as most law-abiding Americans do, but in a cute unit of measure called radians. Radians are simply an alternative way to measure angles, sort of a circle's answer to the metric system. There are 360 degrees in a circle; there are exactly 2π (that's 2 times the constant pi, or 2 times approximately 3.14) radians in a circle. HyperTalk's sin and cos functions expect their angles to be measured in radians. We'll keep that in mind.

Now, with that brief introduction, we're ready to look at what actually happens in lines 30 through 40 (yes, this is still Description #2). Line 30 starts a repeat loop that will go once for each pie slice to be drawn. Our strategy for each slice will be like this: compute a point that lies somewhere within the slice, then save that point and use it later as the place to click with the paint bucket to paint a pattern. Then figure the size of the slice, and draw the line that separates it from the next slice. Finally, fill the slice with a pattern.

How can we easily figure a point that lies somewhere in the slice? Well, we can figure the slice's angle by dividing the current value (which is "item i of values") by the "total" variable, then multiplying that result by $2 * \pi$. So, one convenient way to find a point in the slice is to multiply by pi itself, which will figure an angle half the size of the slice, and that's what happens in line 31. Line 32 updates angle to reflect this new value.

To finish computing the point that's somewhere in the slice, we need to apply our sin and cos formulas, which we do in lines 33 and 34. To ensure that the resulting point (in halfH, halfV) really is in the slice and not on the circle itself, we shorten the radius a little by subtracting 3 pixels in this computation.

Next, we want to calculate the full angle of the slice, so we add the other half

of the angle (line 35) and then figure the coordinates of the point on the circle by using our sin and cos formulas again (lines 36 and 37). By the way, notice that we always use addition for horizontal values (lines 33 and 36), and subtraction for vertical values (lines 34 and 37) because QuickDraw x-coordinates increase from left to right, but y-coordinates decrease from bottom to top. See the “Going Up?” note earlier in this chapter for more information about that.

Finally, we’re ready to draw the line on this sucker. Line 38 gets the correct tool, and line 39 actually draws it by dragging from the center to the smartly-computed point on the circle. At last, the slice is drawn. (End of Description #2. Aren’t you glad you read through that?)

After getting through line 39, we’ve drawn the bottom edge of the slice and we’re ready to fill it in with a pattern. First, we check to see if the slice is really tiny (line 40), and if so, we don’t bother to try to fill it. Otherwise, line 41 cleverly selects from among a rotating batch of patterns by using HyperTalk’s mod function, which gives the remainder as if you were dividing. The numbers listed correspond to patterns that match the decor in Dan’s home.

After choosing the pattern, line 42 selects the paint bucket and line 43 dumps it at the location we figured out earlier. Line 45 ends the repeat loop begun way back in line 30. Line 46 returns us to the familiar world of the browse tool, and that’s it. Somehow, it’s fitting that we should use pi to draw our pie chart, isn’t it?

Miscellaneous Solutions

Finally, we present our last clever handlers that can’t be categorized anywhere else.

A field that indents automatically

In this section, we’ll take a look at a field script that maintains your indentation from line to line. This is a feature commonly used in outline processors, editors for programming languages, and other arcane places.

```

1.  on returnInField
2.      -- script 9-13:  A field that indents automatically.
3.
4.
5.      -- This handler makes a field indent its text
6.      -- automatically.
7.
```



```

8.    get the value of the selectedLine
9.    put empty into indent
10.   repeat with i = 1 to the number of chars in it
11.       if char i of it is " " then put " " after indent
12.       else exit repeat
13.   end repeat
14.   send returnInField to HyperCard
15.   put indent after the selection
16. end returnInField

```

Handler name:

returnInField

What it does:

This is the handler that implements automatic indenting in a field.

Parameters

None.

How it works:

When this handler is called, the insertion point is located at the end of the line that you (or your user) just finished typing. That line is important because it will determine how much to indent the line that follows it. Line 8 grabs the line into "it" so we can look at it. Line 9 initializes the variable we'll be using to tell how much to indent.

In lines 10 through 13, we figure out just how much the line was indented. This works by counting the space characters (blanks) that come at the beginning of the line before the first non-blank (line 11). When a non-blank is found, we get out of the repeat loop (line 12).

By the time we've gotten through the repeat loop and past line 13, the variable called indent holds the correct number of spaces to indent the new line. There's only one problem: we've intercepted the return, so there is no new line yet. If we use "pass" to let the returnInField continue on to HyperCard, we'll lose control and won't get it back.

The solution is to explicitly send a `returnInField` message off to HyperCard, which is just what we do in line 14, strangely enough. When HyperCard gets the message, it inserts a return character into the field, creating a new line, just as God intended `returnInField` to do before we started messing with it. However, after HyperCard inserts the new line, our handler regains control in the next line! At this point, we simply add the indentation in line 15, and we're done.

Summary

- Sometimes, you have to convert between HyperCard's global and local coordinate systems in order to compare things correctly.
- We provided a way to homogenize different kinds of chunk arrays so that they're easier to deal with.
- You can resize a rectangle by inseting or outseting its corners.
- There's a script that will produce a nice calendar for any month of your choice.
- We introduced some scripts that draw charts and graphs, and a few handlers to provide utility functions used by the charting scripts. We refrained from overdoing the pie jokes.
- There's a script to make a field automatically indent each new line, as in an outline.

Dinner Break!

```

on make what
  -- Robin's Spaghetti!
  if what is "spaghetti" then
    put "Kitchen:Cupboard left of stove:Third
      shelf:Ragu"↵
    into JarName
    open container JarName
    put getPot(saucePan) into sink
    set faucet to "open"
    wait until the container is halfFull
    set faucet to "closed"
    put getPot(stockPot) onto stove burner 4
    set heat of stove burner 4 to "high"
    put "kitchen:cupboard left of stove:second
      shelf:pasta"↵
    into pkgName
    open container pkgName
    put value of pkgName into pot stockPot
    wait until (the sauce is "hot") and (the
      noodles are "al dente")
    put strained(noodles) into plate
    put plate onto "Kitchen:table"
  end if
end make

```


ABOUT THE STACKS ON THE DISK

Everything we've talked about up to now has described our library of command scripts and ready-made subroutines that you can use in creating your own stacks. In this chapter, you'll read about some complete stacks that we've included on the disk. These stacks, like the commands in the command library, were designed to demonstrate a slightly different flavor than more typical HyperCard stacks you've probably seen. We hope you get some ideas from their scripts and have fun along the way.

In this short section, we give you a quick look at the stacks you're getting on the disk and tell you enough to get started. But we're not going to take away your fun of exploring by giving thorough instructions on their use. To find out everything they do, just use them!

The Command Library

If you've read the first nine chapters of this book, you've already heard about the command library. This is the place that holds all the handlers we've been talking about. The command library actually consists of three stacks, and here's what you'll find in them.

The first one, named Cookbook Reference, is probably the only one you'll ever open. Each card in the stack describes one or more of the handlers in the command library. The handlers themselves are kept in the stack scripts of the two other command library stacks, named Cookbook Scripts 1 and Cookbook Scripts 2. When you want to look at the scripts while browsing the reference stack, it grabs handlers from the other stacks as necessary. There are two stacks of scripts because we could only jam 32K bytes of text into a single stack script (that's a HyperCard limit, friend).

You should never have to open the cookbook scripts stacks directly. The command library uses HyperCard's start using stack command to insert these stacks into the inheritance path. This allows any messages to seek out the

appropriate handlers in their stack scripts, and everything is cool. For a thorough discussion of the command library architecture and how to use it, see Appendix A.

You'll find some handlers that have "bonus" as their number. These are scripts that were cooked up too late to get into the book, but not too late to be useful to you, and so we included them on the disk.

Some command library features, such as automatic stack compaction and windoid tracking, aren't accessed by typing a command. Instead, these features do their thing when the appropriate message is sent. For example, stack compaction happens when a stack is closed (`closeStack` message), and windoid tracking takes place whenever a window is moved (`idle` message).

Because these features happen automatically, we've provided an easy way to turn them on and off. Each automatic feature's card in Cookbook Reference contains a check box named "Enabled," which turns that feature on and off.

Some automatic features won't get in your way if you decide not to use them. For example, the shorthand system (described in Chapter 2) won't do anything if you don't define any shorthands. We call these unobtrusive features. When you install the command library with the "Install Scripts" button, these unobtrusive automatic features are turned on.

Other automatic features, like smart arrow keys (see Chapter 7), make a fundamental change in the way HyperCard works. When you install the command library, these are left turned off. If you want them, you must turn them on yourself by clicking the "Enabled" box on their cards in Cookbook Reference.

Here's a list that shows the automatic features and tells whether they're enabled when you install the command library:

Number and Description	Name	Initial Status
A-1 Command library access	<code>commandTrigger</code>	enabled
2-3 Button script protection	<code>protectButton</code>	enabled
2-4 Automatic stack compaction	<code>autoCompact</code>	not enabled
2-8 Evaluate message box for shorthands	<code>expandMessageBox</code>	enabled
5-7 Bookmarks: save your place in a stack	<code>saveBookMark...</code>	not enabled
5-9 Create random visual effects	<code>randomVis</code>	not enabled

Number and Description	Name	Initial Status
5-10 Don't hide the menu bar if there's a big screen	bigScreenHide	enabled
5-12 Automatically move windoids	trackWindoids...	not enabled
7-3 Smart arrow key	smartArrowKey	not enabled
8-2 Search for names and nicknames	sloppyFind	not enabled

If you want to change the status of any of these features (as shown in the "initial status" column), just go to that script's card in the Cookbook Reference stack and set the "Enabled" box however you like.

Robin's Paint Box

This nifty stack, created by Robin Shank, shows off some useful and unusual techniques with the paint tools. It's sort of the painting equivalent of a command library. It includes a random "cowhide" border creator, a shape-drawing laboratory, and other fun stuff.

AutoPainter

Another stack by Robin Shank, which traces your drawing and records it in scripts.

Black Dog

Art à la HyperCard, created by Lynn Bekkala. Just click, baby.

HyperGlyphics

HyperGlyphics is Dan's bargain-basement data encryption system, featuring a pair of XFCNs written in Macintosh Programmer's Workshop C to do all the hard stuff. HyperGlyphics will take a text message and encode it so that no mere mortal can figure out what the heck it says. It can be decoded by anyone who has the XFCNs and knows the password (which is called the key).

An interesting feature of HyperGlyphics is that the key is encrypted along with the message. So, an attempt to decode the message with the wrong key brings up an alert that says the wrong key was used, rather than decoding the message into gibberish. It's not quite Pentagon-level security, but it is good

enough to play around with, and for many non-national security uses, as well. The source code for the XFCNs is listed in Appendix B.

HyperParrot

This stack puts together random sentences based on a vocabulary you provide (some sample words are included for your convenience). This will certainly amaze and amuse your friends, and it's great for parties. By the way, kids love this one.

Valentine from Doris

A HyperGreetingCard from Doris Mitsch.

Flowfazer™ Demo

This demo, from Todd Rundgren and David Levine, is a taste of Flowfazer, which Todd calls “music for the eye.” He goes on to say “Flowfazer is to the eye what music is to the ear. The effect on the brain is similar, so Flowfazer is good for relaxation, meditation, stress reduction, and purely esthetic enjoyment. It's not illegal yet.” (Flowfazer requires a color monitor and is available from Utopia Grokware, 300 Valley St., Suite 204, Sausalito, CA 94965.)

Disk-claimer

Because books take longer to make than disks do, we got to work on the disk after we were done working on the book. For this reason, you may find late updates and new information on the disk not described in the book. In these cases, we've written about them on the disk instead, since our goal is to provide the most complete, correct stuff we can.



COMMAND LIBRARY NOTES

When we introduced the idea of a command library in Chapter 1, we wanted to present it as cleanly as possible, with a minimum of clutter to help you absorb the concept. Now that you're fully absorbed, it's time to describe the architecture of the command library in more detail. We'll also discuss how the command trigger (shift-tab) script works, and tell you how to customize the way things work.

Command Library Architecture

The architecture of the command library is pretty simple. The Cookbook Reference stack has a handler named `initCookbook`. When you click the "Install Scripts" button in the Cookbook Reference stack, the button tries to install a statement that says

```
send "initCookbook" to stack "Cookbook Reference"
```

into the `getHomeInfo` handler in your home stack. If you don't have a `getHomeInfo` handler in your home stack, installation fails; otherwise, you're in business. Installing this one line makes the command library system work.

There's an `initCookbook` handler in the Cookbook Reference stack that kicks things off. The script for `initCookbook` is listed here.

```
on initCookbook
  start using stack "Cookbook Scripts 1"
  start using stack "Cookbook Scripts 2"
  start using stack "Cookbook Reference"
  initCookbookFeatures
end initCookbook
```

The first three lines use HyperCard 2.0's "start using stack" command to stick our command library stacks into the inheritance chain. This will allow our stacks to get a chance at any free messages before they're sent to the home stack.

That's how we'll get to handle commands. The fourth line calls yet another handler, `initCookbookFeatures`, which calls on any other command library features that need their own initialization.

Once the "start using stack" commands are done, anything you type into the message box that's not handled in the usual places (your own buttons, fields, cards, backgrounds, or stacks) will be sent to the command library, and we'll take care of it. The command library could be extended to any number of stacks, as long as they were inserted into the inheritance path with "start using stack" commands.

If you want to put your own commands into the command library, just add them to the stack script of Cookbook Scripts 1 or Cookbook Scripts 2. Or, if you'd like to keep your own scripts separate from ours, just create your own Cookbook Scripts stack, put your handlers into the stack's script, and add a "start using stack" command for that stack to `initCookbook`.

This is the kind of thing that "start using stack" was invented for—placing a bunch of commands in a common place and letting HyperCard find them. Without this command, we couldn't get away with such a simple and easy solution for our command library.

Automatic Features

In the "About the Stacks on the Disk" section, we talked about the command library's automatic features, the ones that you can turn on and off with check boxes in the Cookbook Reference stack. These things work a little bit differently from the rest of the command library. Here are the details.

Each automatic feature takes effect when a particular HyperCard-defined message is sent. For example, automatic stack compaction takes place when the `closeStack` message is sent. Each automatic feature's handler is called from a handler for the appropriate message (such as `closeStack`) that's located in Cookbook Reference's script. The features that are turned off are simply commented out by placing two dashes at the front of the line. Here's what the `arrowKey` handler in Cookbook Reference looks like:

```
on arrowKey
  --smartArrowKey
  --randomVis
```



```
pass arrowKey
end arrowKey
```

There are two automatic features that are triggered by the arrowKey message: smartArrowKey and randomVis. When you install the command library, they're both turned off by default, so their lines are commented out, as you see here. If you turn one of them on by clicking the "Enabled" button on its card in the Cookbook Reference, the comment dashes will be taken out and the feature will start working. Disabling a feature puts the comment dashes back, naturally.

Different Strokes

In Chapter 1, we said that HyperCard sends messages when the return, tab, enter, function (on the big keyboard), and arrow keys are pressed, and when the control key is pressed with another key. You can write handlers to catch these messages, of course, and in the handlers you can call any functions, such as optionKey, shiftKey, and commandKey, or test any other conditions you want. You can use any of these as your command trigger.

For example, you could prevent use of the command trigger on weekends if you install the variation listed here, although it wouldn't stop access through the message box or from scripts.

1. on tabKey
2. if the shiftKey is not down then pass tabKey
3. if item 1 of the abbr date is in "Sat Sun" then
 pass tabKey
4. ask "What can I do for you?"
5. if it is not empty
6. then do it
7. end tabKey

Here's an explanation of the pertinent lines in this script. Line 3 uses the HyperTalk function "date" to find out what today is. Like some other HyperTalk functions and properties, you can get information in a slightly different form by using "abbr" (HyperTalk also likes "abbrev" or "abbreviated"). For the date, the standard function returns just month, day, and year, but "the abbr date" adds the day of the week as the first item in a list, like this:

Sun, Dec 26, 1999

By asking for item 1 of the abbreviated date, we get the three-letter abbreviation for the day of the week. Asking if it's in "Sat Sun" is a quick way to find out if it's a weekend. An exercise for you: what happens if you try "word 1 of the abbr date" instead?

Another good choice for calling up the command library is one of the function keys on the Apple Extended Keyboard. To determine if you have an Apple Extended Keyboard, see if your keyboard is so large that it follows the curvature of the earth. If so, you're using the Apple Extended Keyboard. This keyboard has a row of function keys across the top (labeled f1, f2, and so on), most of which are probably not doing anything right now.

Normally, only the first four function keys are defined. By default, they're used as shortcuts for undo, cut, copy, and paste, but you can redefine them as shown below. You can use any of these function keys to trigger your command library. As an example, to make function key 5 get you to the command library, here's the script you'll need:

```
1.   on functionKey whichOne
2.       if whichOne is not 5 then pass functionKey
3.       ask "What can I do for you?"
4.       if it is not empty
5.           then do it
6.   end functionKey
```

This handler will be called if a function key is pressed. Line 2 makes sure that it's function key 5 (just change it for whichever function key you want). If you have an extended keyboard, this is probably the best solution for you. If you don't have the big keyboard, you'll probably want to stick with shift-tab, but the choice is entirely up to you.

The Trigger Script

And now, the moment that probably at least a few of you have been waiting for, the official command library trigger script.

```
1. on commandTrigger
2.   -- script A-1: Command library access.
3.
4.
```



```

5.    global prevTriggerCommand
6.    ask "What can I do for you?" with prevTriggerCommand
7.    if it is not empty then
8.        put it into prevTriggerCommand
9.        get expandShortHand(it)
10.       do it
11.    end if
12.    exit to HyperCard
13. end commandTrigger

```

If you remember the discussion of the original command trigger scripts in Chapter 1, some of this will look familiar to you. The biggest change is that we've factored out the check for the keystroke. That's now in a separate handler that calls this one, `commandTrigger`. In the Command Reference stack's script, there's a `tabKey` handler that checks for the shift key also, then calls `commandTrigger`. To change the command trigger, or add a second way to get to the command box, just write another handler that calls `commandTrigger`.

Lines 5 through 8 implement a nice convenience feature. They use a global called `prevTriggerCommand` to remember the last command used in the command box. When the library is summoned again via the command trigger, the last command is presented as the new default.

The user's command might include a shorthand, so the next step is to expand it through the shorthand system (line 9). Then, line 10 tells HyperCard to execute it.

Remember that there are two ways to call most of the commands in the command library: type them into the message box, or use a special keystroke (usually shift-tab) to bring up the command box and type the command there. The command box has two advantages: it doesn't lose the current selection, and it remembers the last command you typed into it. The message box's advantage is just that it's easy to use.

Miscellaneous Command Library System Notes

Be aware that shift-tab doesn't bring up the command box if the paint tools are chosen. It displays or removes the pattern window instead.

The command box is a handy way to try out HyperTalk's "selected" functions, like "put the selectedChunk." You can't use these functions from the message

box, because typing in the message box removes the selection, thus blowing away all the “selected” functions.

The script listings in this book were added to the manuscript by running a HyperTalk handler that wrote them out to disk files. Then Microsoft Word’s file-include feature read them in automatically. This “untouched by human hands” approach should help minimize mistakes in the book’s listings.

It’s interesting to remember that the inventor of this command library is also the guy who’s responsible for the future of HyperTalk (that’s Dan.) So, don’t be surprised if some of the command scripts and techniques in this book make their way directly into HyperTalk someday.



Appendix B

SOURCE CODE FOR EXTERNALS

This appendix contains the complete source code for the XCMDs and XFCNs that appear on the disk: wordWrap, which supports returnsIn; printPS, which lets you send PostScript commands to the printer; and encrypt and decrypt, which are used in the HyperGlyphics encoding stack.

WordWrap and printPS are written in Macintosh Programmer's Workshop (MPW) Pascal, and encrypt and decrypt are in MPW C. Encrypt and decrypt are in C because they involve lots of bitwise operations that are standard in C, although MPW Pascal has been extended to support these functions, too.

Changing these externals to work with other Pascal and C compilers should be easy, but we haven't tried it.

Source Code for printPS XCMD

(*

PrintPS

-- XCMD to send PostScript to the LaserWriter.

Sample HyperTalk line:

```
printPS "/Times-Roman findfont 12 scalefont setfont " & ¬  
"230 600 moveto (Hello World) show"
```

To compile and link this file using MPW:

```
directory "HardDisk:XCMD's:"
```

```
Pascal :Src:PrintPS.p -o :Obj:PrintPS.p.o
```

```
Link -rt XCMD=462 -m ENTRYPOINT -sg PrintPS -o 0
```

```
"XCMD's" :Obj:PrintPS.p.o "{Libraries}HyperXLib.o" 0
```


{PLibraries}PasLib.o {Libraries}Interface.o

*)

{ \$R- } { no range checking }
{ \$D- } { no Macsbug symbols }

UNIT DummyUnit;

INTERFACE

USES Types, Memory, QuickDraw, Errors, Events, Printing, HyperXCmd;

PROCEDURE EntryPoint (paramPtr: XCmdPtr);

IMPLEMENTATION

PROCEDURE PrintPS (paramPtr: XCmdPtr); FORWARD;

PROCEDURE EntryPoint (paramPtr: XCmdPtr);

BEGIN

PrintPS (paramPtr);

END;

PROCEDURE PrintPS (paramPtr: XCmdPtr);

VAR errStr: Str255;

PROCEDURE Abort (why: Str255);

BEGIN

paramPtr^.returnValue := PasToZero (paramPtr, why);

EXIT (PrintPS);

END;

PROCEDURE PrintPicture (pic: PicHandle; picRect: Rect);

LABEL 8;


```

VAR prRecHdl: THPrint;
    myPrPort: TPPrPort;
    myStRec: TPrStatus;
    dummy: EventRecord;
BEGIN
    prRecHdl := THPrint(NewHandle(SizeOf(TPrint)));
    IF prRecHdl = NIL THEN
        BEGIN
            errStr := 'Out of memory.';
            EXIT(PrintPicture);
        END;

    PrOpen;
    IF PrError <> noErr THEN
        BEGIN
            NumToStr(paramPtr, PrError, errStr);
            errStr := Concat('PrOpen failed. (' , errStr, ')');
            GOTO 8;
        END;

    PrintDefault(prRecHdl);
    IF PrError <> noErr THEN
        BEGIN
            NumToStr(paramPtr, PrError, errStr);
            errStr := Concat('PrintDefault failed. (' , errStr, ')');
            GOTO 8;
        END;

    IF PRValidate(prRecHdl) THEN;
    IF PrError <> noErr THEN
        BEGIN
            NumToStr(paramPtr, PrError, errStr);
            errStr := Concat('PRValidate failed. (' , errStr, ')');
            GOTO 8;
        END;

```



```

myPrPort := PrOpenDoc(prRecHdl,NIL,NIL);
IF PReError <> noErr THEN
    BEGIN
        NumToStr(paramPtr,PReError,errStr);
        errStr := Concat('PrOpenDoc failed. (' ,errStr,')');
        GOTO 8;
    END;

PrOpenPage(myPrPort,NIL);
IF PReError <> noErr THEN
    BEGIN
        NumToStr(paramPtr,PReError,errStr);
        errStr := Concat('PrOpenPage failed. (' ,errStr,')');
        GOTO 8;
    END;

DrawPicture(pic,picRect);

PrClosePage(myPrPort);
IF PReError <> noErr THEN
    BEGIN
        NumToStr(paramPtr,PReError,errStr);
        errStr := Concat('PrClosePage failed. (' ,errStr,')');
        GOTO 8;
    END;

PrCloseDoc(myPrPort);
IF PReError <> noErr THEN
    BEGIN
        NumToStr(paramPtr,PReError,errStr);
        errStr := Concat('PrCloseDoc failed. (' ,errStr,')');
        GOTO 8;
    END;

IF prRecHdl^^.prJob.bJDocLoop = bSpoolLoop
THEN PrPicFile(prRecHdl,NIL,NIL,NIL,myStRec);

```



```

IF PError <> noErr THEN
  BEGIN
    NumToStr(paramPtr,PError,errStr);
    errStr := Concat('PrPicFile failed. (' ,errStr,')');
    GOTO 8;
  END;

PrClose;
IF PError <> noErr THEN
  BEGIN
    NumToStr(paramPtr,PError,errStr);
    errStr := Concat('PrClose failed. (' ,errStr,')');
    GOTO 8;
  END;

8:
  DisposHandle(Handle(prRecHdl));
END;

PROCEDURE DoPrintPS(textHandle: Handle);
CONST PostScriptBegin = 190;
      PostScriptEnd = 191;
      PostScriptHandle = 192;
      TextIsPostScript = 194;
VAR textLen, charNum: LongInt;
    PSPicture: PicHandle;
    textFrame: Rect;
    textPtr: Ptr;
BEGIN
  textLen := StringLength(paramPtr,textHandle^);

  { translate returns into spaces, zero term into return }
  textPtr := textHandle^;
  FOR charNum := 1 TO textLen DO
    BEGIN

```



```

    IF textPtr^ = 13 THEN textPtr^ := ORD(' ');
    textPtr := Pointer(ORD(textPtr)+1);
END;
textPtr^ := 13;
textLen := textLen + 1;

SetRect(textFrame,-MAXINT,-MAXINT,MAXINT,MAXINT);
PSPicture := OpenPicture(textFrame);
  ClipRect(textFrame);
  MoveTo(20,20);
  DrawString(' '); { get around driver optimization }
  PicComment(PostScriptBegin,0,NIL);
  DrawString('This XCMD only works with PostScript
    printers.');
```

Printers.

```

  PicComment(TextIsPostScript,0,NIL);
  DrawString('initgraphics');
  PicComment(PostScriptHandle,textLen,textHandle);
  PicComment(PostScriptEnd,0,NIL);
ClosePicture;
PrintPicture(PSPicture,textFrame);
KillPicture(PSPicture);
END;

BEGIN
  IF paramPtr^.paramCount <> 1
  THEN Abort('Form is: PrintPS <PostScript>');
  errStr := '';
  DoPrintPS(paramPtr^.params[1]);
  IF errStr <> '' THEN Abort(errStr);
END;

END.
```


Source Code for wordWrap XFCN

(*

WordWrap -- XFCN to wrap text according to font,
size, and style.

Form: WordWrap(<text>, <width>, , <size>, <style>)

To compile and link this file using MPW:

```
directory "your disk:XCMD's:"
Pascal WordWrap.p
Link -rt XFCN=26 -m ENTRYPOINT -sg WordWrap 0
-o "Cookbook XCMD's" WordWrap.p.o 0
"{Libraries}HyperXLib.o" "{PLibraries}PasLib.o" 0
"{Libraries}Interface.o"
```

*)

```
{ $R- } { no range checking }
```

```
{ $D- } { no Macsbug symbols }
```

```
UNIT DummyUnit;
```

```
INTERFACE
```

```
USES Types, Memory, Resources, QuickDraw, Fonts, OSUtils,  
HyperXCmd;
```

```
PROCEDURE EntryPoint (paramPtr: XCmdPtr);
```

```
IMPLEMENTATION
```

```
PROCEDURE WordWrap (paramPtr: XCmdPtr); FORWARD;
```

```
PROCEDURE EntryPoint (paramPtr: XCmdPtr);
```

```
BEGIN
```

```
WordWrap (paramPtr);
```

```
END;
```

```
PROCEDURE WordWrap (paramPtr: XCmdPtr);
```

```
VAR wrapWidth: INTEGER;
```

```
PROCEDURE Abort (why: Str255);
```



```

BEGIN
    paramPtr^.returnValue := PasToZero(paramPtr,why);
    EXIT(WordWrap);

END;

PROCEDURE GetArgs;
VAR str: Str255;
    fontSize,fontNum: INTEGER;
    fontStyle: Style;
BEGIN
    WITH paramPtr^ DO
        BEGIN
            IF paramCount <> 5
            THEN Abort('Error: form is
WordWrap(<text>,<width>,<font>,<size>,<style>'));
            ZeroToPas(paramPtr,params[2]^,str);
            wrapWidth := StrToNum(paramPtr,str);
            IF wrapWidth <= 0 THEN Abort(Concat('Error: ',str,'
            is not a valid width'));
            ZeroToPas(paramPtr,params[3]^,str);
            GetFNum(str,fontNum);
            IF fontNum = 0
            THEN Abort(Concat('Error: can't find any font
            named ',str));
            ZeroToPas(paramPtr,params[4]^,str);
            fontSize := StrToNum(paramPtr,str);
            IF fontSize <= 0 THEN Abort(Concat('Error: ',str,'
            is not a valid font size'));
            fontStyle := [];
            IF StringMatch(paramPtr,'bold',params[5]^) <> NIL
            THEN fontStyle := fontStyle + [bold];
            IF StringMatch(paramPtr,'italic',params[5]^) <> NIL
            THEN fontStyle := fontStyle + [italic];
            IF StringMatch(paramPtr,'underline',params[5]^) <> NIL
            THEN fontStyle := fontStyle + [underline];
            IF StringMatch(paramPtr,'outline',params[5]^) <> NIL

```



```

THEN fontStyle := fontStyle + [outline];
IF StringMatch(paramPtr,'shadow',params[5]^) <> NIL
THEN fontStyle := fontStyle + [shadow];
IF StringMatch(paramPtr,'condense',params[5]^) <> NIL
THEN fontStyle := fontStyle + [condense];
IF StringMatch(paramPtr,'extend',params[5]^) <> NIL
THEN fontStyle := fontStyle + [extend];
END;

TextFont(fontNum);
TextFace(fontStyle);
TextSize(fontSize);
END;

PROCEDURE RemoveOldWraps;
{ turn returns into spaces in place }
VAR p: Ptr;
BEGIN
  p := paramPtr^.params[1]^;
  WHILE p^ <> 0 DO
    BEGIN
      IF p^ = 13 THEN p^ := ORD(' ');
      p := Pointer(ORD(p)+1);
    END;
  END;
END;

PROCEDURE ScanToSpace(VAR p: Ptr);
BEGIN
  IF p^ = ORD(' ')
  THEN p := Pointer(ORD(p)+1); { already at a space,
    step past it }
  WHILE (p^ <> ORD(' ')) AND (p^ <> 0)
  DO p := Pointer(ORD(p)+1);
END;

PROCEDURE WrapText;
{ wrap text in place then copy to return value }
VAR linePtr,spacePtr,prevSpacePtr: Ptr;

```



```

BEGIN
  linePtr := paramPtr^.params[1]^;
  prevSpacePtr := linePtr;
  spacePtr := linePtr;
  WHILE linePtr^ <> 0 DO
    BEGIN
      linePtr := prevSpacePtr;
      prevSpacePtr := spacePtr;
      ScanToSpace(spacePtr);

      WHILE (TextWidth(linePtr,0,ORD(spacePtr)-
        ORD(linePtr)) < wrapWidth)
        AND (prevSpacePtr^ <> 0) DO
        BEGIN
          prevSpacePtr := spacePtr;
          ScanToSpace(spacePtr);
        END;

      IF prevSpacePtr^ = ORD(' ') THEN prevSpacePtr^ := 13;
    END;

    paramPtr^.returnValue := paramPtr^.params[1];
    IF HandToHand(paramPtr^.returnValue) <> 0
      THEN Abort('Error: out of memory');
  END;
END;

BEGIN
  GetArgs;
  RemoveOldWraps;
  WrapText;
END;

END.

```


Source Code for Encrypt XFCN

```

/*
 * Encrypt - XFCN to do one time pad encryption
 *
 * Sample HyperTalk line:
 *
 * put Encrypt(key,field 1) into field 2 -- encrypt field
 *   1 with key
 *
 * Build commands:
 *
 * C Encrypt.c -b -mbg off
 * Link -w -rt XFCN=12 -m MAIN -sg Encrypt -o "Cheap-O-
 *   Crypt" @
 *   Encrypt.c.o "{Libraries}HyperXLib.o"
 *   "{Libraries}Interface.o" @
 *   "{CLibraries}CRuntime.o"
 *
 */

#include <Types.h>
#include <Memory.h>
#include <HyperXCmd.h>

#define Abort(result) { paramPtr->returnValue =
    PasToZero(paramPtr,result); return; }
#define ParamStr(num) *paramPtr->params[num]

pascal void main(XCmdPtr paramPtr)
{
    Handle secret;
    char *key,*msg,*lastMsg,*out;
    long msgLen,keyHash;

    if (paramPtr->paramCount != 2) Abort("\pError: Form is
Encrypt(key,string)\n");

```



```

if (!*ParamStr(0)) Abort("\pError: Key must not be
    empty\n");
if (!*ParamStr(1)) return; /* message is empty so return
    empty */

/* append keyHash to message so decrypt can recognize
    valid messages */
keyHash = 0;
for (key = ParamStr(0); *key; key++) keyHash += *key;
msgLen = StringLength(paramPtr, ParamStr(1)) + 1;
SetHandleSize(paramPtr->params[1], msgLen+4);
for (lastMsg = ParamStr(1); *lastMsg; lastMsg++);
+++lastMsg = (keyHash >> 24) & 0x000000FF;
+++lastMsg = (keyHash >> 16) & 0x000000FF;
+++lastMsg = (keyHash >> 8) & 0x000000FF;
+++lastMsg = (keyHash >> 0) & 0x000000FF;

secret = NewHandle((msgLen + 4) * 4 / 3 + 4);
if (!secret) Abort("\pError: Not enough memory to
    encrypt\n");

    /* do the one time pad on the message in place */
for (msg = ParamStr(1), key = ParamStr(0); msg <=
    lastMsg; msg++, key++)
{
    if (!*key) key = ParamStr(0);
    *msg = *msg ^ *key;
}

/* now expand encrypted string into printable text */
for (msg = ParamStr(1), out = *secret; msg <= lastMsg;
    msg += 3, out += 4)
{
    out[0] = '!' + ((msg[0] >> 2) & 0x3F);
    out[1] = '!' + ((msg[0] << 4) & 0x30 | (msg[1] >> 4) &
        0x0F);
    out[2] = '!' + ((msg[1] << 2) & 0x3C | (msg[2] >> 6) &
        0x03);
}

```



```

    out[3] = '!' + (msg[2] & 0x3F);
}
*out = '\0'; /* zero terminate */
paramPtr->returnValue = secret;
}
/* End of Encrypt */

```

Source Code for Decrypt XFCN

```

/*
 * Decrypt - XFCN to do one time pad decryption
 *
 * Sample HyperTalk line:
 *
 * put Decrypt(key,field 1) into field 2 -- decrypt field
 *   1 with key
 *
 * Build commands:
 *
 * C Decrypt.c -b -mbg off
 * Link -w -rt XFCN=13 -m MAIN -sg Decrypt -o "Cheap-O-
 *   Crypt" @
 *   Decrypt.c.o "{Libraries}HyperXLib.o"
 *   "{Libraries}Interface.o" @ "{CLibraries}CRuntime.o"
 *
 *
 */

#include <Types.h>
#include <Memory.h>
#include <HyperXCmd.h>

#define Abort(result) { paramPtr->returnValue =
    PasToZero(paramPtr,result); return; }
#define ParamStr(num) *paramPtr->params[num]
pascal void main(XCmdPtr paramPtr)
{

```



```

Handle clearText;
char *key,*msg,*lastMsg,*secret,*lastSecret;
long keyHash,checkHash;

if (paramPtr->paramCount != 2) Abort("\pError: Form is
Decrypt(key,string)\n");
if (!*ParamStr(0)) Abort("\pError: Key must not be
empty\n");
if (!*ParamStr(1)) return; /* message is empty so return
empty */

clearText =NewHandle((StringLength(paramPtr,ParamStr(1))
+ 1) * 3 / 4 + 3);
if (!clearText) Abort("\pError: Not enough memory to
decrypt\n");

/* pack the message back into binary */
for (lastSecret = ParamStr(1); *lastSecret; lastSecret++);
for (secret = ParamStr(1), msg = *clearText; secret <
lastSecret; secret += 4, msg += 3)
{
    int i;
    for (i = 0; i < 4; i++) secret[i] = (secret[i] - '!') &
0x3F; /* remove base character */
    msg[0] = secret[0] << 2 | secret[1] >> 4;
    msg[1] = secret[1] << 4 | secret[2] >> 2;
    msg[2] = secret[2] << 6 | secret[3];
}
lastMsg = msg + 3;

/* do the one time pad on the message in place */
key = ParamStr(0);
for (msg = *clearText; msg < lastMsg; msg++)
{
    *msg = *msg ^ *key;
    key++;
    if (!*key) key = ParamStr(0);
}

```



```
/* check for keyHash */
keyHash = 0;
for (key = ParamStr(0); *key; key++) keyHash += *key;
for (msg = *clearText; *msg; msg++);
msg++;
checkHash = 0;
checkHash |= (*msg++ << 24) & 0xFF000000;
checkHash |= (*msg++ << 16) & 0x00FF0000;
checkHash |= (*msg++ << 8) & 0x0000FF00;
checkHash |= (*msg++ << 0) & 0x000000FF;
if (checkHash != keyHash)
{
    DisposHandle(clearText);
    Abort("\pError: wrong key\n");
}
paramPtr->returnValue = clearText;
}
/* End of Decrypt */
```


INDEX

A

Alignment

- alignParts, 99-105

- properties of group of parts, 98-105

- Array of items, mixed list

 - converted to, 231-233

- Array of lines, mixed list converted

 - to, 233-235

- ArrowKey, 272-273

- ArrowKeyhandler, 151

- Ask dialog box, 9

- AutoCompact Enable, 35-36

- Automatic scripting, 18-30

 - for installer scripts, 20-30

 - quoteLines, 24-30

 - sourceScript, 21

 - usefulness of, 18

 - writeScript, 20-24

- Automatic stack compaction, 34-36

- AutoPainter, 269

B

Background

- action on each button in

 - background, 124-127

- action on each card in

 - background, 127-129

- action on each field in

 - background, 122-124

- clearPicture, 244-245

- erasing background picture,

 - 243-245

Backgrounds

- action on all backgrounds,

 - 115-116

- group selection of, 84-87

- modes of HyperCard, 96

- Bar chart, creation of, 249-253

- BigScreenHide, 153-154

- BkgndOnly, 123, 124, 125, 126

- Black Dog, 269

- BlindTyping, 65

- Bookmarks, 144-149

 - error handling and, 149

 - GoToBookmark, 145-146

 - note not found, 146-147

 - popMark, 148-149

 - pushMark, 148-149

 - SaveBookmark, 145

 - Boolean function, 97

Broadcasters

- action on each background in

 - stack, 121-122

- action on each button in card,

 - 124-127

- action on each card in back ground, 127-129
- action on each card in stack, 119-121
- action on each field in card, 122-124
- doToEachBkgnd, 121-123
- doToEachButton, 126-127
- doToEachCard, 120-121
- doToEachCardofBkgnd, 127-128
- doToEachField, 123-126
- usefulness of, 119
- Buffers, keystrokes, 43
- Buttons, 90
 - action on each button in card, 124-127
 - properties of, 101
 - protect Button, deletion protection, 31-34
 - reportHidden, 179
 - reportHiddenButtons, 174-176
 - showAllButtons, 169-171, 175
 - showButtons, 179-182

C

- Calendar, 237-243
 - calendar function, 237-243
 - font for, 238
- Card fields
 - note field, creation of, 36-44
 - usefulness of, 36
- CardLoc, 163
- CardNum, 146
- CardOnly, 123, 124, 125, 126

Cards

- action on all cards in group, 112-114
- action on each button in card, 124-127
- action on each card in background, 127-129
- action on each card in stack, 119-121
- action on each field in card, 122-124
- card specifier, use of, 95
- copy from one stack to another, 131-133
- copyTo, 132-133
- dupCard, 135-136
- duplicate cards, 134-135
- group selection of, 80-84
- info, 186
- marked cards, 80-81
- move to another stack, 133-134
- moveTo, 134-135
- new card as first card of stack, 135-137
- newFirst, 136-137
- quick information about, 185-186
- Card window, 184
- CenterOfRect, 246
- Charts
 - bar chart, 249-253
 - line chart, 253-257
 - pie chart, 257-263
- CheckWindoid, 166
- Chunk separators, 251
- Chunks of text, handling of, 231
- Clear, 86, 91, 96

- ClearButton, 90
- Clear Field, 90
- ClearPicture, 244-245
- ClearSelection, 83
- ClearSelectionFlag, 83, 86
- CloseCard, 76
- CloseField, 43, 56-57
- CloseStack, 35-36, 145, 158, 159
- Command box, 11
 - advantages to use, 275-276
 - memory of, 82
- Command-click, 72
- Command library, 7-8
 - architecture of, 271-272
 - automatic features, 268-269
 - automatic features of, 272-273
 - calling commands in, 275
 - command trigger, 12
- Cookbook Reference,
 - 267-268, 272
- on disk, 267-268
- getting into, 13
- me, 128-129
- putting command in, 12-14
- trigger script, 274-275
- usefulness of, 17-18, 273-274

- Command scripts, 8
- CommandTrigger, 275
- Comment delimiter, 138, 139, 176
- Compaction
 - automatic, 34-36
 - closeStack, 35-36
- Compare, text in two fields, 205-207
- Convert, dates, 239-240
- Cookbook reference stack, 13
- Coordinate plane, 227

- Coordinates
 - mouse, display of, 182-185
 - paint, conversion to global
 - coordinates, 230-231
 - rectangle, conversion to local
 - coordinates, 228-230
- Coordinate systems
 - global coordinate system, 227
 - local coordinate system, 228-229
- Copy, 95, 96
- CopyTo, 132-133

D

- Dates, convert, 239-240
- Debugger, use with scripts, 27, 28-29
- Decompression, in shorthands, 46-51
- Decrypt XFCN, source code for,
 - 289-291
- Defaults, 79
- DeleteParts, 88-93
- Deletion
 - execution of delete command,
 - 33-34
 - group of parts, 87-93
 - protect Button, 31-34
- DeselectFlag, 73-75
- DoMenu, 90
- DoneSelecting, 76
- DoToEachBkgnd, 121-123
- DoToEachButton, 126-127
- DoToEachCard, 120-121
- DoToEachCardofBkgnd, 127-128
- DoToEachField, 123-126
- DoToSelectedBkgnds, 115-116
- DoToSelectedCards, 112-114

Drag command, 245
 DupCard, 135-136

E

Edit Bkgnd, 96, 97, 98
 Else clause, 33
 EmptyCat variable, 29, 30
 Empty field, 44
 Encryption, HyperGlyphics, 269-270
 Encrypt XFCN, source code for, 287-289
 End, 58
 Endchar, 60, 141
 EnterInField, 43
 Exit, 33
 Exit to HyperCard, 75
 Expansion, in shorthands, 46-51

F

FieldName, 197
 Fields, 90
 action on each field in card, 122-124
 properties of, 101
 reportHidden, 179
 reportHiddenFields, 176, 177-178
 showAllFields, 171-172
 for text storage, 138
 Find
 messages, 210-211
 names, 214-215
 Find and replace, 215-224
 find selection, 223-224
 global, 216-218

 in one field, 220-221
 with prompting, 218-220
 replace, 217
 replacefield, 220-222
 replacePrompt, 219
 replaceSelection, 222-223
 replace with something else, 221-223

Floating windows. See Windoids
 Flowfazer Demo, 270
 FoundChunk, 218, 220
 Function, 58
 Function handlers, 60

G

GetHandler, 59-61
 GetHomeInfo, 55, 65-66
 global properties of HyperCard, 65
 global variables, 66
 usefulness of, 65, 66
 GetMBarHeight, 159
 GetNote, 140-142, 146, 147
 Global coordinate system, 227
 GlobalPoint, 230
 Global properties, user level, 65
 Global settings
 menu bar, 157
 message box, 158-159
 palettes, 157
 Global variables, GetHomeInfo, 66
 Go, 101
 GoToBookmark, 145-146
 Graphics
 coordinate systems and, 227-228

paint, conversion to global
 coordinates, 230-231
 rectangle, conversion to local
 coordinates, 228-230
 rectangles, changing size of,
 235-237
 See also Charts.
 Groups
 actions of
 action on all backgrounds,
 115-116
 action on all cards in group,
 112-114
 aligning properties of group of
 parts, 98-105
 alignParts, 99-105
 deleteParts, 88-93
 deletion of group of parts, 87-93
 doToSelectedBkgnds, 115-116
 doToSelectedCards, 112-114
 offset group of parts, 105-111
 offsetParts, 106-109
 partPropSpread, 109-112
 pasteParts, 94-98
 pasting group of parts, 93-98
 selection of, 69-87
 group selection of backgrounds,
 84-87
 group selection of cards, 80-84
 by rectangular specification, 76-80
 rectSelectParts, 78-80
 selectCard, 82-84
 selectParts, 72-76
 Guide, 99

H

HandlerName, 59, 62
 Handlers
 beginning/end for, 58
 broadcasters, 119-129
 definition of, 1, 58
 function handlers, 60
 invention for new function, 2-7
 message handlers, 60
 message name and, 3
 parameters and, 83, 85
 Handler in script
 extraction of, 58-60
 getHandler, 59-61
 help, 62-64
 placement of, 60-62
 PutHandler, 61-62
 Hard-coding, disadvantage of, 30
 Help
 enhancement of, 62-64
 methods for, 63
 Hidden fields, 52
 Hidden objects
 reportHidden, 179
 reportHiddenButtons, 174-176
 reportHiddenFields, 176, 177-178
 Hiding menubar, 153, 157
 HyperCard
 descriptions of, xii-xxiii
 GetHome Info, 65-66
 “new” commands, 42
 HyperGlyphics, 269-270
 HyperParrot, 270
 HyperTalk
 keystrokes, special, 10-12

line numbers, 10
translator, 7

I

ID

ID form, 74
long ID, 84, 86-87, 149

IdleTime, 92

Indentation

automatic, field script, 263-265
returnInField, 264-265
script style, 189-190
shiftLeft, 200
shiftRight, 200-202

Indentation spaces, 30

Info, 186

Inheritance chain, 210

hierarchy for, 10

start using stack command and, 9

InitShorthands, 53-54

InitWindoidTracking, 160-164

Insertion points, smartArrowKey,
195-196

InsetRect, 236-237

Installer scripts, automatic
generation of, 20-30

Is within operator, 79

Items

itemArray, 232-233
mixed list converted to array of,
231-233

It variable, 25

K

Keystrokes

buffers, 43
special key strokes, 10-12
changing keystrokes, 12

L

Left, shifting lines in selection to,
200

Line chart, creation of, 253-257

Line numbers, 10

Lines, lineArray, 233-235

Lists

mixed list converted to array of
lines, 233-235
sum of list of numbers, 246-248

Local coordinate system, 228-229

LocalRect, 228-230

Location, meaning of, 80, 81

LockScreen, 92

M

MacFreeSizeOfStack, 35-36

MakeMeLarger, 43, 44

MakeMeSmaller, 43, 44

Manufactured script, 21

Marked cards, 80-81

Me, 128-129

Menu bar

bigScreenHide, 153-154
checking visibility of, 157, 159
displaying on big screen, 152-154
hiding menubar, 153, 157

Message box, 69
 expandMessageBox, 51-52
 showing vs. hiding, 158
 Message handlers, 60
 Messages
 find, 210-211
 post-process find message, 209-212
 Modifier keys, 74
 Mouse
 display mouse coordinates,
 182-185
 xy, 183-185
 MouseDown, 43, 73, 75
 MouseLoc, 184, 185
 MouseUp, 43, 75, 155-156
 MoveTo, 134-135

N

Names
 find, 214-215
 search for, 213-215
 "New" commands, 42
 NewFirst, 136-137
 NewSelection, 222
 No such card error message, 54
 Note contents, 143
 Note field
 closeField, 43
 creation of, 36-44
 empty field, 44
 enterInField, 43
 expansion of, 40-41
 forms of, 40
 MakeMeLarger, 43, 44
 MakeMeSmaller, 43, 44

newNote, 40-41, 44
 skinking field, 44
 NoteName, 140, 141, 143
 Notes
 putting into scripts, 142-143
 retrieving from scripts, 139-142
 Number function, 120, 124
 NumberLines, 191-193
 Numbers
 at beginning of text line, 190-194
 sum, 247-248

O

Object center, selection of, 81
 ObjName, 177
 Offset
 of function, 59-60
 variable versus function, 108
 OffsetParts, 106-109
 On, 58
 OpenStack, 145, 158, 159

P

Paint, conversion to global
 coordinates, 230-231
 Palettes, leaving open, 157
 Parameters
 param, 83
 paramCount, 83
 parameter passing system, 85
 params, 83, 211, 212
 undeclared parameters, 83
 PartPropSpread, 109-112
 Parts

- definition of, 69
- types of, 90
- See also* Groups.
- Password, for stacks, 18
- PasteParts, 94-98, 100
- Pie chart, creation of, 257-263
- PlainTextFlag, 48
- Points, globalPoint, 230
- PopMark, 148-149
- Post-process method, messages, 209-212
- PowerKeys, 65
- PrefName, 166
- PrevTrigger Command, 275
- PrintPS XCMD, source code for, 277-282
- Private Access, 18
- Prop, 106
- Property
 - masterProp, 104, 109
 - setProp, 104
 - short version, 103-104
- Protect Button, 31-34
- Protect Stack, 18
- Push card command, 96
- PushMark, 148-149
- PutHandler, 61-62
- PutNote, 142-143
- Puttable format, scripts, 19-20, 21, 22, 24

Q

- QuoteLines, 24-30

R

- Radio buttons
 - handling groups of, 155-156
 - MouseUp, 155-156
 - shortcomings related to, 154-155
- RandomVis, 151-152
- Recovery of text, 196-199
 - rescueText, 197-199
- Rectangles
 - calculation of center, 245-246
 - centerOfRect, 246
 - changing size of, 235-237
 - conversion to local coordinates, 228-230
 - insetRect, 236-237
 - localRect, 228-230
 - selection of part, 76-80
- RectSelectParts, 78-80
- Replace, 217
- Replacefield, 220-222
- ReplacePrompt, 219
- ReplaceSelection, 222-223
- ReportHidden, 179
- ReportHiddenButtons, 174-176
- Resume, 66
- ReturnInField, 264-265
- ReturnOffsets, 140
- Returns
 - adding to text, 203-206
 - deletion of returns, 202-203
 - returnsIn, 204-205
 - returnsOut, 202-203
- Right, shifting lines in selection to, 200-202
- Robin's Paint Box, 269

S

SaveBookmark, 145

SaveCardScript, 72

SaveShorthands, 54-56

Screen locking, 55, 92

Scripts

automatic scripting, 18-30

for installer scripts, 20-30

usefulness of, 18

command library for, 7-8

command scripts, 8

content delimiter, use of, 138, 139

debugger used with, 27, 28-29

definition of, 1

getNote, 140-142

indentScript, 189-190

installation into another object,
19

manufactured script, 21

putNote, 142-143

puttable format, 19-20, 21, 22, 24

putting note into, 142-143

retrieving note from, 139-142

script to write scripts, write script,
21-30

subroutine scripts, 8

ScriptText, 59, 140, 141, 143

Search

find and replace, 215-224

for names/nicknames, 213-215

SelectCard, 82-84

SelectedChunk, 223

SelectedPartsCard, 89

Select empty message, 33

SelectParts, 72-76

Send keyword, 211

SetProp, 104

SetStmnt, 176-177

Settings, saving, 100

SetUserLevel, 65

Shift-click, 72

ShiftKeyfunction, 74

ShiftLeft, 200

ShiftRight, 200-202

Shorthand

closeField, 56-57

defining new shorthand, 45-46

evaluation of message box for, 51

expanding shorthand, 46-51

initShorthands, 53-54

parameters of command, 46

permanent shorthands, 52-56

SaveShorthands, 54-56

setting up shorthand words, 45-58

ShowAllButtons, 169-171, 175

ShowAllFields, 171-172

ShowButtons, 179-182

ShowEverything, 173-174

ShowStmnt, 176

SmartArrowKey, 195-196

Source, in shorthands, 46

Source code

decrypt XFCN, 289-291

encrypt XFCN, 287-289

printPS XCMD, 277-282

wordWrap XFCN, 283-286

SourceScript, 21

Stacks

action on each card in stack,
119-121

- automatic stack compaction, 34-36
- bookmarks, 144-149
- info, 186
- on the disk, 267-270
- password for, 18
- quick information about, 185-186
- Startchar, 60, 141
- StartUp, 66
- Start using stack command, 9
- Statement
 - setStmnt, 176-177
 - ShowStmnt, 176
- StripNumber, 193-194
- Subroutine scripts, 8
- Sum, list of numbers, 246-248

T

- Target function, 73, 123, 127
- TextArrows, 65-66, 195, 196
- Text processing
 - adding returns, 203-205
 - arrows moving of insertion point, 195-196
 - comparing text in two fields, 205-207
 - deletion of returns, 202-203
 - indentation
 - script style, 189-190
 - shiftLeft, 200
 - shiftRight, 200-202
 - numbers at beginning of lines, 190-194
 - text recovery, 196-199
- TrackWindoid, 164-165

- Translator, 7
- Trigger script, command library, 274-275

U

- UnNumberLines, 191-192, 193
- User level, global properties, 65
- UserName, 66
- UserPreferences, 65

V

- Valentine from Doris, 270
- Value statement, 166
- Visual effects
 - creating random effects, 150-152
 - randomVis, 151-152

W

- Wait command, 6
- WhatToDo, 116
- Windoids, 159-166
 - automatic movement of, 160-166
 - checkWindoid, 166
 - initWindoidTracking, 160-164
 - trackWindoid, 164-165
 - updateWindoids, 164-165
 - use of, 159
 - windoid preferences card, 163
- WorkWrap, 205
 - XFCN, source code for, 283-286
- WriteScript, 20-24
- Write script, script to write scripts, 21-30

X

XCMD, checking menubar visibility,

157, 159

X-coordinate, 246

XFCN, 205

checking menubar visibility,

157, 159

WordWrap source code, 283-286

Xy, 183-185

Y

Y-coordinate, 246

